OhHelp Library Package for Scalable Domain-Decomposed PIC Simulation*

Hiroshi Nakashima
(ACCMS, Kyoto University)

2015/10/23

Abstract

This document describes the usage of a C-code library package named *OhHelp* for domain-decoposed Particle-in-Cell (PIC) simulations. The library has the following three layers. Level-1 code provides a load-balancer function which examines whether particles are distriuted among computation nodes (MPI processes) in a well-balanced manner, reforms the configuration of particle assignment to each node if necessary, and tells you how to move particles among nodes. In Level-2 code, the load balancer function is also capable to move particles among nodes by MPI functions for you. In addition, Level-3 code has vairous useful functions for domain-decomposed simulations such as for exchanging boundary values of electromagnetic fields associated to decopmposed subdomain. Furthermore, the library has two types of extensions, Level-4p and Level-4s, in which the load balancing mechanism takes care of particle positions so that all particles in a grid-voxel are accommodated by a particular node, to implement, e.g., Monte Carlo Collision with the former and Smoothed Particle Hydrodynamics (SPH) method with the latter.

This document also describes the implementation details of the OhHelp library showing every line of each source file. Since the source files are extracted from this file, the descriptions and explanations perfectly corresponds to the real implemenatation with which you play for your own PIC simulation.

*This file has version number v1.1.1, last revised 2015/10/23.

Contents

1 Introduction **10**

2 OhHelp Algorithm **11**
2.1 Overview and Definitions . . . 11
2.2 Secondary Subdomain Assignment . . . 12
2.3 Checking and Keeping Local Balancing . . . 13

3 OhHelp Library **17**
3.1 Library Layers . . . 17
3.2 Applying OhHelp to PIC Simulators . . . 18
3.2.1 Duplication of Data Structures . . . 19
3.2.2 Duplication of Computation . . . 21
3.2.3 Addition of Collective Communications . . . 22
3.2.4 Attachment of Load Balancer . . . 23
3.3 Configuration: Dimension of Simulated Space and Library Level . . . 25
3.4 Level-1 Library Functions . . . 26
3.4.1 `oh1_init()` . . . 27
3.4.2 `oh1_neighbors()` . . . 33
3.4.3 `oh1_families()` . . . 34
3.4.4 `oh1_transbound()` . . . 36
3.4.5 `oh1_accom_mode()` . . . 37
3.4.6 `oh1_broadcast()` . . . 38
3.4.7 `oh1_all_reduce()` . . . 39
3.4.8 `oh1_reduce()` . . . 40
3.5 Level-2 Library Functions . . . 41
3.5.1 Particle Data Type . . . 42
3.5.2 `oh2_init()` . . . 43
3.5.3 `oh2_max_local_particles()` . . . 45
3.5.4 `oh2_transbound()` . . . 45
3.5.5 `oh2_inject_particle()` . . . 46
3.5.6 `oh2_remap_injected_particle()` . . . 46
3.5.7 `oh2_remove_injected_particle()` . . . 47
3.5.8 `oh2_set_total_particles()` . . . 47
3.6 Level-3 Library Functions . . . 48
3.6.1 `oh3_init()` . . . 49
3.6.2 `oh13_init()` . . . 62
3.6.3 `oh3_grid_size()` . . . 64
3.6.4 `oh3_transbound()` . . . 65
3.6.5 `oh3_map_particle_to_neighbor()` . . . 65
3.6.6 `oh3_map_particle_to_subdomain()` . . . 67
3.6.7 `oh3_bcast_field()` . . . 68
3.6.8 `oh3_allreduce_field()` . . . 69
3.6.9 `oh3_reduce_field()` . . . 70
3.6.10 `oh3_exchange_borders()` . . . 70
3.7 Level-4p Extension and Its Functions . . . 72
3.7.1 Position-Aware Particle Management . . . 72
3.7.2 Level-4p Functions . . . 74
3.7.3 `oh4p_init()` . . . 75

3.7.4 oh4p_max_local_particles() . . . 77
3.7.5 oh4p_per_grid_histogram() . . . 78
3.7.6 oh4p_transbound() . . . 79
3.7.7 oh4p_map_particle_to_neighbor() . . . 79
3.7.8 oh4p_map_particle_to_subdomain() . . . 81
3.7.9 oh4p_inject_particle() . . . 81
3.7.10 oh4p_remove_mapped_particle() . . . 82
3.7.11 oh4p_remap_particle_to_neighbor() . . . 83
3.7.12 oh4p_remap_particle_to_subdomain() . . . 83
3.8 Level-4s Extension and Its Functions . . . 84
3.8.1 Position-Aware Particle Management in Level-4s . . . 84
3.8.2 Level-4s Functions . . . 86
3.8.3 oh4s_init() . . . 87
3.8.4 oh4s_particle_buffer() . . . 90
3.8.5 oh4s_per_grid_histogram() . . . 91
3.8.6 oh4s_transbound() . . . 92
3.8.7 oh4s_exchange_border_data() . . . 92
3.8.8 oh4s_map_particle_to_neighbor() . . . 93
3.8.9 oh4s_map_particle_to_subdomain() . . . 94
3.8.10 oh4s_inject_particle() . . . 94
3.8.11 oh4s_remove_mapped_particle() . . . 94
3.8.12 oh4s_remap_particle_to_neighbor() . . . 95
3.8.13 oh4s_remap_particle_to_subdomain() . . . 95
3.9 Particle Injection and Removal . . . 96
3.9.1 Level-1 Injection and Removal . . . 96
3.9.2 Level-2 (and 3) Injection and Removal . . . 96
3.9.3 Level-4p and 4s Injection and Removal . . . 97
3.9.4 Identification of Injected Particles . . . 98
3.10 Statistics . . . 98
3.10.1 Timing Statistics Keys and Header File oh_stats.h . . . 99
3.10.2 Arguments of oh1_init() for Statistics . . . 101
3.10.3 oh1_init_stats() . . . 101
3.10.4 oh1_stats_time() . . . 102
3.10.5 oh1_show_stats() . . . 102
3.10.6 oh1_print_stats() . . . 103
3.11 Verbose Messaging . . . 105
3.12 Aliases of Functions . . . 106
3.13 Sample Code . . . 106
3.13.1 Fortran Sample Code . . . 108
3.13.2 C Sample Code . . . 119
3.14 How to make . . . 128

4 Implementation 130
4.1 Naming Convention . . . 130
4.2 Header File ohhelp1.h . . . 132
4.2.1 Header File Inclusion . . . 132
4.2.2 Constants and Shorthands . . . 132
4.2.3 Basic Process Configuration Variables . . . 134
4.2.4 Particle Histograms . . . 136
4.2.5 Node Descriptors . . . 139

4.2.6 Heap Structures for Rebalancing . . . 141
4.2.7 Variables for Particle Transfer Scheduling . . . 142
4.2.8 Variables for Family Communicators . . . 145
4.2.9 Variables for Neighboring Information . . . 146
4.2.10 Variables for Statistics and Verbose Messaging . . . 147
4.2.11 Function Prototypes . . . 152
4.2.12 Macro `Verbose()` . . . 155
4.3 C Source File ohhelp1.c . . . 157
4.3.1 Header File Inclusion . . . 157
4.3.2 Function Prototypes . . . 157
4.3.3 `oh1_init()` and `init1()` . . . 158
4.3.4 `mem_alloc()` . . . 167
4.3.5 `mem_alloc_error()` . . . 167
4.3.6 `errstop()` and `local_errstop()` . . . 168
4.3.7 `oh1_neighbors()` . . . 168
4.3.8 `oh1_families()` . . . 169
4.3.9 `set_total_particles()` . . . 170
4.3.10 `oh1_transbound()` and `transbound1()` . . . 171
4.3.11 `try_primary1()` . . . 175
4.3.12 Macro `Special_Pexc_Sched()` . . . 176
4.3.13 `try_stable1()` . . . 176
4.3.14 `count_stay()` . . . 184
4.3.15 `assign_particles()` . . . 185
4.3.16 `compare_int()` . . . 188
4.3.17 `schedule_particle_exchange()` . . . 189
4.3.18 `count_real_stay()` . . . 193
4.3.19 `sched_comm()` . . . 193
4.3.20 `make_comm_count()` . . . 197
4.3.21 `make_recv_count()` . . . 200
4.3.22 `make_send_count()` . . . 200
4.3.23 `count_next_particles()` . . . 201
4.3.24 `oh1_broadcast()` . . . 201
4.3.25 `rebalance1()` . . . 203
4.3.26 `build_new_comm()` . . . 206
4.3.27 `push_heap()` . . . 209
4.3.28 `pop_heap()` . . . 210
4.3.29 `remove_heap()` . . . 210
4.3.30 `oh1_accom_mode()` . . . 211
4.3.31 `oh1_all_reduce()` . . . 212
4.3.32 `oh1_reduce()` . . . 213
4.3.33 `oh1_init_stats()` . . . 214
4.3.34 `clear_stats()` . . . 215
4.3.35 `oh1_stats_time()` . . . 215
4.3.36 `stats_primary_comm()` . . . 216
4.3.37 `stats_secondary_comm()` . . . 216
4.3.38 `stats_comm()` . . . 217
4.3.39 `oh1_show_stats()` . . . 218
4.3.40 Macro `Round()` . . . 219
4.3.41 `update_stats()` . . . 219
4.3.42 Macro `Stats_Reduce_Part_{Min,Max,Sum}()` . . . 221

4.3.43 stats_reduce_part() . . . 221
4.3.44 print_stats() . . . 222
4.3.45 stats_reduce_time() . . . 223
4.3.46 oh1_print_stats() . . . 223
4.3.47 oh1_verbose() . . . 223
4.3.48 Macros Vprint() and Vprint_Norank() . . . 224
4.3.49 vprint() . . . 224
4.3.50 dprint() . . . 225
4.4 Header File ohhelp2.h . . . 226
4.4.1 Header File Inclusion . . . 226
4.4.2 Particle Buffers and Related Variables . . . 226
4.4.3 Macro Particle_Spec() . . . 229
4.4.4 Macros Decl_Grid_Info(), Subdomain_Id() and Primarize_id() 229
4.4.5 Function Prototypes . . . 231
4.5 C Source File ohhelp2.c . . . 234
4.5.1 Header File Inclusion . . . 234
4.5.2 Function Prototypes . . . 234
4.5.3 oh2_init() and init2() . . . 235
4.5.4 oh2_transbound() and transbound2() . . . 238
4.5.5 try_primary2() . . . 239
4.5.6 exchange_primary_particles() . . . 240
4.5.7 try_stable2() . . . 243
4.5.8 rebalance2() . . . 244
4.5.9 move_to_sendbuf_primary() . . . 245
4.5.10 move_to_sendbuf_secondary() . . . 248
4.5.11 set_sendbuf_disps() . . . 252
4.5.12 exchange_particles() . . . 253
4.5.13 move_to_sendbuf_uw() . . . 257
4.5.14 move_to_sendbuf_dw() . . . 260
4.5.15 move_injected_to_sendbuf() . . . 261
4.5.16 move_injected_from_sendbuf() . . . 262
4.5.17 receive_particles() . . . 263
4.5.18 send_particles() . . . 264
4.5.19 oh2_inject_particle() . . . 265
4.5.20 oh2_remap_injected_particle() . . . 266
4.5.21 oh2_remove_injected_particle() . . . 267
4.5.22 oh2_set_total_particles() . . . 268
4.5.23 oh2_max_local_particles() . . . 268
4.6 Header File ohhelp3.h . . . 270
4.6.1 Control Variable . . . 270
4.6.2 Domain and Subdomain Descriptors . . . 270
4.6.3 Domain and Subdomain Boundaries . . . 273
4.6.4 Field Array Descriptors . . . 274
4.6.5 Boundary Communication Descriptors . . . 275
4.6.6 Function Prototypes . . . 277
4.7 C Source File ohhelp3.c . . . 282
4.7.1 Header File Inclusion . . . 282
4.7.2 Function Prototypes . . . 282
4.7.3 oh3_init() and oh13_init() . . . 283
4.7.4 init3() . . . 287

4.7.5 init_subdomain_actively() . . . 289
4.7.6 init_subdomain_passively() . . . 293
4.7.7 comp_xyz() . . . 296
4.7.8 Macro Field_Disp() . . . 297
4.7.9 init_fields() . . . 297
4.7.10 set_field_descriptors() . . . 301
4.7.11 set_border_exchange() . . . 302
4.7.12 set_border_comm() . . . 304
4.7.13 clear_border_exchange() . . . 309
4.7.14 oh3_grid_size() . . . 310
4.7.15 oh3_transbound() and transbound3() . . . 311
4.7.16 Macro Map_Particle_To_Neighbor() . . . 312
4.7.17 Macro Neighbor_Id() . . . 312
4.7.18 oh3_map_particle_to_neighbor() . . . 313
4.7.19 Macros Map_Particle_To_Subdomain() and Adjust_Subdomain() 314
4.7.20 oh3_map_particle_to_subdomain() . . . 315
4.7.21 map_irregular_subdomain() . . . 317
4.7.22 map_irregular() . . . 317
4.7.23 map_irregular_range() . . . 318
4.7.24 oh3_bcast_field() . . . 319
4.7.25 oh3_reduce_field() . . . 319
4.7.26 oh3_allreduce_field() . . . 320
4.7.27 oh3_exchange_borders() . . . 320
4.8 Level-4p Library Overview . . . 323
4.9 Header File ohhelp4p.h . . . 325
4.9.1 Constants . . . 325
4.9.2 Macros for Grid-Position . . . 325
4.9.3 Per-Grid Histograms and Related Variables . . . 328
4.9.4 Variables for Particle Transfer Scheduling . . . 335
4.9.5 Variables for Neighboring Information . . . 341
4.9.6 Variable for Boundary Condition . . . 342
4.9.7 Function Prototypes . . . 343
4.10 C Source File ohhelp4p.c . . . 346
4.10.1 Header File Inclusion . . . 346
4.10.2 Function Prototypes . . . 346
4.10.3 Macros If_Dim(), For_Z(), For_Y(), Do_Z(), Do_Y(), Coord_To_Index() and Index_To_Coord() . . . 351
4.10.4 Macros Decl_For_All_Grid(), For_All_Grid(), For_All_Grid_Abs(), The_Grid(), Grid_X(), Grid_Y() and Grid_Z() . . . 352
4.10.5 Constants URN_PRI, URN_SEC and URN_TRN . . . 354
4.10.6 oh4p_init() and init4p() . . . 354
4.10.7 oh4p_max_local_particles() . . . 361
4.10.8 oh4p_per_grid_histogram() . . . 361
4.10.9 oh4p_transbound() and transbound4p() . . . 362
4.10.10 try_primary4p() . . . 364
4.10.11 try_stable4p() . . . 366
4.10.12 rebalance4p() . . . 367
4.10.13 Macros Parent_Old(), Parent_New(), Parent_New_Same() and Parent_New_Diff() . . . 368

4.10.14 exchange_particles4p() . . . 369
4.10.15 exchange_population() . . . 371
4.10.16 add_population() . . . 373
4.10.17 mpi_allreduce_wrapper() . . . 374
4.10.18 reduce_population() . . . 374
4.10.19 make_recv_list() . . . 375
4.10.20 sched_recv() . . . 379
4.10.21 make_send_sched() . . . 383
4.10.22 make_send_sched_body() . . . 386
4.10.23 gather_hspot_recv() . . . 390
4.10.24 gather_hspot_send() . . . 392
4.10.25 gather_hspot_send_body() . . . 393
4.10.26 scatter_hspot_send() . . . 395
4.10.27 scatter_hspot_recv() . . . 398
4.10.28 scatter_hspot_recv_body() . . . 399
4.10.29 update_descriptors() . . . 401
4.10.30 update_neighbors() . . . 401
4.10.31 set_grid_descriptor() . . . 402
4.10.32 adjust_field_descriptor() . . . 403
4.10.33 update_real_neighbors() . . . 404
4.10.34 upd_real_nbr() . . . 408
4.10.35 exchange_xfer_amount() . . . 409
4.10.36 count_population() . . . 410
4.10.37 sort_particles() . . . 411
4.10.38 move_and_sort_primary() . . . 413
4.10.39 sort_received_particles() . . . 415
4.10.40 Macros Local_Grid_Position() and Move_Or_Do() . . . 416
4.10.41 move_to_sendbuf_sec4p() . . . 418
4.10.42 move_to_sendbuf_uw4p() . . . 420
4.10.43 move_to_sendbuf_dw4p() . . . 422
4.10.44 move_and_sort_secondary() . . . 422
4.10.45 set_sendbuf_disps4p() . . . 424
4.10.46 xfer_particles() . . . 425
4.10.47 Macro Check_Particle_Location() . . . 427
4.10.48 Macros Map_Particle_To_Neighbor() and Adjust_Neighbor_Grid() . . . 428
4.10.49 oh4p_map_particle_to_neighbor() . . . 429
4.10.50 Macros Map_To_Grid, Map_Particle_To_Subdomain() and Local_Coordinate() . . . 432
4.10.51 oh4p_map_particle_to_subdomain() . . . 434
4.10.52 oh4p_inject_particle() . . . 436
4.10.53 oh4p_remove_mapped_particle() . . . 437
4.10.54 oh4p_remap_particle_to_neighbor() . . . 438
4.10.55 oh4p_remap_particle_to_subdomain() . . . 439
4.11 Level-4s Library Overview . . . 440
4.12 Header File ohhelp4s.h . . . 442
4.12.1 Constants . . . 442
4.12.2 Macros for Grid-Position . . . 442
4.12.3 Per-Grid Histograms and Related Variables . . . 444
4.12.4 Variables for Particle Transfer Scheduling . . . 454

4.12.5 Variables for Neighboring Information . . . 458
4.12.6 Variable for Boundary Condition . . . 459
4.12.7 Function Prototypes . . . 459
4.13 C Source File ohhelp4s.c . . . 462
4.13.1 Header File Inclusion . . . 462
4.13.2 Function Prototypes . . . 462
4.13.3 Macros `If_Dim()`, `For_Y()`, `For_Z()`, `Do_Y()`, `Do_Z()` and `Coord_To_Index()` . . . 467
4.13.4 Macros `Decl_For_All_Grid()`, `For_All_Grid()`, `For_All_Grid_Abs()`, `The_Grid()`, `Grid_X()`, `Grid_Y()` and `Grid_Z()` . . . 468
4.13.5 Constants `URN_PRI`, `URN_SEC` and `URN_TRN` . . . 469
4.13.6 `oh4s_init()` and `init4s()` . . . 469
4.13.7 `oh4s_particle_buffer()` . . . 478
4.13.8 `oh4s_per_grid_histogram()` . . . 479
4.13.9 `oh4s_transbound()` and `transbound4s()` . . . 480
4.13.10 `try_primary4s()` . . . 482
4.13.11 `try_stable4s()` . . . 483
4.13.12 `rebalance4s()` . . . 483
4.13.13 Macros `Parent_Old()`, `Parent_New()`, `Parent_New_Same()` and `Parent_New_Diff()` . . . 484
4.13.14 `exchange_particles4s()` . . . 485
4.13.15 `count_population()` . . . 492
4.13.16 `exchange_population()` . . . 492
4.13.17 `reduce_population()` . . . 494
4.13.18 `add_population()` . . . 495
4.13.19 `make_recv_list()` . . . 496
4.13.20 `sched_recv()` . . . 499
4.13.21 `make_send_sched()` . . . 500
4.13.22 Macros `For_All_Grid_Z()`, `For_All_Grid_XY()`, `Grid_Exterior_Boundary()` and `Grid_Interior_Boundary()` . . . 503
4.13.23 `make_send_sched_body()` . . . 505
4.13.24 `make_send_sched_self()` . . . 507
4.13.25 `make_send_sched_hplane()` . . . 511
4.13.26 `update_descriptors()` . . . 512
4.13.27 `update_neighbors()` . . . 512
4.13.28 `set_grid_descriptor()` . . . 513
4.13.29 `adjust_field_descriptor()` . . . 514
4.13.30 `update_real_neighbors()` . . . 515
4.13.31 `upd_real_nbr()` . . . 516
4.13.32 `exchange_xfer_amount()` . . . 516
4.13.33 `make_bxfer_sched()` . . . 518
4.13.34 Macros `Add_Pillar_Voxel()`, `Is_Pillar_Voxel()`, `Pillar_Lower()` and `Pillar_Upper()` . . . 519
4.13.35 `make_bsend_sched()` . . . 520
4.13.36 `make_brecv_sched()` . . . 523
4.13.37 Macros `Local_Grid_Position()` and `Move_Or_Do()` . . . 524
4.13.38 `move_to_sendbuf_4s()` . . . 526
4.13.39 `move_to_sendbuf_uw4s()` . . . 529
4.13.40 `move_to_sendbuf_dw4s()` . . . 530

4.13.41 Macro Sort_Particle() . . . 531
4.13.42 sort_particles() . . . 532
4.13.43 move_and_sort() . . . 533
4.13.44 sort_received_particles() . . . 535
4.13.45 set_sendbuf_disps4s() . . . 535
4.13.46 xfer_particles() . . . 536
4.13.47 xfer_boundary_particles_v() . . . 537
4.13.48 xfer_boundary_particles_h() . . . 540
4.13.49 oh4s_exchange_border_data() . . . 542
4.13.50 exchange_border_data_v() . . . 542
4.13.51 exchange_border_data_h() . . . 545
4.13.52 Macro Check_Particle_Location() . . . 546
4.13.53 Macros Map_Particle_To_Neighbor() and Adjust_Neighbor_Grid() . . . 547
4.13.54 oh4s_map_particle_to_neighbor() . . . 548
4.13.55 Macros Map_To_Grid, Map_Particle_To_Subdomain() and Local_Coordinate() . . . 549
4.13.56 oh4s_map_particle_to_subdomain() . . . 550
4.13.57 oh4s_inject_particle() . . . 551
4.13.58 oh4s_remove_mapped_particle() . . . 552
4.13.59 oh4s_remap_particle_to_neighbor() . . . 552
4.13.60 oh4s_remap_particle_to_subdomain() . . . 553
4.14 Sample make Files . . . 554
4.14.1 samplef.mk for Fortran . . . 554
4.14.2 samplec.mk for C . . . 555

1 Introduction

Particle-in-Cell (PIC) simulations have played an indispensable role in theoretical and practical research of high-energy physics, space plasma physics, cloud modeling, combustion engineering, and so on, since early 1980's. In typical PIC simulations, a huge number of charged particles interact with electromagnetic field mapped onto a large number of grid points, governed by Maxwell's equations and the Lorentz force law. These hugeness and largeness of the simulation essentially require to parallelize the computation not only for efficient execution but also for feasible implementation on distributed memory systems which are the majority of modern supercomputers. That is, the simulation has to be decomposed almost equally so that good load balancing is achieved and, more importantly, each decomposed subproblem is accommodated by a local memory of limited capacity. This almost-equal decomposition is a necessary condition to make the simulation *scalable* so that we fully utilize larger scale systems with nearly stable efficiency by enlarging the problem size proportionally to the system size.

However, this necessary condition is satisfied neither by simple particle-decomposed simulations, by also simple static domain-decomposed ones, nor even by sophisticated dynamic domain-decomposed simulations, because a process in these conventional methods would have too large (sub)domain or too many particles. Therefore, we have proposed a new domain-decomposed PIC simulation method named *OhHelp*[1] which is scalable in terms of the number of particles as well as the domain size. Its problem decomposition and load balancing mechanisms are outlined as follows.

1. The space domain is equally partitioned to assign each subdomain to each node as its *primary* subdomain.

2. If one or more subdomains have too many particles, i.e., more than average plus a certain tolerance, every but one node is responsible for another subdomain which has particles more than average as its *secondary* subdomain.

3. A part of particles in the secondary subdomain of a node are assigned to the node so that no nodes have too many particles.

Since a node has to have at most two subdomains, OhHelp is scalable with respect to the domain size. As for the number of particles, OhHelp keeps its excess over the per-node average less than the tolerance by dynamically rearranging the secondary subdomain assignment and thus also achieves good scalability.

In the rest of this document, we describe OhHelp and its library as follows. In the next §2, OhHelp algorithm is explained more detailedly. Then §3, the heart of this document, describes API of the OhHelp library so that you incoroparte OhHelp into your own PIC simulator. Finally, the §4, being another important part of this document, gives the complete explanation of the OhHelp library implementation showing every line of its source code files.

References

[1] H. Nakashima, Y. Miyake, H. Usui and Y. Omura. OhHelp: A Scalable Domain-Decomposing Dynamic Load Balancing for Particle-in-Cell Simulations. In *Proc. Intl. Conf. Supercomputing*, pp. 90–99, June 2009.

2 OhHelp Algorithm

2.1 Overview and Definitions

As shown in Figure 1, OhHelp simply partitions the simulated D-dimensional space domain ($D \leq 3$) into (almost) equal-size N subdomains and assings each subdomain n ($n \in [0, N-1]$) to each of N (MPI) processeses, or computation *node*, whose MPI rank, or identifier, is also n, as its *primary subdomain*. In the figure, non-italic black numbers are the identifiers of nodes and also those of primary subdomains assigned to them. Each node n is responsible for its primary subdomain n, and also all the particles in it if the numbers of those *primary particles* in subdomains are balanced well, or more specifically, if the number of particles P_n in a subdomain n satisfies the following inequality for all n,

$$P_n \leq (P/N)(100 + \alpha)/100 \equiv P_{\max} \tag{1}$$

where P is the total number of particles and α is the tolerance factor percentage greater than 0 and less than 100. We refer to the simulation phases in this fortunate situation as those in *primary mode*.

Otherwise, i.e., if the inequality (1) is not satisfied for some subdomain n as shown in Figure 1, the simulation is performed in *secondary mode*. In this mode, every node, except for one node (12 in the figure), is responsible for a *secondary subdomain* having particles more than the average, in addition to its primary one. For example, the subdomain 22 has *helper* nodes 02, 30 and 33 shown in italic and blue letters in Figure 1. The particles in a densely populated subdomain are also distributed to its helper nodes as their *secondary particles* so that each node n has Q_n particles in total, which are the union of Q_n^n primary particles in the primary subdomain n and Q_n^m secondary particles in the secondary subdomain m, satisfying the following inequality for balancing similar to (1) for all n.

$$Q_n = Q_n^n + Q_n^m \leq (P/N)(100 + \alpha)/100 = P_{\max} \tag{2}$$

Note that since all but one nodes have secondary subdomains, a node whose primary subdomain is densely populated, e.g., node 22, is not only helped by other nodes but also helps another node 20, as the balancing algorithm discussed in §2.2 orders.

Also note that the load in secondary mode is balanced not only in the number of particles but also in the size of responsible subdomains, although the latter load is twice as heavy

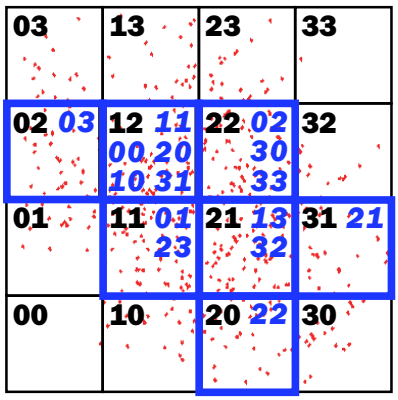


Figure 1: Space domain partitioning.

as that in the primary mode. This is another justification for making a node with densely populated primary subdomain help another node.

The examination whether the load is balanced well and the mode switching possibly with load rebalancing are performed as follows every simulation time step in which particles can move crossing subdomain boundaries[1].

1. If the inequality (1) is satisfied for all subdomains, the mode stays in or turns to primary. In the case of staying, only the particles crossing subdomain boundaries are transferred between nodes by neighboring communications. Otherwise, in addition to boundary crossing ones, particles that have been secondary are transferred to nodes responsible for them as primary particles.

2. If the current mode is secondary and the inequality (1) is not satisfied but (2) is satisfiable keeping the secondary subdomain assignment, the mode stays in secondary without global rebalancing. Particles may be transferred among the helpers and their *helpand*[2] for the local load balancing in addition to the transfer of the particles crossing boundaries. The statisfiability check for (2) and the local balancing are discussed in §2.3.

3. Otherwise, the secondary subdomain assignments are performed (or modified) so that Q_n is equal to P/N for all n to accomplish perfect balancing[3]. The subdomain assignment algorithm is discussed in §2.2.

2.2 Secondary Subdomain Assignment

When it is detected that the inequality (1) or (2) is unsatisfiable in primary or secondary mode respectively, secondary subdomains are assigned to nodes, by modifying the original assignment if the mode has already been in secondary, to accomplish perfect balancing. The assignment algorithm is quite simple as follows.

(b1) Split the set of nodes into two disjoint subsets $\mathcal{L} = \{n \,|\, P_n < P/N\}$ and $\mathcal{G} = \{n \,|\, P_n \geq P/N\}$. Let the tentative value of Q_n be P_n for all n.

(b2) Repeat the following steps (b3) through (b5) until $\mathcal{L}$ becomes empty.

(b3) Remove an element l from $\mathcal{L}$ such that $Q_l = \min_{n\in\mathcal{L}}\{Q_n\}$ and remove an element g from $\mathcal{G}$ as follows.

- If the mode is secondary and l has been helping a node n in $\mathcal{G}$, let g be n.
- Otherwise, the node g is chosen such that $Q_g = \max_{n\in\mathcal{G}}\{Q_n\}$.

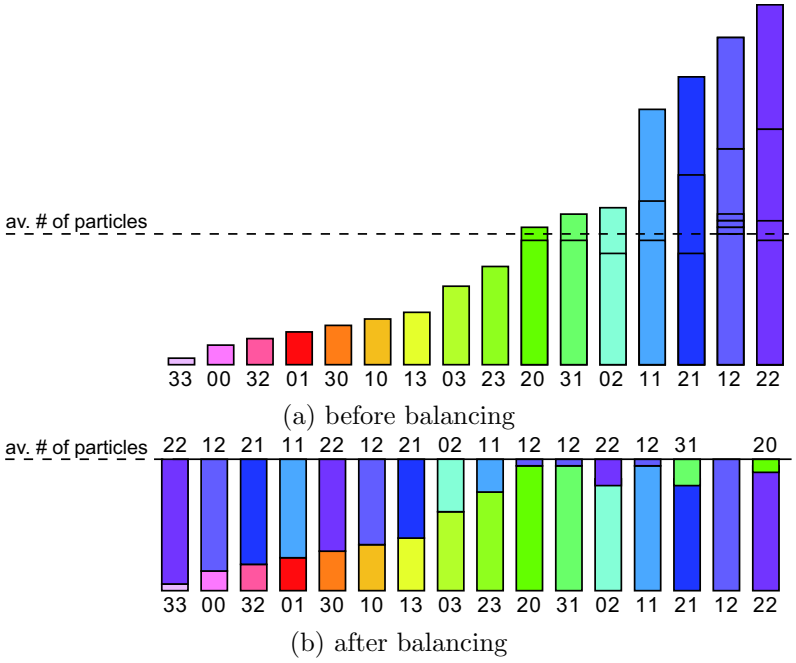
(b4) Assign the subdomain g to the node l as its secondary subdomain and also assign $Q_l^g = (P/N) - Q_l$ particles in the subdomain g to the node l so that $Q_l \leftarrow Q_l + Q_l^g = P/N$. Now Q_g becomes $Q_g - Q_l^g$.

(b5) If $Q_g < P/N$, add g to $\mathcal{L}$. Otherwise add g back to $\mathcal{G}$.

[1] You may reduce the frequency of these operations by overlapping adjacent subdomains a little bit more heavily and by exploiting the fact that the velocity of a particle is limited to some upper bound, e.g., light speed.

[2] We know English does not has such a word but dare to neologize to mean "the node helped by other nodes."

[3] If P is a multiple of N. Otherwise, Q_n is $\lfloor P/N \rfloor$ or $\lceil P/N \rceil$, but we assume P is a multiple of N in this section for the sake of explanation simplicity.



(a) before balancing

(b) after balancing

Figure 2: Subdomain assignment with perfect balancing of number of particles.

(b6) If $\mathcal{G}$ has two or more elements, pick an arbitrary element r from $\mathcal{G}$ and assign the subdomain r to other nodes in $\mathcal{G}$ without particle assignment. Otherwise, i.e., $\mathcal{G}$ has only one element, let r be this node.

It is obvious the algorithm stops making every node n except for r have a secondary subdomain and $Q_n = P/N$ for all n. As mentioned in §2.1, the key for perfect balancing is the step (b5) where we add g with $P_g \geq P/N$ but $Q_g < P/N$ to $\mathcal{L}$ so that it helps other node when it has deputed so many particles to its helpers that Q_g becomes less than P/N tentatively. Figure 2 shows an example balancing result for the particle distribution shown in Figure 1 providing we suddenly faces the imbalance due to, for example, initial particle positioning. The number of particles in each subdomain (a) and that assigned to each node (b) are illustrated by the bar whose color and numbers above and below it represent the subdomain and the node.

2.3 Checking and Keeping Local Balancing

In the secondary mode, the particle movements crossing subdomain boundaries could break the satisfiability of the inequality (2) if we stuck to the secondary subdomain assignment. To examine the satisfiability and to keep the local balancing among a helpand-helper *family*, we form a tree T whose vertices are the computation nodes and edges represent helpand-helper relationship. That is, the root of the tree is the node r defined in the step (b6) of the previous section, and the parent of a non-root node is its helpand. The tree corresponding to the balancing result in Figure 2(b) is show in Figure 3.

The examination of the satisfiability of (2) is performed by traversing the tree T in a bottom-up (leaf-to-root) manner as follows.

(e1) Let a set of nodes $\mathcal{S}$ be that of leaves of the tree T. Let $P_n^{\min}$ be P_n for all $n \in \mathcal{S}$. If

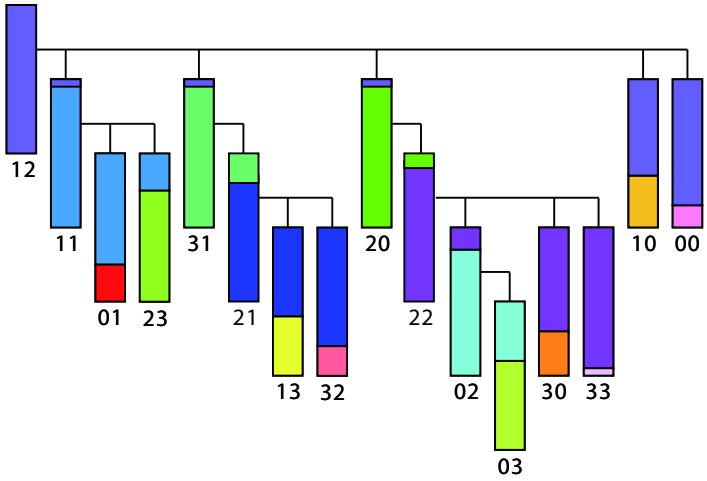


Figure 3: Helpand-helper tree for balancing result in Figure 2(b).

there is an element $n \in \mathcal{S}$ such that $P_n = P_n^{\min} > P_{\max}$, the examination fails.

(e2) Repeat the following steps (e3) and (e4) until $\mathcal{S}$ becomes $\{r\}$.

(e3) Find a node n such that the set of its helpers $H(n)$ is a subset of $\mathcal{S}$, and remove $H(n)$ from $\mathcal{S}$.

(e4) Add n to $\mathcal{S}$ and let $P_n^{\min}$ be as follows.

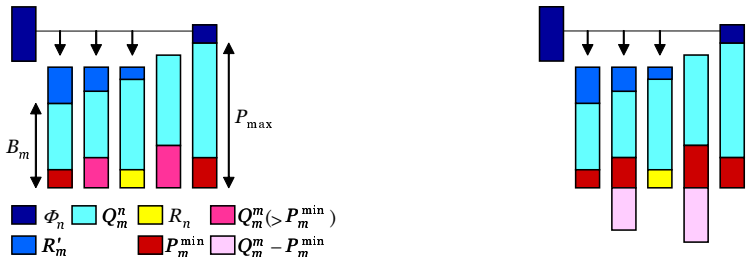
$$P_n^{\min} = \max(0, P_n - \sum_{m \in H(n)} (P_{\max} - P_m^{\min}))$$

If $P_n^{\min} > P_{\max}$, the examination fails.

Since a leaf node does not have helpers, the failure in the step (e1) obviously means that the inequality (2) cannot be satisfied. As for the failure in (e4), since $\sum_{m \in H(n)} (P_{\max} - P_m^{\min})$ means the maximum particle amount which n's helpers accommodate as their secondary particles and thus $P_n^{\min}$ is the minimum number of particles in n which the node n has to be responsible for, $P_n^{\min} > P_{\max}$ leads us that the inequality (2) is unsatisfiable. Therefore, the algorithm is complete. On the other hand, when the algorithm stops at (e2) with $P_n^{\min} \leq P_{\max}$ for all n, it is assured that, for all n, P_n particles can be distributed among n and its helpers keeping $Q_m \leq P_{\max}$ for all $m \in F(n)$ where $F(n)$ is defined as $\{n\} \cup H(n)$. That is, even if n has to accommodate $P_{\max} - P_n^{\min}$ particles for its helpand, $P_n - P_n^{\min}$ particles can be accommodated by its helpers because they are at most $\sum_{m \in H(n)} (P_{\max} - P_m^{\min})$. Therefore, the algorithm is sound.

If the examination passes, a part of particles in a subdomain n are redistributed to the members of the family $F(n)$, i.e., the node n and its helpers in $H(n)$. The target of the redistribution is the following, where Q_k^n is the number of particles in the subdomain n and currently accommodated by the node k.

- Particles currently in a node $m \notin F(n)$, which has just crossed a boundary and moved into the subdomain n from other subdomain.



(a) without pushing down (b) with pushing down

Figure 4: Particle redistribution in a family.

- Particles overflown from a node $m \in F(n)$. More specifically, particles are overflown from m in either of the following cases.
  - $m \neq n$ and $Q_m^n + P_m^{\min} > P_{\max}$ and thus $Q_m^n + P_m^{\min} - P_{\max}$ particles are overflown to satisfy the minimum requirement defined by $P_m^{\min}$.
  - $m = n$ and $Q_n^n + R_n > P_{\max}$ where R_n is the number of particles assigned to n as the result of the redistribution for the family rooted by $p = \mathit{parent}(n)$ to which n belongs as a helper. That is, $R_n = Q_n^p$ at the beginning of the next simulation step. The number of overflown particles is $Q_n^n + R_n - P_{\max}$.

Note that the criteria above are to minimize the amount of particle transfer rather than to minimize the load deviation among the nodes. Let R_n^{flt} be the total number of redistributed particles defined above or, more specifically, be as follows.

$$R_n^{\mathrm{flt}} = \sum_{k \notin F(n)} Q_k^n + \sum_{m \in H(n)} \max(0,\, Q_m^n + P_m^{\min} - P_{\max}) + \max(0,\, Q_n^n + R_n - P_{\max})$$

The local balancing in a helpand-helper family is partly achieved by the following algorithm traversing the tree T in a top-down manner.

(d1) Let a set of node $\mathcal{S} = \{r\}$, and $R_r = 0$.

(d2) Repeat the following steps (d3) to (d6) until $\mathcal{S}$ becomes empty.

(d3) Remove a node n from $\mathcal{S}$. If n is the leaf node, let Q_n be $P_n + R_n$ and skip the following steps (d4) to (d6). Otherwise, add the helpers of n, i.e., $H(n)$, to $\mathcal{S}$.

(d4) If the following inequality is satisfied;

$$P_n + R_n + \sum_{m \in H(n)} \max(P_m^{\min}, Q_m^m) \leq P_{\max} \cdot |F(n)|$$

we need not to *push down* primary particles of any node m to its own helpers. If this holds, let $B_m = \min(P_{\max},\, Q_m^n + \max(P_m^{\min}, Q_m^m))$ for all $m \in H(n)$ to represent the *baseline* number of particles above which we place particles to be redistributed as shown in Figure 4(a). Otherwise, let the baseline B_m be $\min(P_{\max},\, Q_m^n + P_m^{\min})$ to allow us to push down $Q_m^m - P_m^{\min}$ particles as shown in Figure 4(b). In both cases, let B_n, the baseline of n, be $\min(P_{\max},\, Q_n^n + R_n)$.

(d5) Find the minimum subset $F_l(n)$ of $F(n)$ such that the followings are satisfied.

$$\forall m' \in F_l(n), \forall m \in F(n) - F_l(n) : B_{m'} \leq B_m$$
$$\forall m \in F(n) - F_l(n) : R_n^{\text{flt}} + \sum_{m' \in F_l(n)} B_{m'} \leq B_m \cdot |F_l(n)|$$

(d6) Let R_m for all $m \in H(n)$ and Q_n be the followings.

$$R'_m = \begin{cases} (R_n^{\text{flt}} + \sum_{m' \in F_l(n)} B_{m'})/|F_l(n)| - B_m & m \in F_l(n) \\ 0 & m \notin F_l(n) \end{cases}$$
$$R_m = R'_m \qquad Q_n = B_n + R'_n$$

The step (d5) is to find the leftmost three bars (nodes) in Figure 4(a) and (b) for the local load balancing among these lightly loaded nodes by distributing R'_m given in the step (d6).

3 OhHelp Library

3.1 Library Layers

The OhHelp library package has three fundamental layers which are referred to as level-1, level-2 and level-3, and (so far) two extensional layers level-4p and level-4s. The functions provieded by each layer are summarized as follows.

level-1: This level provieds a load-balancer function named `oh1_transbound()` which examines whether particles are distriuted among nodes in a well-balanced manner, (re)builds helpand-helper configuration if necessary, and tells you how to move particles among nodes. That is, this function implements the OhHelp algorithm described in §2. In addition, level-1 library has functions for collective communications in helpand-helper families, and those for statistics and verbose messaging. See §3.4 for functions excluding those for statistics and verbose messaging which are explained in §3.10 and §3.11 respectively.

level-2: In this level, the load-balancer function `oh2_transbound()` does what its level-1 counterpart does, and transfers particles among nodes according to the schedule determined by the level-1 function. See §3.5 for detailed explanation of level-2 API functions.

level-3: Functions for particle manipulation added in this level are to determine the identifier of the subdomain where a given particle resides. The other useful functions are for inter-node communications of arrays having vectors/scalars associated with grid points in a subdomain, i.e., those for electromagnetic field, current density, and so on. See §3.6 for detailed explanation of level-3 API functions.

level-4p: This extensional level is for *position-aware particle management* with which the load balancing mechanism takes care of particle positions so that all particles in a *grid-voxel* are accommodated by a particular node (almost) always. Moreover, primary/secondary particles of a specific species in a node are *sorted* according to the coordinates of the grid-voxels in which they reside so that you easily find a set of particles in a particular grid-voxel for, e.g., Monte Carlo collision. See §3.7 for detailed explanation of level-4p extension and its functinos.

level-4s: This extensional level is to provide yet another position-aware mechanism for, e.g., SPH (Smoothed Particle Hydrodynamics) method. The differences between this extension and level-4p one are as follows; each node is responsible of all particles in a cuboid split from the subdomain for the node by slicing it by planes perpendicular to z-axis; and each node accommodates not only the particles in the cuboid but also those in the grid-voxels surrouding the cuboid as *halo* particles so that the compuation on a particle in the cuboid may refer to particles nearby the particle. See §3.8 for detailed explanation of level-4s extension and its functinos.

Functions in each fundamental layer are composed in a level-specific source file, namely ohhelp1.c, ohhelp2.c and ohhelp3.c which require header files of same names, i.e., ohhelp1.h, ohhelp2.h and ohhelp3.h. To have a library of level-2 or level-3, it is required to compile lower level libraries as well, and thus you will have all functions in all layers if you are to use level-3 library. However, this does not mean that you have to use all functionalities provided by all level libraries. In fact, except for the essential functionality given by `oh1_transbound()`, you are almost free to pick functions you like to use. Therefore, API functions are named with prefixes '`oh1_`', '`oh2_`' or '`oh3_`' to show which level they belong to.

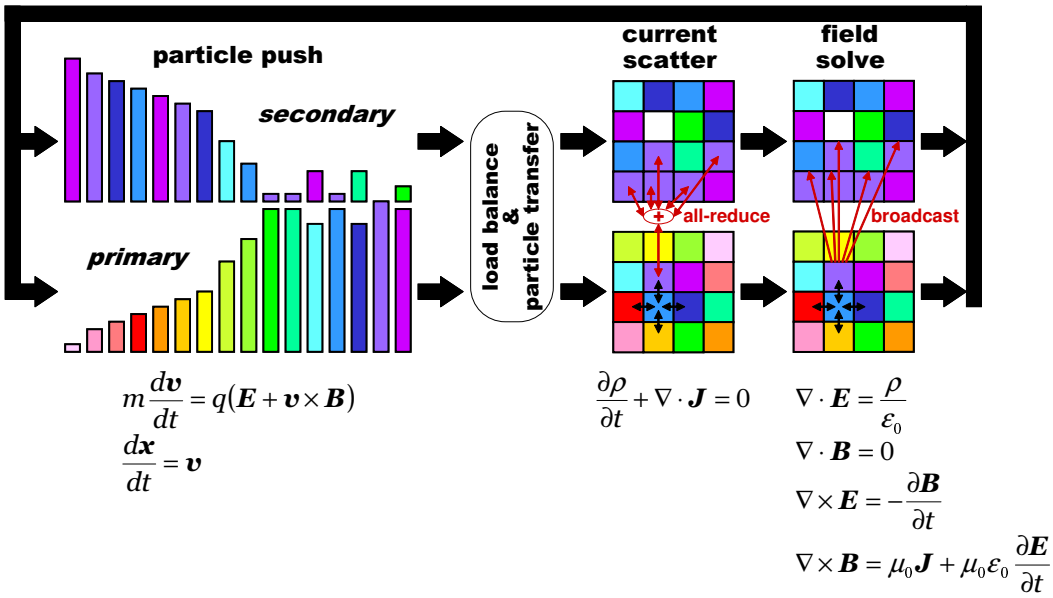


Figure 5: Typical 3D PIC simulator with OhHelp.

On the other hand, the level-4p and level-4s extensions implemented by ohhelp4p.c, ohhelp4p.h, ohhelp4s.c, ohhelp4s.h are only for users who need position-aware particle management given by API functions having prefix '`oh4p_`' or '`oh4s_`' respectively. Therefore, if your simulation is not position-aware, it is safe to exclude these files for the extension from your `make` file. Otherwise, you are required not only to compile and link them but also to activate the level-4p/4s extension by editing the header file oh_config.h as discussed in §3.3. Note that level-4p and level-4s extensions are mutually exclusive.

This naming rule shown above could be too rigid for you to use all functions provided by your preferred layer and lower, because it will be tiresome to remember the layer number which a function belongs to. Therefore, the library has special header files ohhelp_f.h for Fortran programmers and ohhelp_c.h for those who love C, in order to give API functions aliases which just have a common prefix '`oh_`' as discussed in §3.12.

3.2 Applying OhHelp to PIC Simulators

Figure 5 shows a typical configuration of OhHelp'ed PIC simulators. In the figure, it is assumed that the baseline simulator to apply OhHelp is domain-decomposed and its main loop consists of four phases, *particle pushing*, *particle transferring*, *current scattering*, and *field solving* as follows.

particle pushing: Each node accelerates particles residing in the subdomain assigned to the node by electric and the Lorentz force law referring to electromagnetic field data $\boldsymbol{E}$ and $\boldsymbol{B}$ associated to the grid points in its subdomain. Then the node moves particles according to their updated velocities. Particle movements crossing subdomain boundaries will be taken care of by the next phase.

particle transferring: Each node transfers paritcles, which has crossed its subdomain boundaries, to the nodes responsible for adjacent subdomains.

current scattering: Each node calculates the contributions of the movement of its particles to the current density $\boldsymbol{J}$ at the grid points in its subdomain. Then the boundary values of $\boldsymbol{J}$ are exchanged between adjacent subdomains.

field solving: Each node locally updates the values of $\boldsymbol{E}$ and $\boldsymbol{B}$ at the grid points in its subdomain using, for example, leapfrog method to solve Maxwell's equations. Then the boundary values of $\boldsymbol{E}$ and $\boldsymbol{B}$ are exchanged between adjacent subdomains.

Applying OhHelp to the baseline simulator outlined above is fairly easy. In fact, required modifications to the main simulation loop of the baseline simulator are just as follows.

duplication of data structures: Data structures for the subdomain and particles in it should be duplicated so that a node has primary and secondary subdomains and particles.

duplication of computation: The phases except for particle transferring of the main loop should be duplicated to locally update particle and field data.

addition of collective communications: Current densities for a secondary subdomain is calculated locally and thus should be summed up to have the complete data for the subdomain. The boundary or whole values of electromagnetic field should be broadcasted from each helpand to its helpers.

attachment of load balancer: To transfer particles among nodes, the library function for load balancing should be called to have the transfer schedule or to do the transfer itself.

In the following subsections, the modifications above are explained more detailedly.

3.2.1 Duplication of Data Structures

Since each node may have primary and secondary subdomains and particles, you have to duplicate data structure for electromagnetic field and current density to have those for primary subdomain and for secondary subdomain. For example, suppose the baseline simulator is coded in Fortran and the electromagnetic field for a subdomain is declared and allocated as;

```
real*8,allocatable :: eb(:,:,:,:)
allocate(eb(6, φ_x^l:φ_x^u−1, φ_y^l:φ_y^u−1, φ_z^l:φ_z^u−1))
```

where the first dimension is for three components of electric field vector and those of magnetic field vector, and ϕ_x^l and ϕ_x^u and their counterparts of y and z axes are lower and upper boundaries of the subdomain including a few planes for the overlap of adjacent subdomains. An OhHelp'ed version of this four-dimensional array has one additional dimension for primary and secondary ones and is declared and allocated as;

```
real*8,allocatable :: eb(:,:,:,:,:)
allocate(eb(6, φ_x^l:φ_x^u−1, φ_y^l:φ_y^u−1, φ_z^l:φ_z^u−1, 2))
```

to have primary field data in the subarray `eb(:,:,:,:,1)` while secondary data are stored in the other subarray `eb(:,:,:,:,2)`.

The other example for a C-coded simulator is given with the following declaration and allocation.

```
struct ebfield {double ex,ey,ez,bx,by,bz} *eb;
eb = (struct ebfield*)
     malloc(sizeof(struct ebfield)*(φ^u_x−φ^l_x)(φ^u_y−φ^l_y)(φ^u_z−φ^l_z));
```

A reasonable way to apply OhHelp to the example above is;

```
struct ebfield {double ex,ey,ez,bx,by,bz} *eb[2];
eb[0] = (struct ebfield*)
        malloc(sizeof(srtuct ebfield)*(φ^u_x−φ^l_x)(φ^u_y−φ^l_y)(φ^u_z−φ^l_z)*2);
eb[1] = eb[0] + (φ^u_x−φ^l_x)(φ^u_y−φ^l_y)(φ^u_z−φ^l_z);
```

Note that, for both examples above, ϕ_x^u, ϕ_y^u and ϕ_z^u in OhHelp'ed version could have to be larger than those in the original version because they must be for the largest subdomain in the system rather than for the primary subdomain for the node if the subdomain size is not uniform in the system. That is, the node should be able to be responsible for any subdomain in the system. Also note that it is not necessary to represent the electromagnetic field by one array, but you may have two arrays for electric and magnetic fields, or even six arrays for each component of electric and magnetic field vectors. However, you have to remember that splitting arrays should cost in the communication of them for boudnary data exchange and broadcast and/or reduction in helpand-helper families.

On the other hand, adding a dimension to the array for particles to accommodate primary and secondary ones is not a good idea, because the number of particles in each category is not fixed. Therefore, the array must have $P_{\max}$ elements[4] defined in the inequality (1) in §2.1. Then, in the node n, the first part of the array should accommodate Q_n^n primary particles while the second part, which directly follows the first part, should have Q_n^p particles for $p = \mathit{parent}(n)$. The values of Q_n^n and Q_n^p are given by the library function for load balancing as discussed later.

The other remark on the array of particles is that if the array is partitioned into portions for S *species*, the library should know it. For example, suppose the baseline simulator has two particle species, one for (super)ions and the other for (super)electrons, and the particle array is partitioned into two regions to store ions in the first region and electrons in the second region. This partitioning is done, for example, to save memory space eliminating species identifier and/or physical quantities of species such as the charge and mass of a particle from the array element representing a particle, and/or to save operations for the references to these quantities and for the calculations on them. Since the layout of two types of particles should be kept after the particle transfer, the library function for load balancing have to aware that $S = 2$ to make transfer schedule and, if you desire to do, to transfer particles. The function is also capable to report you the number of particles for each species and each of primary/secondary categories.

Note that particle transferring for a simulation step should consist of S transfers for each species, a large S, say 10 or more, may cause a too large communication overhead to benefit from the array partitioning. Therefore, if your simulation has a large number of species, it is recommended to attach the species identifier and/or the physical quantities to each particle and tell the library that $S = 1$.

Also note that if you apply level-2 library or above[5], a particle should be represented by a structured data which should include particle position coordinates, velocity vector components, and other necessary information as discussed in §3.5.1. Otherwise, i.e., if you

[4] If the total number of particles in the system fluctuates due to, for example, particle injection and/or removal, P for $P_{\max}$ calculation in the inequality (1) should be the maximum number of total particles in the simulation.

[5] Unless you choose partial application of level-3 disabling level-2 functions, which is discussed in §3.6.2.

use level-1 only and transfer particles among nodes by yourself, the set of particles accommodated in a node can be represeted in two or more arrays paying some communication overhead.

3.2.2 Duplication of Computation

Since a particle is accommodated by only one node, the node is of course fully responsible for the particle. Therefore, each node should perform particle pushing and current scattering for its primary and secondary particles. A reasonable way to implement this duplicated computation for particles is to call functions corresponding to the operations twice.

For example, if your simulator has a Fortran subroutine named `particle_push()` with three arguments for the particle array, its size and electromagnetic field, fundamental operation to duplicate particle pushing is easy as follows, providing the array `pbuf` has particles each of which is represented by a structured data.

```
call particle_push(pbuf(1), Q_n^n, eb(:,:,:,:,1))
call particle_push(pbuf(Q_n^n+1), Q_n^p, eb(:,:,:,:,2))
```

However, this is not sufficient because two instances of `particle_push()` should have different *base coordinates* by which the particle position in the coordinate system of whole domain is mapped onto local coordinate system for a subdomain. That is, suppose the base simulator calculates the particle velocity in `particle_push()` by;

```
call lorentz(eb, pbuf(i)%x-xl, pbuf(i)%y-yl, pbuf(i)%z-zl, acc(1:3))
pbuf(i)%vx = pbuf(i)%vx + acc(1)
pbuf(i)%vy = pbuf(i)%vy + acc(2)
pbuf(i)%vz = pbuf(i)%vz + acc(3)
```

where the structure elements `x`, `y`, `z`, `vx`, `vy` and `vz` are for $x/y/z$-components of the position and the velocity of the `i`-th particle, `lorentz()` is the subroutine to calculate accelaration vector `acc(1:3)` referring to electromagnetic field vectors on the grid points surrounding the particle, and `xl`, `yl` and `zl` are the base coordinates of the subdomain, i.e., the coordinates of the west-south-bottom corner of the subdomain.

The code above should be modified to refer to subdomain dependent base coordinates. A reasonable way is to have a map of subdomain boundaries, say `sdoms(2,3,`N`)` whose element `sdoms(`β`,`d`,`n`)` has lower ($\beta = 1$) or upper ($\beta = 2$) boundary of d-th dimension of the subdomain n. With this array, the modified version of `particle_push()` has an additional array argument, say `sdom(2,3)` for the subdomain in problem and is called as follows where $p = parent(n)$.

```
call particle_push(pbuf(1), Q_n^n, eb(:,:,:,:,1), sdoms(:,:,n))
call particle_push(pbuf(Q_n^n+1), Q_n^p, eb(:,:,:,:,2), sdoms(:,:,p))
```

Then at the beginning of the body of `particle_push()`, the following assignment is added for the base coordinates where `sdom` is the fourth argument array passed to the subroutine.

```
xl = sdom(1,1)
yl = sdom(1,2)
zl = sdom(1,3)
```

Note that the upper boundaries `sdom(2,:)` will also be used in the function to detect the particles crossing the subdomain boundaries. Remember that you are responsible for counting number of particles in each subdomain, each species and each primary/secondary category and for reporting it to the library. Also note that the array equivalent to `sdoms(:,:,:)`

can be given by the initialization function `oh3_init()` of level-3 library as discussed in §3.6.1. A C-code version of the example above looks as follows.

```
particle_push(pbuf, Q_n^n, eb[0], sdoms[n]);
particle_push(pbuf+Q_n^n, Q_n^p, eb[1], sdoms[p]);
...
void particle_push(struct S_particle *pbuf, int nofparticles,
                   struct ebfield *eb, int sdom[2][3]) {
  int xl=sdom[0][0], yl=sdom[0][1], zl=sdom[0][2];  double acc[3];
  ...
  lorentz(eb, pbuf[i].x-xl, pbuf[i].y-yl, pbuf[i].z-zl, acc);

  pbuf[i].vx += acc[0];
  pbuf[i].vy += acc[1];
  pbuf[i].vz += acc[2];
  ...
}
```

The modification of current scattering can be implemented similarly, but it needs collective communications to sum local results of the scattering calculated by nodes in the family. The sum is obtained by a simple reduce operation or by an all-reduce operation to share the sum in family members, depending on the implementation of field solving as discussed below. Also, the (all-)reduce communication is discussed in §3.2.3.

As for field solving, there are two reasonable ways to modify its baseline implementation. The first candidate is to simply broadcast the solution of primary subdomain from each helpand to its helpers. That is, each node updates electromagnetic field vectors in its primary subdomain, exchanges boundary data between adjacent nodes, and broadcasts the whole field vectors in its primary subdomain and a few boundary planes to its helpers by the method discussed in §3.2.3. In this broadcast-type implementation, since the current density on each grid point in a subdomain is referred to only by the node responsible for the subdomain as primary, summing current densities will be performed by a simple one-way reduction followed by boundary exchange.

The other candidate is to duplicate the calculation of field solving. That is, each node updates electromagnetic field vectors in both primary and secondary subdomains. A reasonable way to obtain boudary values is to exchange boundary planes of adjacent primary subdomains and then to broadcast planes to the helpers. In this duplicate-type implementation, since the current density on each grid point in a subdomain is referred to by all the nodes responsible for the subdomain as primary or secondary, summing current densities will be performed by an all-reduce communication followed by boundary exchange between primary subdomains and boradcast boundary planes from the helpand to its helpers.

The choice from these two candidates should be determined by trading off the computation cost of field solving and the communication cost of broadcasting. In practice, if your simulator performs one leapfrog solving per one simulation step, the duplicate-type should be chosen because a leapfrog update of a subdomain is faster than broadcast. On the other hand, if your simulator adopts sub-stepping method to iterate leapfrog multiple times in a simulation step with, for example, particle-fluid hybrid method, the broadcast-type can be better.

3.2.3 Addition of Collective Communications

As discussed in §3.2.2, you need to add at least the following collective communications.

- A simple one-way reduction or an all-reduce communication to sum the current density among family members. In the latter case, the current density vectors of grid points in boundary planes should be broadcasted from the helpand to its helpers.
- Broadcast of electromagnetic field vectors of the whole of or the boundaries of the subdomain from the helpand to its helpers.
- Broadcast of electromagnetic field vectors when the helpand-helper tree is reconfigured due to an unacceptable imbalance and each node has new helpand.

Fundamentally, the collective operations above are performed by MPI functions, `MPI_Reduce()` or `MPI_Allreduce()` and `MPI_Bcast()` with argument `comm` being the communicator for the family which each node belongs to. Simply calling these functions, however, should cause a severe performance problem because a node may belong to *two* families, one as the helpand and the other as a helper. That is, if we carelessly perform collective communications by doing them, for example, as the helpand and then as a helper, it may cause unnecessary serialization because the root family must wait the completion of the communications in the second generation families which must wait those in the third ones and so on. Reversing the helpand/helper order cannot solve the problem because the bottom families must wait the completion in the second-bottom ones and so on.

This problem is solved by a simple red-black technique which paints families of odd-number generations by red and even ones by black and performs communications of red families first and then of black families. Since families of same color are mutually exclusive, the communications among them are performed in parallel.

The library provides various means for the red-black collective communications as follows.

- Level-1 library manages the family communicators and report you the communicators for the families which the local node belongs to, together with their colors and the ranks of the roots in the communicators. These information is sufficient to implement your own version of collective communications besides those provided by the library shown below.
- Level-1 library also provides you with functions for one-way reduction, all-reduce and broadcast with given data buffers, data counts and data types. All of these functions take care of the red-black ordering and special treatment for the tree root and leaves, each of which belongs to only one family.
- Level-3 library provides functions for you to perform one-way reduction, all-reduce and broadcast of the current vectors, electromagnetic field, and other arrays, for example that having charge densities, if necessary. The usage of the functions is much simpler than the level-1 counterparts, because you simply need to *register* each of *field arrays*, which are arrays for current density vectors, electromagnetic field and so on having elements associated to the grid points in a subdomain, and call these functions with primary and secondary arrays and the identifier of the array.
- Level-3 library also provides a function to exchange boundaries of field-arrays optionally followed by broadcast of boundary data from the helpand to its helpers.

3.2.4 Attachment of Load Balancer

Attaching OhHelp load balancer to your simulator is of course essential. What you need to do is simply calling `oh`l`_transbound()`, where l is level identifier in $\{1, 2, 3, 4\mathrm{p}, 4\mathrm{s}\}$ with

a few explicit arguments. In addition, if you use a fundamental level library (i.e., 1 to 3), you have to (implicitly) give it a histogram of particles accommodated by the local node. That is, if your code is written in Fortran, you have to have an array, say `nphgram(`N`,`S`,2)` whose element `nphgram(`m`+1,`s`,`c`)` has the number of particles residing in the subdomain $m \in [0, N)$, categorized in the species $s \in [1, S]$ and accommodated by the local node as its primary ($c = 1$) or secondary ($c = 2$) ones.

Similarly, C-coded simulator should have a conceptually three-dimensional array `nphgram[`$N \times S \times 2$`]` whose element `nphgram[`$m + N(s + Sc)$`]` has the number of particles residing in the subdomain m, categorized in the species $s \in [0, S - 1]$ and accommodated by the local node as its primary ($c = 0$) or secondary ($c = 1$) ones. If you like to access the array elelment by `nphgram[`c`][`s`][`m`]` in your ANSI-C code, you have to do the followings.

```
int **nphgram[2];
nphgram[0] = (int**)malloc(sizeof(int*)*S*2);
nphgram[1] = nphgram[0] + S;
nphgram[0][0] = (int*)malloc(sizeof(int)*N*S*2);
nphgram[1][0] = nphgram[0][0] + N*S;
for (i=0; i<2; i++)  for (j=1; j<S; j++)
  nphgram[i][j] = nphgram[i][j-1] + N;
```

Alternatively, you may choose C99 to simplify the code snip above to have the following.

```
int (*nphgram)[S][N]=(int(*)[S][N])malloc(sizeof(int)*N*S*2);
```

The main output you will obtain from the level-1 `oh1_transbound()` is a pair of (conceptually) three-dimensional arrays, say `rcounts(`N`,`S`,2)` and `scounts(`N`,`S`,2)` for Fortran or `rcounts[`$N \times S \times 2$`]` and `scounts[`$N \times S \times 2$`]` for C, which tell you incoming and outgoing particle transfer schedules. That is, `rcounts(`m`+1,`s`,`c`)` and `scounts(`m`+1,`s`,`c`)` notify you how many particles of species s should be received/sent from/to the node m as receiver's primary ($c = 1$) or secondary ($c = 2$) ones. Similary, `rcounts[`$m + N(s + Sc)$`]` and `scounts[`$m + N(s + Sc)$`]` tell you the receiving/sending counts of primary ($c = 0$) or secondary($c = 1$) particles for the node m and species s.

On the other hand, level-2 function `oh2_transbound()` and its level-3 counterpart `oh3_transbound()` perform particle transfer on behalf of you. To make the functions do the job easily, you have to give an additional tip to show which subdomain each particle has moved into. That is, each element of the array of particles, say `pbuf`, should be a structured data having an element `nid` to have the identifier of the subdomain where the particle is residing after particle pushing. Therefore, your subroutine/function for particle pushing should modify this element for each particle which has just crossed the subdomain boundary. Remember that level-3 library has functions to calculate the subdomain identifier from the particle position.

An important notice is that the transfer schedule given by `oh1_transbound()` and that used in `oh2_transbound()` and `oh3_transbound()` are *unaware* of particle positions. That is, in secondary mode, a pair of closely located particles may be parted from each other to be accommodated by two different nodes in the family for the subdomain where the pair resides. Therefore, if your simulator takes care of proximal particle-particle interactions using, for example, Monte Carlo Collision method, you have to use position-aware level-4p or level-4s library and their function `oh4p_transbound()` or `oh4s_transbound()`. Unlike its lower level counterparts, they do not need the per-subdomain particle histogram `nphgram` because the histogram is maintained inside of the library. For other important functionality of the level-4p/4s libraries such as *per-grid histogram* and *sorted layout* of particle buffer, see §3.7 and/or §3.8.

3.3 Configuration: Dimension of Simulated Space and Library Level

The OhHelp library can be applied to PIC simulations of one-dimensional, two-dimensional or three-dimensional space domain. For the sake of efficiency, however, the number of dimensions D is hard-coded in the library source code using a C constant macro named `OH_DIMENSION` whose default value is three. Therefore, if your simulator is one- or two-dimensional, you have to explicitly define the macro through the compiler option `-DOH_DIMENSION=1` or `-DOH_DIMENSION=2`, or have to edit the header file oh_config.h in which the default definition of `OH_DIMENSION` is given as follows.

```
#ifndef OH_DIMENSION
#define OH_DIMENSION    3
#endif
```

Remember that oh_config.h is included by ohhelp_f.h and ohhelp_c.h for function aliasing and thus modifying oh_config.h is easier to have consistent definition if you use aliases. Also remember that oh_config.h has the following lines which you may modify (remove comment) to `#define` a macro named `OH_LIB_LEVEL_4P` or `OH_LIB_LEVEL_4S` for the activation of level-4p or level-4s extension, which we will discuss in §3.7 and §3.8 respectively. Note that the lines following the commented-out definitions `#define`s another macro `OH_LIB_LEVEL_4PS` if you `#define` either `OH_LIB_LEVEL_4P` or `OH_LIB_LEVEL_4S` by removing the comment surrounding the definition.

```
/* If you want to activate level-4p functions, remove this comment surrounding
   the line below.
#define OH_LIB_LEVEL_4P
*/
/* If you want to activate level-4s functions, remove this comment surrounding
   the line below.
#define OH_LIB_LEVEL_4S
*/
#ifdef  OH_LIB_LEVEL_4P
#define OH_LIB_LEVEL_4PS
#endif
#ifdef  OH_LIB_LEVEL_4S
#define OH_LIB_LEVEL_4PS
#endif
```

The final contents of oh_config.h is the definition of `OH_LIB_LEVEL` to control the level-dependent function/subroutine name aliases which we will discuss in §3.12. You may edit the following line to define it so that it has 1, 2 or 3 representing the layer you choose unless you use the level-4p/4s extension. Otherwise, i.e., with level-4p/4s extension, `OH_LIB_LEVEL` is set to 4 and you have options to `#define` the following two macros which we will discuss in §3.7.

- `OH_BIG_SPACE` for your simulation with a significantly large space domain.
- `OH_NO_CHECK` for your well-debugged simulation code which does not need the argument consistency check in some API functions.

.

```
#ifdef  OH_LIB_LEVEL_4PS
#define OH_LIB_LEVEL 4
/* If you want to use level-4p/4s functions with a large simulation space,
```

```
   remove this comment surrounding the line below.
#define OH_BIG_SPACE
*/
/* If you want to let level-4p/4s's particle mapping functions run without
   checking the consistency of their arguments, remove this comment
   surrounding the line below.
#define OH_NO_CHECK
*/
#else
#ifndef OH_LIB_LEVEL
#define OH_LIB_LEVEL 3
#endif
#endif
```

3.4 Level-1 Library Functions

Level-1 library provides the following functions.

`oh1_init()` receives fundamental parameters and arrays by which the library interacts with your simulator body, and initializes internal data structures.

`oh1_neighbors()` receives an array through which the neighbors of the local node is reported each time the helpand-helper tree is reconfigured.

`oh1_families()` receives two arrays through which the configuration of all families is reported each time the helpand-helper tree is reconfigured.

`oh1_transbound()` implements the core algorithm of OhHelp and reports the particle transfer schedule.

`oh1_accom_mode()` shows whether particle accommodation by nodes are normal or anywhere, i.e., particles accommodated by a node are in the subdomains assigned to the node and in the neighbors of them, or not.

`oh1_broadcast()` performs broadcast communication in helpand-helper families.

`oh1_all_reduce()` performs all-reduce communication in helpand-helper families.

`oh1_reduce()` performs simple one-way reduce communication in helpand-helper families.

`oh1_init_stats()`
`oh1_stats_time()`
`oh1_show_stats()`
`oh1_print_stats()`
See §3.10 for the functions for statistics above.

`oh1_verbose()` is explained in §3.11.

The function API for Fortran programs is given by the module named `ohhelp1` in the file oh_mod1.F90, while API for C is embedded in ohhelp_c.h.

3.4.1 `oh1_init()`

The function (subroutine) `oh1_init()` receives a few fundamental parameters and arrays through which `oh1_transbound()` interacts with your simulator body. It also initializes internal data structures used in level-1 library. Among its thirteen arguments, other library functions directly refer to only the contents of the argument array `nphgram` as their implicit inputs. Therefore, after the call of `oh1_init()`, modifying the *bodies* of other arguments has no effect to library functions.

Fortran Interface

```
subroutine oh1_init(sdid, nspec, maxfrac, nphgram, totalp, rcounts, &
                    scounts, mycomm, nbor, pcoord, stats, repiter, verbose)
  use oh_type
  implicit none
  integer,intent(out)   :: sdid(2)
  integer,intent(in)    :: nspec
  integer,intent(in)    :: maxfrac
  integer,intent(inout) :: nphgram(:,:,:)
  integer,intent(out)   :: totalp(:,:)
  integer,intent(out)   :: rcounts(:,:,:)
  integer,intent(out)   :: scounts(:,:,:)
  type(oh_mycomm),intent(out) :: mycomm
  integer,intent(inout) :: nbor(3,3,3)        ! for 3D codes.
  integer,intent(in)    :: pcoord(OH_DIMENSION)
  integer,intent(in)    :: stats
  integer,intent(in)    :: repiter
  integer,intent(in)    :: verbose
end subroutine
```

`sdid(2)` will have the identifiers of primary and secondary subdomain of the local node in `sdid(1)` and `sdid(2)` respectively. Therefore, `sdid(1)` is always equivalent to the MPI rank number of the calling process. On the other hand, `sdid(2)` intially has -1 to mean we are in primary mode initially, but will be set to a non-negative number in $[0, N-1]$ to identify the secondary subdomain by `oh1_transbound()` if it turns the mode to secondary. Note that, even in secondary mode, `sdid(2)` may have -1 if the local node is the root of the helpand-helper tree.

`nspec` should have the number of species S.

`maxfrac` should have the tolerance factor percentage of load imbalance α greater than 0 and less than 100.

`nphgram(`N`,`S`,2)` should be an array whose element `nphgram(`m+1`,`s`,`c`)` should have the number of particles residing in the subdomain $m \in [0, N-1]$ categorized in the species $s \in [1, S]$ and accommodated by the local node as its primary ($c = 1$) or secondary ($c = 2$) ones. The contents of the array can be undefined at the call of `oh1_init()` but must be completely defined at the call of `oh1_transbound()`. Upon returning from `oh1_init()` and `oh1_transbound()`, the contents of the array will be zero-cleared, so that you can (re)start counting.

`totalp(`S`,2)` should be an array whose element `totalp(`s`,`c`)` will have the number of primary ($c = 1$) or secondary ($c = 2$) particles of species s to be accommodated by

the local node as the result of load balancing performed by `oh1_transbound()`. Note that `oh1_init()` does not give any values to the array.

`rcounts(`N`,`S`,2)` should be an array whose element `rcounts(`m`+1,`s`,`c`)` will have the number of particles of species s which the local node should receive from the node m as primary ($c = 1$) or secondary ($c = 2$) ones of the local node, after each call of `oh1_transbound()`. Remember that `rcounts(`n`+1,`s`,`c`)` for the local node n itself can be non-zero when it has particles residing in its primary (secondary) subdomain moving to its secondary (primary) subdomain.

`scounts(`N`,`S`,2)` should be an array whose element `scounts(`m`+1,`s`,`c`)` will have the number of particles of species s which the local node should send to the node m as primary ($c = 1$) or secondary ($c = 2$) ones of the node m (not of the local node), after each call of `oh1_transbound()`. Remember that `scounts(`n`+1,`s`,`c`)` for the local node n itself can be non-zero when it has particles residing in its primary (secondary) subdomain moving to its secondary (primary) subdomain.

`mycomm` should be a structured data of `oh_mycomm` type whose definition is given in `oh_type.F90` as a part of the module named `oh_type` and will have the following integers when `oh1_transbound()` (re)builds a new helpand-helper configuration.

`prime` is the MPI communicator for the family which the local node belongs to as the helpand, or `MPI_COMM_NULL` if it is a leaf of the helpand-helper tree.

`sec` is the MPI communicator for the family which the local node belongs to as a helper, or `MPI_COMM_NULL` if it is the root of the helpand-helper tree.

`rank` is the rank of the local node in the `prime` communicator, or -1 if it is a leaf.

`root` is the rank of the helpand node in the `sec` communicator, or -1 if the local node is the root.

`black` is 0 if the `prime` communicator is colored red, or 1 if colored black.

That is, `oh_mycomm` is defined as follows.

```
type oh_mycomm
  sequence
  integer :: prime, sec, rank, root, black
end type oh_mycomm
```

`nbor(3,...,3)` should be a D-dimensional array of three elements for each dimension and its element `nbor(`ν_1`,...,`ν_D`)` ($\nu_d \in [1, 2, 3]$) must have the identifier of the subdomain adjacent to the primary subdomain of the local node. More specifically, let $(\pi_1, \ldots, \pi_D)$ be the coordinates for the local node in a conceptual D-dimensional integer coordinate system in which computational nodes (or equivalently their primary subdomains) are laid out, and $rank(\pi'_1, \ldots, \pi'_D)$ be the function to map the grid point $(\pi'_1, \ldots, \pi'_D)$ to the identifier (MPI rank) of the node located at the point. With these definitions, an element of the array `nbor` should have the following (Figure 6).

$$\texttt{nbor(}\nu_1\texttt{,...,}\nu_D\texttt{)} = rank(\pi_1 + \nu_1 - 2,\ \ldots,\ \pi_D + \nu_D - 2)$$

If $D = 3$, for example, `nbor(1,1,1)` should have the idenetifier of the *neighbor* node whose primary subdomain contacts with that of the local node only at its west-south-bottom corner, `nbor(1,2,3)` should be for the node which shares west-top edge of

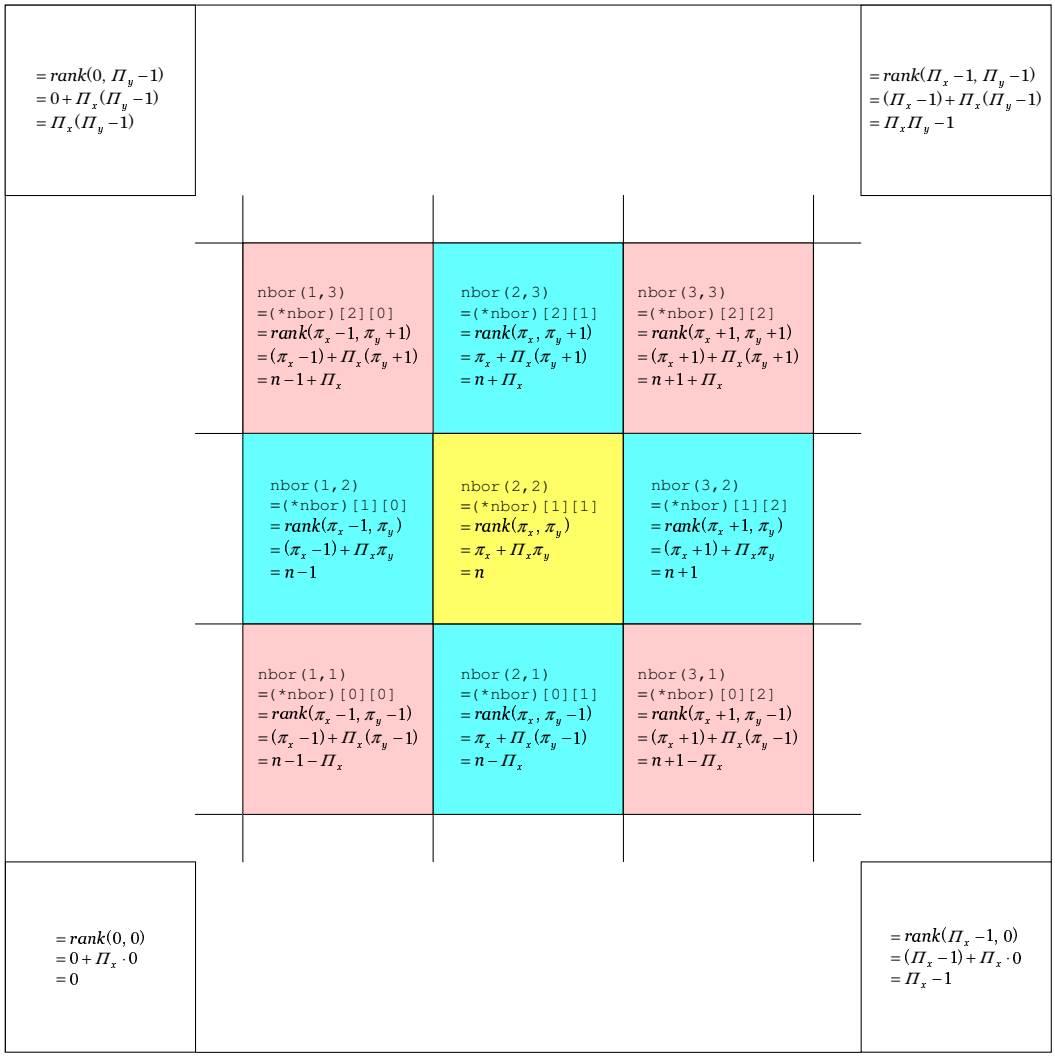


Figure 6: `nbor` and its default setting in $\Pi_x \times \Pi_y$ node coordinate system given by `pcoord`.

the local node, `nbor(3,2,2)` should be the east neighbor of the local node, and `nbor(2,2,2)` is the local node itself.

Note that the neighboring relationship may or may not be periodic along each axis. That is, if the node coordinate system is $[0, \Pi_x-1] \times [0, \Pi_y-1] \times [0, \Pi_z-1]$ and the local node is located at $(0, 0, 0)$, it may have west neighbor $(\Pi_x-1, 0, 0)$ while its south neighbor can be nonexistent. In the latter case for nonexistent neighbors, `nbor` can have elements being -2 (or less) to indicate that the corresponding neighboring grid points have no nodes. Also note that nonexistent neighbors can be found not only *outside* the node coordinate system but also in its *inside* for, e.g., *holes*.

Alternatively, if the work to define `nbor` is tiresome for you, you may delegate it to `oh1_init()` by making `nbor(1,...,1)` $= -1$, and giving the size of node coordinate system $\Pi_1 \times \cdots \times \Pi_D = N$ through the argument array `pcoord(`D`)=(/`Π_1`,...,`Π_D`/)`. In this case, `oh1_init()` initializes `nbor` assuming fully periodic coordinate system of $[0, \Pi_1-1] \times \ldots [0, \Pi_D-1]$ and $r = rank(\pi_1, \ldots, \pi_D)$ is given as follows.

$$r_D = \pi_D \qquad r_d = r_{d+1}\Pi_d + \pi_d \qquad r = r_1$$

`pcoord(`D`)` should be an array whose element `pcoord(`d`)` has the size of the d-th dimension Π_d of the conceptual integer coordnate system of $[0, \Pi_1-1] \times \cdots \times [0, \Pi_D-1]$ in which $N = \Pi_1 \times \cdots \times \Pi_D$ computational nodes are layed out, if you delegate the setting of the array `nbor(3,...,3)` to `oh1_init()`. Otherwise, the array can have any values.

`stats` defines how statistics data is collected. See §3.10 for more details.

`repiter` defines how frequently statistics data is reported when `stats` $= 2$. See §3.10 for more details.

`verbose` defines how verbosely the execution progress is reported. See §3.11 for more details.

C Interface

```
void oh1_init(int **sdid, int nspec, int maxfrac, int **nphgram,
              int **totalp, int **rcounts, int **scounts, void *mycomm,
              int **nbor, int *pcoord, int stats, int repiter, int verbose);
```

`**sdid` should be a double pointer to an array of two elements, or a pointer to `NULL` (not `NULL` itself) to order `oh1_init()` to allocate the array and *return* the pointer to it through the argument. The array will have the identifiers of primary and secondary subdomains of the local node in `(*sdid)[0]` and `(*sdid)[1]` respectively. Therefore, `(*sdid)[0]` is always equivalent to the MPI rank number of the calling process. On the other hand, `(*sdid)[1]` intially has -1 to mean we are in primary mode initially, but will be set to a non-negative number in $[0, N-1]$ to identify the secondary subdomain by `oh1_transbound()` if it turns the mode to secondary. Note that, even in secondary mode, `(*sdid)[1]` may have -1 if the local node is the root of the helpand-helper tree.

`nspec` should have the number of species S.

`maxfrac` should have the tolerance factor percentage of load imbalance α greater than 0 and less than 100.

**nphgram should be a double pointer to an array of $2 \times S \times N$ elements to form nphgram[2][S][N] conceptually, or a pointer to NULL (not NULL itself) to order oh1_init() to allocate the array and return the pointer to it through the argument. Its element nphgram[c][s][m][6] has the number of particles residing in the subdomain $m \in [0, N-1]$, categorized in the species $s \in [0, S-1]$ and accommodated by the local node as its primary ($c = 0$) or secondary ($c = 1$) ones. The contents of the array can be undefined at the call of oh1_init() but must be completely defined at the call of oh1_transbound(). Upon returning from oh1_init() and oh1_transbound(), the contents of the array will be zero-cleared, so that you can (re)start counting.

**totalp should be a double pointer to an an array of $2 \times S$ elements to form totalp[2][S] conceptually, or a pointer to NULL (not NULL itself) to order oh1_init() to allocate the array and return the pointer to it through the argument. Its element totalp[c][s] will have the number of primary ($c = 0$) or secondary ($c = 1$) particles of species s to be accommodated by the local node as the result of load balancing performed by oh1_transbound(). Note that oh1_init() does not give any values to the array.

**rcounts should be a double pointer to an array of $2 \times S \times N$ elements to form rcounts[2][S][N] conceptually, or a pointer to NULL (not NULL itself) to order oh1_init() to allocate the array and return the pointer to it through the argument. Its element rcounts[c][s][m] will have the number of particles of species s which the local node should receive from the node m as primary ($c = 0$) or secondary ($c = 1$) ones of the local node, after each call of oh1_transbound(). Remember that rcounts[c][s][n] for the local node n itself can be non-zero when it has particles residing in its primary (secondary) subdomain moving to its secondary (primary) subdomain.

**scounts should be a double pointer to an array of $2 \times S \times N$ elements to form scounts[2][S][N] conceptually, or a pointer to NULL (not NULL itself) to order oh1_init() to allocate the array and return the pointer to it through the argument. Its element scounts[c][s][m] will have the number of particles of species s which the local node should sent to the node m as primary ($c = 0$) or secondary ($c = 1$) ones of the node m (not of the local node), after each call of oh1_transbound(). Remember that scounts[c][s][n] for the local node n itself can be non-zero when it has particles residing in its primary (secondary) subdomain moving to its secondary (primary) subdomain.

*mycomm should be a pointer to a structured data named S_mycommc whose definition is given in ohhelp_c.h. Alternatively, it can be NULL (itself) if you do not want to bother to play with family communicators but use only library functions for collective communications among family members. If you give the pointer to a S_mycommc structure, you will have the followings when oh1_transbound() (re)builds a new helpand-helper configuration.

MPI_comm prime is the MPI communicator for the family which the local node belongs to as the helpand, or MPI_COMM_NULL if it is a leaf of the helpand-helper tree.

[6] For the sake of conciseness, an element of conceptual n-dimensional array a of $m_0 \times \cdots m_{n-1}$ elements, which is one-dimensional in reality with ANSI-C, is denoted by a[i_0]...[i_{n-1}] which should be a[j] in reality where j is defined by $j_0 = i_0$, $j_k = i_k + j_{k-1} m_k$, $j = j_{n-1}$. Therefore nphgram[c][s][m] is (*nphgram)[$m + N(s + Sc)$] in reality with ANSI-C, while the three-dimensional notation can be used with C99.

`MPI_comm sec` is the MPI communicator for the family which the local node belongs to as a helper, or `MPI_COMM_NULL` if it is the root of the helpand-helper tree.

`rank` is the rank of the local node in the `prime` communicator, or -1 if it is a leaf.

`root` is the rank of the helpand node in the `sec` communicator, or -1 if the local node is the root.

`int black` is 0 if the `prime` communicator is colored red, or 1 if colored black.

That is, `S_mycommc` is defined as follows.

```
struct S_mycommc {
  MPI_Comm prime, sec;
  int rank, root, black;
};
```

`**nbor` should be a double pointer to an array of 3^D elements to form `nbor`$[3]\dots[3]$ conceptually, or a pointer to `NULL` (not `NULL` itself) if you want the library to allocate and initialize the array and return the pointer to it through the argument. If you prepare the array, its element `nbor`$[\nu_{D-1}]\dots[\nu_0]$ ($\nu_d \in [0,1,2]$) must have the identifier of the subdomain adjacent to the primary subdomain of the local node. More specifically, let $(\pi_0,\dots,\pi_{D-1})$ be the coordinates for the local node in a conceptual D-dimensional integer coordinate system in which computational nodes (or equivalently their primary subdomains) are laid out, and $rank(\pi'_0,\dots,\pi'_{D-1})$ be the function to map the grid point $(\pi'_0,\dots,\pi'_{D-1})$ to the identifier (MPI rank) of the node located at the point. With these definitions, an element of the array `nbor` should have the following (Figure 6).

$$\texttt{nbor}[\nu_{D-1}]\dots[\nu_0] = rank(\pi_0+\nu_0-1,\ \dots,\ \pi_{D-1}+\nu_{D-1}-1)$$

If $D=3$, for example, `nbor[0][0][0]` should have the idenetifier of the *neighbor* node whose primary subdomain contacts with that of the local node only at its west-south-bottom corner, `nbor[2][1][0]` should be for the node which shares west-top edge of the local node, `nbor[1][1][2]` should be the east neighbor of the local node, and `nbor[1][1][1]` is the local node itself.

Note that the neighboring relationship may or may not be periodic along each axis. That is, if the node coordinate system is $[0,\Pi_x-1]\times[0,\Pi_y-1]\times[0,\Pi_z-1]$ and the local node is located at $(0,0,0)$, it may have west neighbor $(\Pi_x-1,0,0)$ while its south neighbor can be nonexistent. In the latter case for nonexistent neighbors, `nbor` can have elements being -2 (or less) to indicate that the corresponding neighboring grid points have no nodes. Also note that nonexistent neighbors can be found not only *outside* the node coordinate system but also in its *inside* for, e.g., *holes*.

Alternatively, if the work to define `nbor` is tiresome for you, you may delegate it to `oh1_init()` by passing a pointer to `NULL` or by making `**nbor` $= -1$, and giving the size of node coordinate system $\Pi_0\times\cdots\Pi_{D-1} = N$ through the argument array `pcoord`$[D]$ = `{`$\Pi_0,\dots,\Pi_{D-1}$`}`. In this case, `oh1_init()` initializes `(*nbor)` assuming fully periodic coordinate system of $[0,\Pi_1-1]\times\dots[0,\Pi_D-1]$ and $r = rank(\pi_0,\dots,\pi_{D-1})$ is given as follows.

$$r_{D-1} = \pi_{D-1} \qquad r_d = r_{d+1}\Pi_d + \pi_d \qquad r = r_0$$

`*pcoord` should be a pointer to an array of D elements and each element `pcoord`[d] should have the size of the d-th dimension Π_d of the conceptual integer coordinate system of $[0, \Pi_0-1] \times \cdots \times [0, \Pi_{D-1}-1]$ in which $N = \Pi_0 \times \cdots \times \Pi_{D-1}$ computational nodes are layed out, if you delegate the setting of the array `(*nbor)`[3^D] to `oh1_init()`. Otherwise, `pcoord` can be `NULL` or the array can have any values.

`stats` defines how statistics data is collected. See §3.10 for more details.

`repiter` defines how frequently statistics data is reported when `stats` = 2. See §3.10 for more details.

`verbose` defines how verbosely the execution progress is reported. See §3.11 for more details.

3.4.2 `oh1_neighbors()`

The function (subroutine) `oh1_neighbors()` receives an array `nbor` through which `oh1_transbound()` will report the neighbors of the local nodes to your simulator body.

Fortran Interface

```
subroutine oh1_neighbors(nbor)
  implicit none
  integer,intent(inout) :: nbor(3,3,3,3)      ! for 3D codes.
end subroutine
```

C Interface

```
void oh1_neighbors(int **nbor);
```

`nbor` should be a $(D+1)$-dimensional array `nbor(3,...,3,3)` in Fortran or a double pointer to an array of $3 \cdot 3^D$ elements to form `nbor`[3]...[3][3] conceptually in C. When $D = 3$ for example, `nbor(:,:,:,1)` or `nbor`[0][][][] will always have what ν`(:,:,:)` or ν[][][] has where ν is the array which you gave to `oh1_init()` (or its higher-level counterpart) through its argument `nbor`. On the other hand, `nbor(:,:,:,2)` or `nbor`[1][][][] will have ν`(:,:,:)` or ν[][][] in the helpand of the local node to show you the neighbors of its secondary subdomain, when we are in secondary mode. In addition, `nbor(:,:,:,3)` or `nbor`[2][][][] will have what `nbor(:,:,:,2)` or `nbor`[1][][][] had just before you call `oh1_transbound()` (or its higher-level counterpart) which returns -1 to mean helpand-helper configuration is (re)built. That is, `nbor(:,:,:,3)` or `nbor`[2][][][] has neighbors of the *old* secondary subdomain which the local node was responsible for *before* the helpand-helper reconfiguration.

The function helps you to find a subdomain adjacent to the local node's primary and, particularly, secondary subdomains. For example, if you find a set of secondary particles crossing the west-top edge of the secondary subdomain, you will know the destination subdomain looking up `nbor(1,2,3,2)` or `nbor[1][2][1][0]` if the last `oh1_transbound()` returns 1, while `nbor(1,2,3,3)` or `nbor[2][2][1][0]` will show you the destination if the return value is -1 because the node is still responsible for sending the particles crossing a boundary of the old secondary subdomain.

As described above, the argument array `nbor` has a tight relationship with the array ν being `nbor` of `oh1_init()`. More specifically, the relationship is maintained as follows.

- The simplest way is to give the same array to `oh1_init()` and `oh1_neighbors()`. For example, if your Fortran array is `myneighbor(3,3,3,3)`, you may give `myneighbor(:,:,:,1)` to `oh1_init()` and then `myneighbor(:,:,:,:)` to `oh1_neighbors()`. If your simulator is written in C and your array is `myneighbor[3][3][3][3]` on the other hand, both functions will work perfectly well receiving `(int**)(&&myneighbor[0][0][0][0])` commonly. Alternatively, a simulator in C may give a double pointer `to_myneighbor` pointing `NULL` to `oh1_init()` to allocate an array of $3 \cdot 3^3$ integers, and then give the same pointer to `oh1_neighbors()` to have the access to the array through `*to_myneighbor`.
- If you have some reason to have two arrays, say `nbor_a` and `nbor_b`, for `oh1_init()` and `oh1_neighbors()` respectively, the contents of `nbor_a(:,:,:)` or `*nbor_a`[][][] are copied into `nbor_b(:,:,:,1)` or `*nbor_b`[0][][][] by `oh1_neighbors()` automatically. In this case, your C simulator may give a double pointer `nbor_b` such that `*nbor_b` = `NULL` to let `oh1_neighbors()` allocate the array and to let `*nbor_b` point the head of the array.
- Though it is recommended to call `oh1_neighbors()` after the call of `oh1_init()`, you may call the function before the call of `oh1_init()`. If you do it, the array given to `oh1_neighbors()` is initialized by `oh1_init()` consistently.

Note that `nbor(:,:,:,2)` or `nbor`[1][][][] is meaningless when the local node does not have a secondary subdomain, except for the timing the mode is switched from secondary to primary by the last `oh1_transbound()`. In this critical timing, the subarray remembers the neighbors of the old secondary subdomain to be released from the local node. Similarly, `nbor(:,:,:,3)` or `nbor`[2][][][] is meaningless when the last `oh1_transbound()` did not returns -1, or the local node did not have a secondary subdomain before the call even if the return value was -1.

3.4.3 `oh1_families()`

The function (subroutine) `oh1_families()` receives arrays `famindex` and `members` through which `oh1_transbound()` will report the configuration of all families to your simulator body each time the helpand-helper tree is reconfigured.

Fortran Interface

```
subroutine oh1_families(famindex, members)
  implicit none
  integer,intent(inout) :: famindex(:)
  integer,intent(inout) :: members(:)
end subroutine
```

C Interface

```
void oh1_families(int **famindex, int **members);
```

`famindex` should be a one-dimensional array of $N+1$ elements (or larger) in Fortran, or a double pointer to the array in C, to have indices of the array `members` below.

`members` should be a one-dimensional array of $2N$ elements (or larger) in Fortran, or a double pointer to the array in C, to have ranks of the members of all families as described below.

Note that a simulator body in C may give double pointers to `NULL` for the arguments above to let the library allocate the arrays.

As discussed in §2.3, in secondary mode, each subdomain m has a family of nodes $F(m) = m \cup H(m)$ where $H(m)$ is the set of helpers for m and satisfies;

$$\bigcap_{m=0}^{N-1} H(m) = \emptyset \qquad \bigcup_{m=0}^{N-1} H(m) = [0, N) - \{r\}$$

with a *root* node r. The arrays `famindex` and `members` represent $F(m)$ for all $m \in [0, N)$ as follows.

$$\texttt{famindex}(m{+}1) = \texttt{famindex}[m] = i_m = \sum_{j=0}^{m-1} |F(j)|$$
$$\texttt{members}(i_m{+}1) = \texttt{members}[i_m] = m$$
$$\{\texttt{members}(k) \,|\, i_m{+}1 < k \le i_{m+1}\} = \{\texttt{members}[k] \,|\, i_m{+}1 \le k < i_{m+1}\} = H(m)$$

Note that $\texttt{famindex}(N+1) = \texttt{famindex}[N] = 2N-1$ to make $i_{m+1} - i_m = |F(m)|$ for all $m \in [0, N)$ because $\sum_{m=0}^{N-1} |F(m)| = 2N-1$ always. Moreover, the last element of `members` namely $\texttt{members}(2N) = \texttt{members}[2N{-}1]$ has r being the rank of root node. Therefore, for any subdomain m you can identify all members in $F(m)$ by scanning elements $\texttt{members}(i_m{+}1), \ldots, \texttt{members}(i_{m+1})$ or $\texttt{members}[i_m], \ldots, \texttt{members}[i_{m+1}{-}1]$. Moreover, you can traverse the helpand-helper tree from the root r.

The two arrays represent $F(m)$ even when we are in primary mode in which $F(m) = \{m\}$ for all m resulting in $\texttt{famindex}(m{+}1) = \texttt{members}(m{+}1) = m$ or $\texttt{famindex}[m] = \texttt{members}[m] = m$.

When you try to perform inter-node particle transfer by yourself, you will consult the two arrays and `nbor` given to `oh1_neighbors()` to find the family members of the subdomain adjacent to the primary or secondary subdomain of the local node. For example, the following code snip is to send particles in a one-dimensional array `sbuf`. In this example, it is supposed that primary (`ps` = 0) or secondary (`ps` = 1) particles of species `s` moving to (or staying in) the neighbor subdomain identified by `x`, `y` and `z` are in the region in `sbuf` from the index `head(x,y,z,s,ps+1)` or `head[ps][s][z][y][x]`, and the number of particles to be sent to a node `m` $\in [0, N)$ is given by `scounts(m+1,s,ps+1)` or `scounts[ps][s][m]` being an argument array of `oh1_init()`.

```
! Fortran
r=1
do ps=1,2
  do s=1,nspec
    do z=1,3;  do y=1,3;  do x=1,3
      n=neighbor(x,y,z,ps)
      from=famindex(n+1)+2;  to=famindex(n+2)
      h=head(x,y,z,s,ps);  c=scounts(n+1,s,1)
      call MPI_Isend(sbuf(h), c, ptype, n, tag, MPI_COMM_WORLD, req(r), e)
      h=h+c;  r=r+1
      do k=from,to
        m=members(k);  c=scounts(m+1,s,2)
        call MPI_Isend(sbuf(h), c, ptype, m, tag, MPI_COMM_WORLD, req(r), e)
        h=h+c;  r=r+1
      end do
    end do;  end do;  end do;
```

```
  end do
end do

//C
r=0;
for (ps=0;ps<2;ps++)
  for (s=0;s<nspec;s++)
    for (z=0;z<3;z++)  for (y=0;y<3;y++)  for (x=0;x<3;x++) {
      n=neighbor[ps][z][y][x];
      from=famindex[n]+1;  to=famindex[n+1];
      h=head[ps][s][z][y][x];  c=scounts[0][s][n];
      MPI_Isend(sbuf+h, c, ptype, n, tag, MPI_COMM_WORLD, req[r]);
      h+=c;  r++;
      for (k=from; k<to; k++) {
        m=members[k];  c=scounts[1][s][m];
        MPI_Isend(sbuf+h, c, ptype, m, tag, MPI_COMM_WORLD, req[r]);
        h+=c;  r++;
      }
    }
```

3.4.4 oh1_transbound()

The function `oh1_transbound()` performs global collective communications of `nphgram` to examine whether the number of particles in nodes are well balanced. If it finds load imbalance is unacceptably large, it (re)builds helpand-helper configuration updating the structure `mycomm`. In addition, if `oh1_neighbors()` and/or `oh1_families()` have been called prior to this function, it updates the arrays given to these functions (subroutines) to show your simulator body the new neighbors and families corresponding to the new helpand-helper configuration. Finally, it makes particle transfer schedule to report it through the arrays `rcounts` and `scounts`, and updates the array `totalp` so that the array has the number of particles accommodated by the local node *after* the particle transfer. It also makes `nphgram` zero-cleared to give initial values of particle counting in the next simulation step. Note that the arrays `nphgram`, `rcounts`, `scounts` and `totalp` and the structure `mycomm` were given to `oh1_init()` as its arguments.

Besides these *global* arrays and structure, `oh1_transbound()` takes two arguments and returns an integer value to show you the mode in the next simulation step, as follows.

Fortran Interface

```
integer function oh1_transbound(currmode, stats)
  implicit none
  integer,intent(in) :: currmode
  integer,intent(in) :: stats
end function
```

C Interface

```
int oh1_transbound(int currmode, int stats);
```

`currmode` should have an integer in $[0, 1]$ to represent current execution mode as follows.

- 0 means we are in primary mode.

- 1 means we are in secondary mode.

`stats` inactivates statistics collection if 0, regardless the specification given by `stats` argument of `oh1_init()`. You may inactivate the statistics collection temporarily on, for example, the first call of `oh1_transbound()` for initial load balancing as discussed in §3.10.

return value is an integer in $\{-1, 0, 1\}$ to represent the execution mode in the next simulation step as follows.

- -1 means helpand-helper configuration is (re)built and thus we will be in secondary mode. This also means that you have to broadcast field-arrays from helpands to their helpers if their replications are necessary for helpers.
- 0 means we will be in primary mode.
- 1 means we will be in secondary mode but helpand-helper configuration has been kept.

Usually, telling the current execution mode and receiving that in the next simulation step to/from `oh1_transbound()` is easily implmented by having your own `currmode` variable. That is, the following should be necessary and sufficient.

- Give 0 to `oh1_transbound()` at the first call because you have not yet built helpand-helper configuration even if the initial particle distribution causes an unacceptable load imbalance.
- Let your own `currmode` be the return value. If it is negative, let it be 1 and broadcast field-arrays if necessary. Then give it to `oh1_transbound()` on the second call and repeat this for successive calls.

3.4.5 `oh1_accom_mode()`

The function `oh1_accom_mode()` shows its caller whether particle accommodation by nodes are *normal* or *anywhere* through its return value. That is, if all nodes have its primary and secondary particles in its corresponding primary and secondary subdomains or the neighbor of the subdomains, the function returns 0 to indicate normal accommodation with which, for example, your own particle transfer mechanism may exchange particles in a local node only with its family members and those of neighbors. Otherwise, i.e., if a node has a particle residing in a subdomain other than its primary or secondary subdomain or a neighbor of the subdomain due to initial particle distribution, particle warp, particle injection into an arbitrary position, and so on, the function returns 1 to indicate anywhere accommodation which requires an all-to-all-type global communication for particle transfer.

Note that the accommodation mode is according to the last call of `oh1_transbound()` and, if it made the helpand-helper (re)configuration, to the subdomain assignments *before* the (re)configuration. Therefore in normal accmmodation, if we were in secondary mode and the helpand-helper reconfiguration took place, a node may have secondary particles in its *old* secondary subdomain and the neighbors of the subdomain to be sent to the members of the *new* families for the subdomains. Similarly, if we were in secondary mode but in primary mode now, a node may have secondary particles in its *old* secondary subdomain and the neighbors of the subdomain to be sent to the nodes responsible for the subdomains as their primary subdomains.

Fortran Interface

```
integer function oh1_accom_mode()
  implicit none
end function
```

C Interface

```
int  oh1_accom_mode();
```

return value is an integer in $\{0, 1\}$ to represent the accmmodation mode either of normal (0) or anywhere (1).

3.4.6 `oh1_broadcast()`

The function (subroutine) `oh1_broadcast()` performs red-black broadcast communications in the families the local node belongs to. The arguments of the function `pbuf`, `pcount` and `ptype` specify the data to be broadcasted in the *primary family* which the local node belongs to as the helpand, while `sbuf`, `scount` and `stype` are for the data to be broadcasted in the *secondary family* which the local node belongs to as a helper, as shown in Figure 7. You may be unaware that the local node really has its primary or secondary family, because the function will skip the primary broadcast if it is a leaf and the secondary one if it is the root.

Fortran Interface

```
subroutine oh1_broadcast(pbuf, sbuf, pcount, scount, ptype, stype)
  implicit none
  real*8,intent(in)  :: pbuf
  real*8,intent(out) :: sbuf
  integer,intent(in) :: pcount
  integer,intent(in) :: scount
  integer,intent(in) :: ptype
  integer,intent(in) :: stype
end subroutine
```

C Interface

```
void oh1_broadcast(void *pbuf, void *sbuf, int pcount, int scount,
                   MPI_Datatype ptype, MPI_Datatype stype);
```

`pbuf`[7] should be (the pointer to) the first element of the data buffer which the local node broadcasts to its helpers in its primary family.

`sbuf` should be (the pointer to) the first element of the data buffer to receive data broadcasted in the secondary family.

`pcount` should have the number of `ptype` elements to be broadcasted in the primary family. This value should match `scount` of the call in the helpers.

[7] In the Fortran module file `oh_mod1.F90`, the arguments `pbuf` and `sbuf` of `oh1_broadcast()`, `oh1_all_reduce()` and `oh1_reduce()` are declared as `real*8` type hoping it matches the type of the elements in your buffers. If this is incorrect, feel free to modify the declaration or to remove it, so that your compiler accept your calls of the library subroutines.

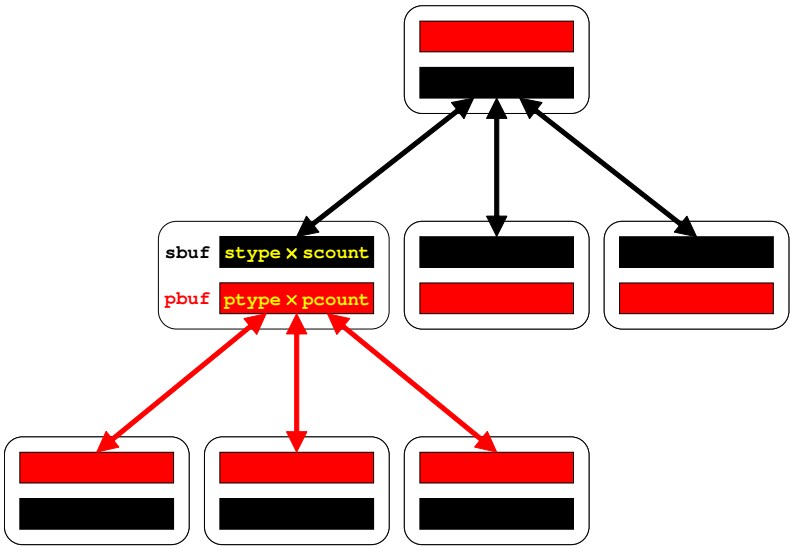


Figure 7: Red-black collective communication by `oh1_broadcast()`, `oh1_all_reduce()` and `oh1_reduce()`.

`scount` should have the number of `stype` elements to be broadcasted in the seconary family. This value should match `pcount` of the call in the helpand.

`ptype` should have the MPI data-type of elements to be broadcasted in the primary family. This value should match `stype` of the call in the helpers.

`stype` should have the MPI data-type of elements to be broadcasted in the secondary family. This value should match `ptype` of the call in the helpand.

3.4.7 `oh1_all_reduce()`

The function (subroutine) `oh1_all_reduce()` performs red-black all-reduce communications in the families the local node belongs to. The arguments of the function `pbuf`, `pcount`, `ptype`, `pop` specify the data to be reduced in the primary family, while `sbuf`, `scount`, `stype` and `sop` are for the data to be reduced in the secondary family. You may be unaware that the local node really has its primary or secondary family, because the function will skip the primary reduction if it is a leaf and the secondary one if it is the root.

Fortran Interface

```
subroutine oh1_all_reduce(pbuf, sbuf, pcount, scount, ptype, stype, &
                          pop, sop)
  implicit none
  real*8,intent(inout) :: pbuf
  real*8,intent(inout) :: sbuf
  integer,intent(in)   :: pcount
  integer,intent(in)   :: scount
  integer,intent(in)   :: ptype
  integer,intent(in)   :: stype
  integer,intent(in)   :: pop
```

```
  integer,intent(in)   :: sop
end subroutine
```

C Interface

```
void oh1_all_reduce(void *pbuf, void *sbuf, int pcount, int scount,
                    MPI_Datatype ptype, MPI_Datatype stype,
                    MPI_Op pop, MPI_Op sop);
```

`pbuf` should be (the pointer to) the first element of the data buffer to be reduced in the primary family. The buffer is replaced with the reduction result.

`sbuf` should be (the pointer to) the first element of the data buffer to be reduced in the secondary family. The buffer is replaced with the reduction result.

`pcount` should have the number of `ptype` elements to be reduced in the primary family. This value should match `scount` of the call in the helpers.

`scount` should have the number of `stype` elements to be reduced in the secondary family. This value should match `pcount` of the call in the helpand.

`ptype` should have the MPI data-type of elements to be reduced in the primary family. This value should match `stype` of the call in the helpers.

`stype` should have the MPI data-type of elements to be reduced in the secondary family. This value should match `ptype` of the call in the helpand.

`pop` should have the MPI operator for the reduction in the primary family. This value should match `sop` of the call in the helpers.

`sop` should have the MPI operator for the reduction in the secondary family. This value should match `pop` of the call in the helpers.

3.4.8 `oh1_reduce()`

The function (subroutine) `oh1_reduce()` performs red-black simple one-way reduce communications in the families the local node belongs to. The arguments of the function `pbuf`, `pcount`, `ptype`, `pop` specify the data to be reduced in the primary family, while `sbuf`, `scount`, `stype` and `sop` are for the data to be reduced in the secondary family. You may be unaware that the local node really has its primary or secondary family, because the function will skip the primary reduction if it is a leaf and the secondary one if it is the root.

Fortran Interface

```
subroutine oh1_reduce(pbuf, sbuf, pcount, scount, ptype, stype, pop, sop)
  implicit none
  real*8,intent(inout) :: pbuf
  real*8,intent(in)    :: sbuf
  integer,intent(in)   :: pcount
  integer,intent(in)   :: scount
  integer,intent(in)   :: ptype
  integer,intent(in)   :: stype
  integer,intent(in)   :: pop
  integer,intent(in)   :: sop
end subroutine
```

C Interface

```
void oh1_reduce(void *pbuf, void *sbuf, int pcount, int scount,
                MPI_Datatype ptype, MPI_Datatype stype,
                MPI_Op pop, MPI_Op sop);
```

`pbuf` should be (the pointer to) the first element of the data buffer to be reduced in the primary family. The buffer is replaced with the reduction result.

`sbuf` should be (the pointer to) the first element of the data buffer to be reduced in the secondary family. The buffer will remain unchanged.

`pcount` should have the number of `ptype` elements to be reduced in the primary family. This value should match `scount` of the call in the helpers.

`scount` should have the number of `stype` elements to be reduced in the secondary family. This value should match `pcount` of the call in the helpand.

`ptype` should have the MPI data-type of elements to be reduced in the primary family. This value should match `stype` of the call in the helpers.

`stype` should have the MPI data-type of elements to be reduced in the secondary family. This value should match `ptype` of the call in the helpand.

`pop` should have the MPI operator for the reduction in the primary family. This value should match `sop` of the call in the helpers.

`sop` should have the MPI operator for the reduction in the secondary family. This value should match `pop` of the call in the helpand.

3.5 Level-2 Library Functions

Level-2 library provides the following functions.

`oh2_init()` performs initialization similar to what `oh1_init()` does and that of level-2's own for particle buffers.

`oh2_max_local_particles()` calculates the size of particle buffers.

`oh2_transbound()` performs load balancing similar to `oh1_transbound()` and transfers particles according to the schedule.

`oh2_inject_particle()` injects a particle to the bottom of the particle buffer.

`oh2_remap_injected_particle()` maintains library's internal state for an injected but not mapped particle.

`oh2_remove_injected_particle()` removes an injected particle maintaining library's internal state.

`oh2_set_total_particles()` tells the library you will inject/remove particles *before* your first call `oh2_transbound()`.

The function API for Fortran programs is given by the module named `ohhelp2` in the file oh_mod2.F90, while API for C is embedded in ohhelp_c.h.

3.5.1 Particle Data Type

Since `oh2_transbound()` and its higher-level counterparts transfer particles among nodes, they need to know how each particle is represented. The default configuration of the `struct` to represent a particle for C-coded simulator body and the library, namely `S_particle` is defined in the C header file oh_part.h, while its Fortran counterpart `oh_particle` is given in oh_type.F90. Both definitions are of course consistent with the following elements.

`x`, `y` and `z` are for the $x/y/z$ coordinates of the position at which a particle resides.

`vx`, `vy` and `vz` are for the $x/y/z$ components of the velocity of a particle.

`pid` is the unique identifier of a particle by which, for example, you can trace the trajectory of the particle.

`nid` is the identifier of the subdomain in which a particle resides.

`spec` is the identifier of the species which a particle belongs to.

In the elements listed above, `nid` is essential for the library and must have the identifier of the subdomain in which the particle resides at the call of `oh2_transbound()`. In addition, `spec` is also necessary if $S > 1$ and you inject particles by the library function `oh2_inject_particle()` or its level-4p/4s counterparts `oh4p_inject_particle()` or `oh4s_inject_particle()`, and must has a value in $[1, S]$ if your simulator is coded in Fortran, or in $[0, S-1]$ for C-coded simulators.

On the other hand, you may freely modify the definitions in oh_part.h and, if your simulator is coded in Fortran, oh_type.F90, by adding, removing and/or renaming other elements. However, if you use the level-4p/4s extension, `S_particle` in oh_part.h should have elements `x`, `y` and `z` (or the first one or two if $D < 3$), their type should be `double` or `float`, and oh_type.F90 should be consistent with them if you work with Fortran. As for `spec`, you may remove it toghether with `#define` of `OH_HAS_SPEC` if $S = 1$ or you use neither `oh2_inject_particle()`, `oh4p_inject_particle()` nor `oh4p_inject_particle()` and want to save four bytes for each particle.

Another caution to the user of level-4p/4s extension is that the `nid` element can be a 64-bit integer rathan than 32-bit if you made `OH_BIG_SPACE` defined in oh_config.h (see §3.3). C programmers should also notice that the element has type `OH_nid_t` which is defined as `long␣long␣int` or `int` when `OH_BIG_SPACE` is defined or not, respectively. We will revisit this issue in some further details in §3.7.

The verbatim definitions of `S_particle` and `oh_particle` are as follows.

```
#include "oh_config.h"
#ifdef  OH_BIG_SPACE
typedef long long int OH_nid_t;
#else
typedef int OH_nid_t;
#endif

struct S_particle {
  double x, y, z, vx, vy, vz;
  long long int pid;
  OH_nid_t nid;
  int spec;
};
#define OH_HAS_SPEC
```

```
  type oh_particle
    sequence
    real*8    :: x, y, z, vx, vy, vz
    integer*8 :: pid
#ifdef OH_BIG_SPACE
    integer*8 :: nid
#else
    integer   :: nid
#endif
    integer   :: spec
  end type
```

3.5.2 oh2_init()

The function (subroutine) oh2_init() receives a few fundamental parameters and arrays through which oh2_transbound() interacts with your simulator body. It also initializes internal data structures used in level-1 and level-2 libraries. Among its fourteen arguments, other library functions directly refer to only the bodies of the arguments nphgram and pbuf as their implicit inputs. Therefore, after the call of oh2_init(), modifying the bodies of other arguments has no effect to library functions.

Fortran Interface

```
      subroutine oh2_init(sdid, nspec, maxfrac, nphgram, totalp, &
                          pbuf, pbase, maxlocalp, mycomm, nbor, pcoord, &
                          stats, repiter, verbose)
        use oh_type
        implicit none
        integer,intent(out)   :: sdid(2)
        integer,intent(in)    :: nspec
        integer,intent(in)    :: maxfrac
        integer,intent(inout) :: nphgram(:,:,:)
        integer,intent(out)   :: totalp(:,:)
        type(oh_particle),intent(inout) :: pbuf(:)
        integer,intent(out)   :: pbase(3)
        integer,intent(in)    :: maxlocalp
        type(oh_mycomm),intent(out) :: mycomm
        integer,intent(inout) :: nbor(3,3,3)        ! for 3D codes.
        integer,intent(in)    :: pcoord(OH_DIMENSION)
        integer,intent(in)    :: stats
        integer,intent(in)    :: repiter
        integer,intent(in)    :: verbose
      end subroutine
```

C Interface

```
 void oh2_init(int **sdid, int nspec, int maxfrac, int **nphgram,
               int **totalp, struct S_particle **pbuf, int **pbase,
               int maxlocalp, void *mycomm, int **nbor,
               int *pcoord, int stats, int repiter, int verbose);
```

sdid
nspec

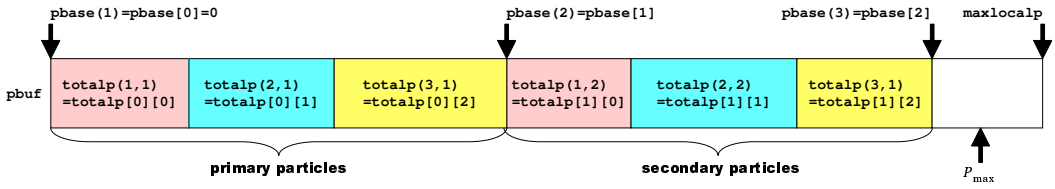


Figure 8: Particle buffer and related variables.

`maxfrac`
`nphgram`
`totalp`
: See §3.4.1 because the arguments above are perfectly equivalent to those of `oh1_init()`.

`pbuf(`P_{lim}`)` (for Fortran)
`**pbuf` (for C)
: The argument `pbuf` should be an one-dimensional array of `oh_particle` type structure elements in Fortran, while it should be a double pointer to an array of `S_particle` structure in C. The array have to be large enough to accommodate P_{lim} particles, where P_{lim} is given through the argument `maxlocalp` and should not be less than $P_{\max}$ at any time (Figure 8). In C code, `pbuf` can be a pointer to `NULL` (not `NULL` itself) to make `oh2_init()` allocate the buffer for you and return the pointer to it through the argument.

`pbase(3)` (for Fortran)
`**pbase` (for C)
: The argument `pbase` should be an one dimensional array of three elements in Fortran, while it should be a double pointer to such an array in C. After zero-cleared by `oh2_init()`, each call of `oh2_transbound()` make the array for the local node n have 0, Q_n^n and Q_n in this order to represent the zero-origin displacement of the first primary particle and the first secondary particle, and the head of unused region of `pbuf`. That is, the first Q_n^n portion of `pbuf` is used for primary particles, while the second $Q_n^{parent(n)} = Q_n - Q_n^n$ particles are for secondary particles. In C code, `pbase` can be a pointer to `NULL` (not `NULL` itself) to make `oh2_init()` allocate the array for you and return the pointer to it through the argument.

`maxlocalp` should have the absolute limit of the particle buffer `pbuf` and thus defines P_{lim}. You may ask the library function `oh2_max_local_particles()` to calculate P_{lim} from the system-wide absolute limit. Note that `oh2_init()` allocates a buffer for particle transfer and thus your machine should have memory large enough to have $2 \times P_{lim}$ particles per computation node.

`mycomm`
`nbor`
`pcoord`
`stats`
`repiter`
`verbose`
: See §3.4.1 because the arguments above are perfectly equivalent to those of `oh1_init()`.

Note that `oh2_init()` has neither arguments `rcounts` nor `scounts` which `oh1_init()` has, because particle transfer in `oh2_transbound()` makes it unnecessary to report transfer schedule.

3.5.3 `oh2_max_local_particles()`

The function `oh2_max_local_particles()` calculates the absolute maximum number of particles which a node can accommodate and returns it to its caller. The return value can be directly passed to the argument `maxlocalp` of `oh2_init()`.

Fortran Interface

```
integer function oh2_max_local_particles(npmax, maxfrac, minmargin)
  implicit none
  integer*8,intent(in) :: npmax
  integer,intent(in)   :: maxfrac
  integer,intent(in)   :: minmargin
end function
```

C Interface

```
int  oh2_max_local_particles(long long int npmax, int maxfrac,
                             int minmargin);
```

`npmax` should be the absolute maximum number of particles which your simulator is capable of as a whole.

`maxfrac` should have the tolerance factor percentage of load imbalance α and should be same as the argument `maxfrac` of `oh2_init()`.

`minmargin` should be the minimum margin by which the return value P_{lim} has to clear over the per node average of `npmax`.

return value is the number of particles P_{lim} given by the following.

$$\overline{P} = \lceil \texttt{npmax}/N \rceil \qquad P_{lim} = \max(\lceil \overline{P}(100+\alpha)/100 \rceil\,,\ \overline{P} + \texttt{minmargin})$$

Note that `minmargin` is the margin over $\overline{P}$ to be kept besides the tolerance factor α for, e.g., initial particle accommodation in each node. Therefore it does not assure that a node has a room for `minmargin` particles in simulation. If you need such a room for, e.g., particle injection, add the room to P_{lim} to give it the argument `maxlocalp` of `oh2_init()`.

3.5.4 `oh2_transbound()`

The function `oh2_transbound()` at first performs operetaions for load balancing as same as that `oh1_transbound()` does; examination of `nphgram` to check the balancing and (re)building of helpand-helper configuration updating `mycomm` if necessary. Then, instead of reporting the particle transfer schedule, it sends particles in `pbuf` to other nodes and receives them into `pbuf`, updates `totalp` and `pbase` according to the transfer result, and clears `nphgram` with zeros. Note that the arrays `nphgram`, `pbuf`, `totalp` and `pbase` and the structure `mycomm` were given to `oh2_init()` as its arguments.

The arguments of `oh2_transbound()` and its return value, besides these global arrays and structures, are perfectly equivalent to those of `oh1_transbound()` and thus see §3.4.4 for them.

Fortran Interface

```
integer function oh2_transbound(currmode, stats)
  implicit none
  integer,intent(in) :: currmode
  integer,intent(in) :: stats
end function
```

C Interface

```
int oh2_transbound(int currmode, int stats);
```

3.5.5 `oh2_inject_particle()`

The function (subroutine) `oh2_inject_particle()` injects a given particle at the bottom of `pbuf` and increase an element of `nphgram` according to its residence subdomain and species. Note that the number of particles injected in a simulation step should not be greater than $P_{lim} - Q_n$.

Fortran Interface

```
subroutine oh2_inject_particle(part)
  use oh_type
  implicit none
  type(oh_particle),intent(in) :: part
end subroutine
```

C Interface

```
void oh2_inject_particle(struct S_particle *part);
```

`part` (for Fortran)
`*part` (for C)
: The argument `part` should be a `oh_particle` structure in Fortran, or a pointer to `S_particle` structure in C, to be injected. Elements in the given particle structure should be completely set with significant values in advance, especially for `nid` and, if $S \neq 1$, `spec` elements which are referred to by the function to update `nphgram`. See §3.9 for further discussion on injection.

3.5.6 `oh2_remap_injected_particle()`

The function (subroutine) `oh2_remap_injected_particle()` maintains library's internal state of a particle injected by `oh2_inject_particle()` with `nid` element -1.

Fortran Interface

```
subroutine oh2_remap_injected_particle(part)
  use oh_type
  implicit none
  type(oh_particle),intent(in) :: part
end subroutine
```

C Interface

```
void oh2_remap_injected_particle(struct S_particle *part);
```

`part` (for Fortran)
`*part` (for C)
: The argument `part` should be `pbuf`$(Q_n + k)$ being `oh_particle` structure in Fortran, or a pointer `pbuf` $+ Q_n + k - 1$ to `S_particle` structure in C, for the k-th injected particle in a simulation step. Elements in the given particle structure should be completely set with significant values in advance, especially for `nid` and, if $S \neq 1$, `spec` elements which are referred to by the function to update `nphgram`. See §3.9 for further discussion on injection.

3.5.7 `oh2_remove_injected_particle()`

The function (subroutine) `oh2_remove_injected_particle()` removes a particle injected by `oh2_inject_particle()`.

Fortran Interface

```
subroutine oh2_remove_injected_particle(part)
  use oh_type
  implicit none
  type(oh_particle),intent(inout) :: part
end subroutine
```

C Interface

```
void oh2_remove_injected_particle(struct S_particle *part);
```

`part` (for Fortran)
`*part` (for C)
: The argument `part` should be `pbuf`$(Q_n + k)$ being `oh_particle` structure in Fortran, or a pointer `pbuf` $+ Q_n + k - 1$ to `S_particle` structure in C, for the k-th injected particle in a simulation step. Elements in the given particle structure should be completely set with significant values in advance, especially for `nid` and, if $S \neq 1$, `spec` elements which are referred to by the function to update `nphgram`. See §3.9 for further discussion on injection.

3.5.8 `oh2_set_total_particles()`

The function (subroutine) `oh2_set_total_particles()` tells the library that you will inject and/or remove particles *before* the first call of `oh2_transbound()`. This function consults the array `nphgram` which must be consistent with the contents of particle buffer `pbuf`, and updates (initializes) `totalp` according to `nphgram`.

Fortran Interface

```
subroutine oh1_set_total_particles
end subroutine
```

C Interface

```
void oh2_set_total_particles();
```

The library internally maintains a copy of `totalp` to know the layout of `pbuf` at the call of `oh2_transbound()` and update `totalp` and the copy upon its return. However, at the first call of `oh2_transbound()` the library does not know the layout and thus consults `nphgram` assuming it is consistent with the layout. This assumption is usually correct unless particles are injected/removed *before* the first call of `oh2_transbound()`. Therefore, if by some reason your simulator code needs to inject particles by `oh2_inject_particle()` and/or remove them by setting their `nid` to be -1 in initializing process, you have to set `nphgram` so that it describes the contents of `pbuf` correctly, then call this function `oh2_set_total_particles()` to let the library recognize the layout of `pbuf`, and then inject/remove particles before the first call of `oh2_transbound()`. Calling this function in other occasions are unnecessary but safe providing that `nphgram` correctly describes the layout of `pbuf`.

3.6 Level-3 Library Functions

Level-3 library provides the following functions.

`oh3_init()` performs initialization similar to what `oh2_init()` does and that of level-3's own for communications of field-arrays.

`oh13_init()` performs initialization similar to what `oh3_init()` does but excludes that for the particle buffer. That is, roughly speeking, `oh13_init()` is equal to `oh3_init()` minus `oh2_init()` plus `oh1_init()`.

`oh3_grid_size()` specifies the grid size of each dimension.

`oh3_transbound()` performs load balancing almost equivalent to `oh2_transbound()` or `oh1_transbound()` depending on the initializer you choose.

`oh3_map_particle_to_neighbor()` finds the subdomain which will be the residence of a boudary crossing particle and is neighboring to the primary or secondary subdomain of the local node, and then returns its identifier.

`oh3_map_particle_to_subdomain()` finds the subdomain which will be the residence of a boudary crossing particle and may be anywhere in the whole space domain, and then returns its identifier.

`oh3_bcast_field()` performs broadcast communication of a field-array in helpand-helper families.

`oh3_allreduce_field()` performs all-reduce communication of a field-array in helpand-helper families.

`oh3_reduce_field()` performs simple one-way reduce communication of a field-array in helpand-helper families.

`oh3_exchange_borders()` performs neighboring communication to exchange subdomain boundary data of a field-array.

The function API for Fortran programs is given by the module named `ohhelp3` in the file oh_mod3.F90, while API for C is embedded in ohhelp_c.h.

3.6.1 `oh3_init()`

The function (subroutine) `oh3_init()` receives a number of fundamental parameters and arrays through which `oh3_transbound()` and other subroutines/functions interacts with your simulator body. It also initializes internal data structures used in level-1, level-2 and level-3 libraries. Among its twenty-three (23!!) arguments, other library functions directly refer to only the bodies of the arguments `nphgram` and `pbuf` as their implicit inputs. Therefore, after the call of `oh3_init()`, modifying the bodies of other arguments has no effect to library functions.

Fortran Interface

```
subroutine oh3_init(sdid, nspec, maxfrac, nphgram, totalp, &
                    pbuf, pbase, maxlocalp, mycomm, nbor, pcoord, &
                    sdoms, scoord, nbound, bcond, bounds, ftypes, &
                    cfields, ctypes, fsizes, &
                    stats, repiter, verbose)
  use oh_type
  implicit none
  integer,intent(out)   :: sdid(2)
  integer,intent(in)    :: nspec
  integer,intent(in)    :: maxfrac
  integer,intent(inout) :: nphgram(:,:,:)
  integer,intent(out)   :: totalp(:,:)
  type(oh_particle),intent(inout) :: pbuf(:)
  integer,intent(out)   :: pbase(3)
  integer,intent(in)    :: maxlocalp
  type(oh_mycomm),intent(out) :: mycomm
  integer,intent(inout) :: nbor(3,3,3)        ! for 3D codes.
  integer,intent(in)    :: pcoord(OH_DIMENSION)
  integer,intent(inout) :: sdoms(:,:,:)
  integer,intent(in)    :: scoord(2,OH_DIMENSION)
  integer,intent(in)    :: nbound
  integer,intent(in)    :: bcond(2,OH_DIMENSION)
  integer,intent(inout) :: bounds(:,:,:)
  integer,intent(in)    :: ftypes(:,:)
  integer,intent(in)    :: cfields(:)
  integer,intent(in)    :: ctypes(:,:,:,:)
  integer,intent(out)   :: fsizes(:,:,:)
  integer,intent(in)    :: stats
  integer,intent(in)    :: repiter
  integer,intent(in)    :: verbose
end subroutine
```

sdid
nspec
maxfrac
nphgram

`totalp`
: See §3.4.1 because the arguments above are perfectly equivalent to those of `oh1_init()`.

`pbuf`

`pbase`

`maxlocalp`
: See §3.5.2 because the arguments above are perfectly equivalent to those of `oh2_init()`.

`mycomm`

`nbor`

`pcoord`
: See §3.4.1 because the arguments above are perfectly equivalent to those of `oh1_init()`.

`sdoms(2,`D`,`N`)` should be an array whose element `sdoms(`β`,`d`,`m`+1)` should have the d-th ($d \in [1, D]$) dimensional integer coordinate of the lower ($\beta = 1$) or upper ($\beta = 2$) boundary of the subdomain $m \in [0, N-1]$, namely $\delta^l_d(m)$ or $\delta^u_d(m)$ respectively. For example, for the 3-dimensional cuboid subdomain m whose grid points at west-south-east and east-north-top corners are ($\delta^l_x(m)$, $\delta^l_y(m)$, $\delta^l_z(m)$) and ($\delta^u_x(m)-1$, $\delta^u_y(m)-1$, $\delta^u_z(m)-1$), the subarray `sdoms(1:2,1:3,`m`+1)` should have the followings (Figure 9).

$$\texttt{sdoms(1:2,1:3,}m\texttt{+1)} = \texttt{reshape((/}\delta^l_x(m)\texttt{,}\delta^u_x(m)\texttt{,}\delta^l_y(m)\texttt{,}\delta^u_y(m)\texttt{,}\delta^l_z(m)\texttt{,}\delta^u_z(m)\texttt{/), (/2,3/))}$$

Note that if the subdomain m is the d-th dimensional lower (upper) neighbor of n sharing a $(D-1)$-dimensional plane perpendicular to d-th axis (e.g., a neighbor along x-axis sharing a yz-plane), n's lower (upper) boundary plane has to be m's upper (lower) boundary plane. For example, if $D = 3$ and m is n's lower neighbor along x-axis, the following must be satisfied.

$$\Delta^l_x = \min_{m\in[0,N-1]}\{\delta^l_x(m)\} \qquad \Delta^u_x = \max_{m\in[0,N-1]}\{\delta^u_x(m)\}$$

$$(\delta^l_x(n) = \delta^u_x(m) \ \vee\ \delta^l_x(n) = \delta^u_x(m) - (\Delta^u_x - \Delta^l_x) \ \vee\ \delta^l_x(n) = \delta^u_x(m) + (\Delta^u_x - \Delta^l_x)) \ \wedge$$
$$\delta^l_y(n) = \delta^l_y(m) \ \wedge\ \delta^u_y(n) = \delta^u_y(m) \ \wedge\ \delta^l_z(n) = \delta^l_z(m) \ \wedge\ \delta^u_z(n) = \delta^u_z(m)$$

Alternatively, if the work to define `sdoms` is bothersome for you, you may delegate it to `oh3_init()` by making `sdoms(1,1,1)` > `sdoms(2,1,1)`, and giving the lower and upper boundaries of the whole space domain $[\Delta^l_1, \Delta^u_1-1] \times \ldots \times [\Delta^l_D, \Delta^u_D-1]$ through the argument array `scoord(2,`D`)` as follows.

$$\texttt{scoord(:,:)} = \texttt{reshape((/}\Delta^l_1\texttt{,}\Delta^u_1\texttt{,...,}\Delta^l_D\texttt{,}\Delta^u_D\texttt{/), (/2,}D\texttt{/))}$$

In this case, `oh3_init()` also refers to the argument array `pcoord(`D`)=(/`Π_1`,...,`Π_D`/)` and defines `sdoms(`β`,`d`,`$m+1$`)` for $m = rank(\pi_1, \ldots, \pi_D)$ as follows.

$$a_d = \lfloor(\Delta^u_d - \Delta^l_d)/\Pi_d\rfloor$$
$$k_d = \Pi_d - ((\Delta^u_d - \Delta^l_d) \bmod \Pi_d)$$
$$\texttt{sdoms(1,}d\texttt{,}m+1\texttt{)} = \begin{cases} \Delta^l_d + \pi_d \cdot a_d & \pi_d \le k_d \\ \Delta^l_d + \pi_d \cdot a_d + (\pi_d - k_d) & \pi_d > k_d \end{cases}$$

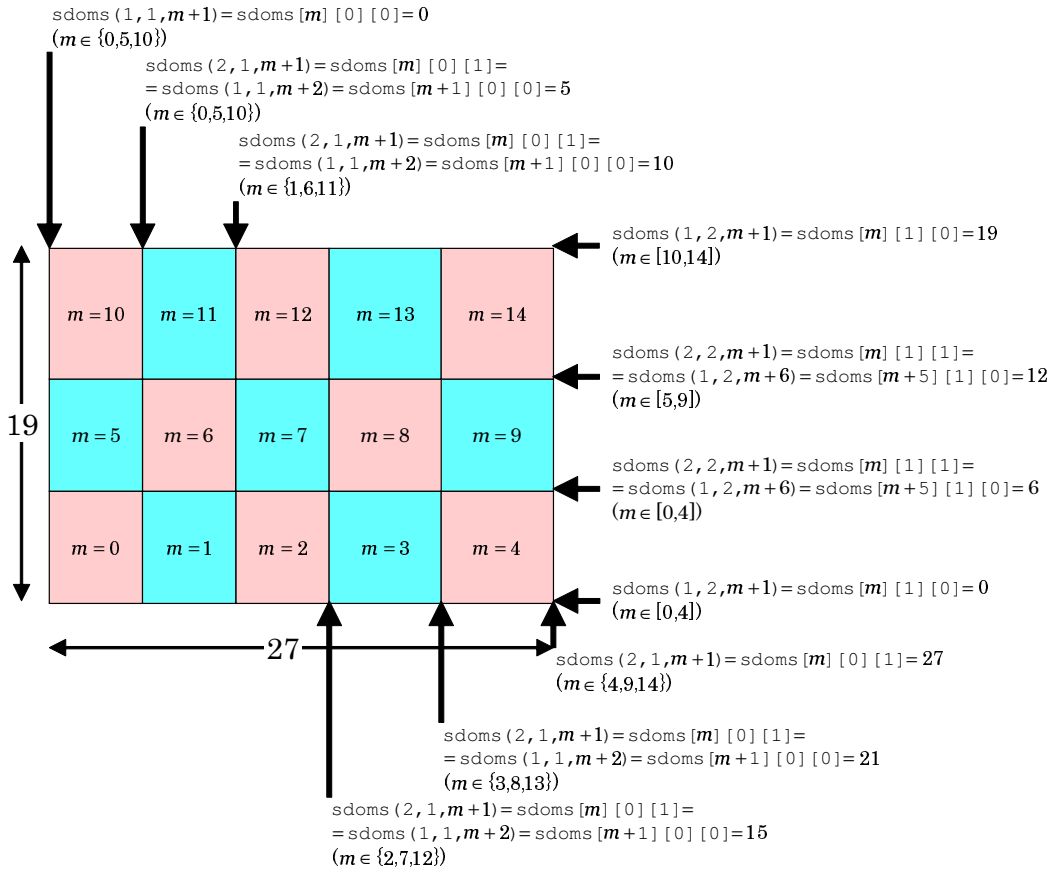


Figure 9: `sdoms` and its default setting for space domain of 27×19 given by `scoord` and node coordinate system of 5×3 given by `pcoord`.

$$m_d^+ = rank(\pi_1, \ldots, \pi_d + 1, \ldots, \pi_D)$$

$$\texttt{sdoms(2,}d\texttt{,}m+1\texttt{)} = \begin{cases} \texttt{sdoms(1,}d\texttt{,}m_d^+ + 1\texttt{)} & \pi_d < \Pi_d - 1 \\ \Delta_d^u & \pi_d = \Pi_d - 1 \end{cases}$$

That is, if we have Π_x subdomains along x-axis and the lower and upper boundaries of the whole domain along x-axis are Δ_x^l and Δ_x^u, eastmost $[(\Delta_x^u - \Delta_x^l) \bmod \Pi_x]$ subdomains have x-edges of $\lceil (\Delta_x^u - \Delta_x^l)/\Pi_x \rceil$ while remaining western ones have x-edges of $\lfloor (\Delta_x^u - \Delta_x^l)/\Pi_x \rfloor$. Note that the delegation of setting `sdoms(:,:,:)` also means that for the argument array `bounds(:,:,:)`.

`scoord(2,`D`)` should be an array whose element `scoord(`β`,`d`)` has the d-th ($d \in [1, D]$) dimensional integer coordinate of the lower ($\beta = 1$) or upper ($\beta = 2$) boundary of the whole space domain, if you delegate the setting of the array `sdoms(2,`D`,`N`)` to `oh3_init()`. That is, `scoord(1:2,1:`D`)` should have the following for the space domain of $[\Delta_1^l,\ \Delta_1^u - 1] \times \ldots \times [\Delta_D^l,\ \Delta_D^u - 1]$.

$$\texttt{scoord(:,:)} = \texttt{reshape((/}\Delta_1^l\texttt{,}\Delta_1^u\texttt{,}\ldots\texttt{,}\Delta_D^l\texttt{,}\Delta_D^u\texttt{/), (/2,}D\texttt{/))}$$

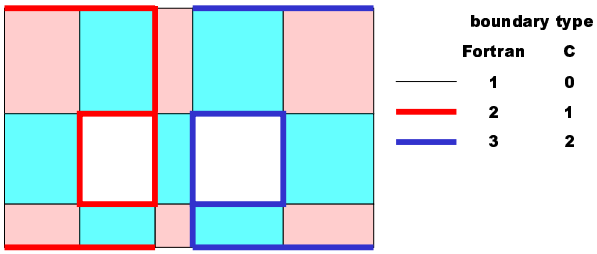


Figure 10: Complicated subdomains and their boundaries with walls and holes.

Otherwise, i.e., if you completely specify `sdoms(:,:,:)` by yourself, the array can have any values.

`nbound` should be a positive integer representing the number of boundary condition types B of the space domain. That is, you can specify a type of boundary condition $b \in [1, B]$ for each boundary of the whole space domain through the argument `bcond(2,`D`)` or of each subdomain through the argument `bounds(2,`D`,`N`)`. Then also you can specify how the communication through a boundary of a specific type is performed through the argument `ctypes(3,2,`B`,`C`)`. Remember that the boundary condition type 1 is special and reserved for periodic boundaries.

`bcond(2,`D`)` should be an array whose element `bcond(`β`,`d`)` has the type of boundary condition $b \in [1, B]$ for the lower ($\beta = 1$) or upper ($\beta = 2$) boundary plane of the whole space domain perpendicular to the d-th axis, if you delegate the setting of the array `sdoms(2,`D`,`N`)` and `bounds(2,`D`,`N`)` to `oh3_init()`. Otherwise, the array can have any values.

`bounds(2,`D`,`N`)` should be an array whose element `bounds(`β`,`D`,`$m+1$`)` has the type of boundary condition $b \in [1, B]$ for the lower ($\beta = 1$) or upper ($\beta = 2$) boundary plane of the subdomain m perpendicular to the d-th axis, if you specify `sdoms(:,:,:)` by yourself. Remember that, for a pair of adjacent subdomains, the boundary condition of the boundary plane shared by them must have type 1, unless the plane is a special *wall*. Also remember that a subdomain boundary, which is also a boundary of the whole space domain with periodic boundary condition, should have type 1 too. See Figure 10 for an example of complicated subdomain boundaries with walls and holes.

Otherwise, i.e., you delegate the setting of the array `sdoms(2,`D`,`N`)` to `oh3_init()`, it is assumed that you also delegate the setting of `bounds(:,:,:)`. In this case, `oh3_init()` gives the type 1 to *internal* boundaries, while *external* boundaries of the whole space domain will have corresponding types specified by `bcond(:,:)` as shown in Figure 11.

`ftypes(7,`F`+1)` should be an array whose elements `ftypes(1:7,`f`)` should have the followings to specify the field-array associated to grid points in a subdomain and identified by the integer $f \in [1, F]$, while `ftypes(1,`F`+1)` should be 0 (or less) to tell `oh3_init()` that you have F types of arrays.

`ftypes(1,`f`)` is the number of elements associated to a grid point of a type f field-array. For example, if f is for electromagnetic field array namely `eb(6,:,:,:,2)` whose first dimension is for three electric and three magnetic field vector components, `ftypes(1,`f`)` should be 6.

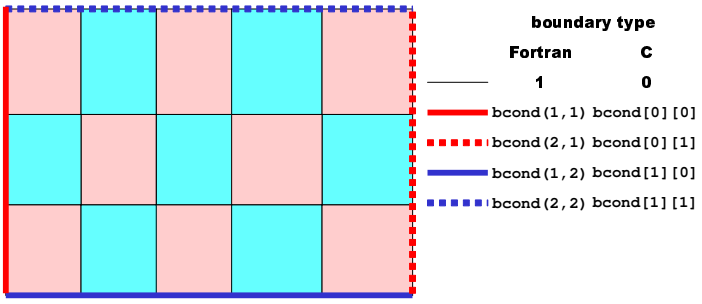


Figure 11: Default setting of subdomain boundaries.

`ftypes(2:3,`f`)` defines lower (2) and upper (3) *extensions* $e_l(f)$ and $e_u(f)$ required for the type f field-array, besides extensions for communication. That is, for a subdomain of $[0,\sigma_1-1]\times\cdots\times[0,\sigma_D-1]$, the array for f is at least as large as;

$$(e_l(f):\sigma_1+e_u(f)-1,\ \ldots\ ,e_l(f):\sigma_D+e_u(f)-1)$$

Note that if the field-arrays of type f do not need such non-communicational extensions, you should let $e_l(f)=e_u(f)=0$.

`ftypes(4:5,`f`)` defines lower (4) and upper (5) extensions $e^b_l(f)$ and $e^b_u(f)$ for the broadcast of the type f field-array. For example, for your electromagnetic filed `eb(6,:,:,:,2)` of type f for a subdomain of $[0,\sigma_x-1]\times[0,\sigma_y-1]\times[0,\sigma_z-1]$, `oh3_bcast_field()` sends elements in the range[8];

from `eb(1,`$e^b_l(f)$`,`$e^b_l(f)$`,`$e^b_l(f)$`,1)`
to `eb(6,`$\sigma_x+e^b_u(f)-1$`,`$\sigma_y+e^b_u(f)-1$`,`$\sigma_z+e^b_u(f)-1$`,1)`

to the helpers of the local node(Figure 12). Note that if the field-arrays of type f are never broadcasted, you should let $e^b_l(f)=e^b_u(f)=0$.

`ftypes(6:7,`f`)` defines lower (6) and upper (7) extensions $e^r_l(f)$ and $e^r_u(f)$ for the reduction of the type f field-array. For example, for your current density array of type f namely `cd(3,:,:,:,2)` for a subdomain of $[0,\sigma_x-1]\times[0,\sigma_y-1]\times[0,\sigma_z-1]$, `oh3_allreduce_field()` or `oh3_reduce_field()` performs the reduction of the elements in the range[9];

from `cd(1,`$e^r_l(f)$`,`$e^r_l(f)$`,`$e^r_l(f)$`,1)`
to `cd(3,`$\sigma_x+e^r_u(f)-1$`,`$\sigma_y+e^r_u(f)-1$`,`$\sigma_z+e^r_u(f)-1$`,1)`

to have the sum in the primary family of the local node. Note that if you will never perform reductions on the field-arrays of type f, you should let $e^r_l(f)=e^r_u(f)=0$.

`cfields(`C`+1)` should be an array whose element `cfields(`c`)` has $f\in[1,F]$ to identify a field-array type for which a type of boundary communication identified by the integer $c\in[1,C]$ is defined, while `ctypes(`C`+1)` should be 0 (or less) to tell `oh3_init()` that you have C types of boundary communications.

This array implies that a field-array may have two or more boundary communication types according to the timing of the communication, or no boundary communication may be taken for the field-array.

[8] Not the subarray `eb(:,`$e^b_l(f):\sigma_x+e^b_u(f)-1$`,`$e^b_l(f):\sigma_y+e^b_u(f)-1$`,`$e^b_l(f):\sigma_z+e^b_u(f)-1$`,1)`.
[9] Not the subarray `cd(:,`$e^r_l(f):\sigma_x+e^r_u(f)-1$`,`$e^r_l(f):\sigma_y+e^r_u(f)-1$`,`$e^r_l(f):\sigma_z+e^r_u(f)-1$`,1)`.

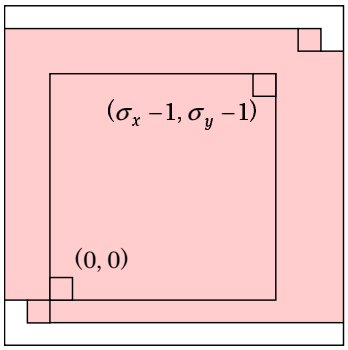


Figure 12: Type f field-array of $(\sigma_x + 5) \times (\sigma_y + 5)$ for a subdomain of $[0, \sigma_x - 1] \times [0, \sigma_y - 1]$ and its elements (painted) broadcasted by `oh3_bcast_field()` with setting of $e_l^b(f) = -1$ and $e_u^b(f) = 2$.

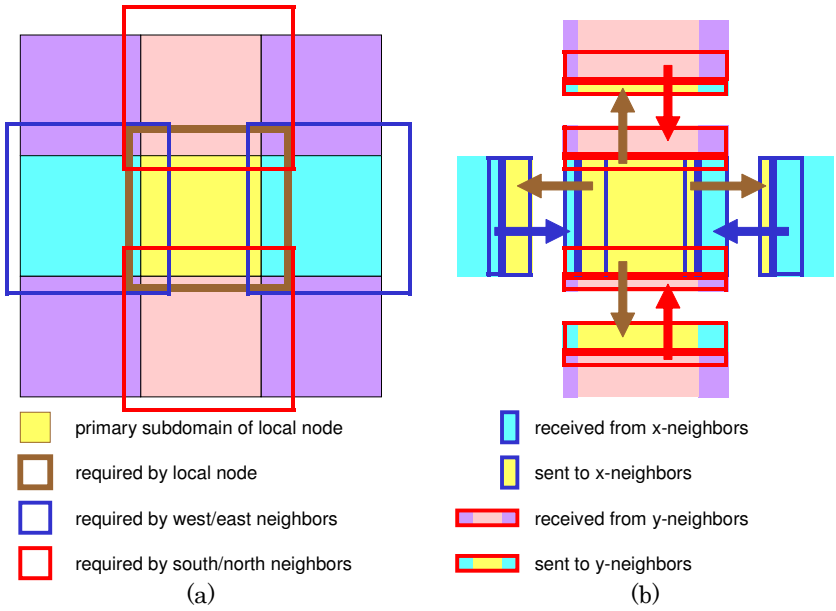


Figure 13: Field-array with downward communication $(e_f, e_t, s) = (0, 0, 2)$ and upward communication $(e_f, e_t, s) = (-1, -1, 1)$.

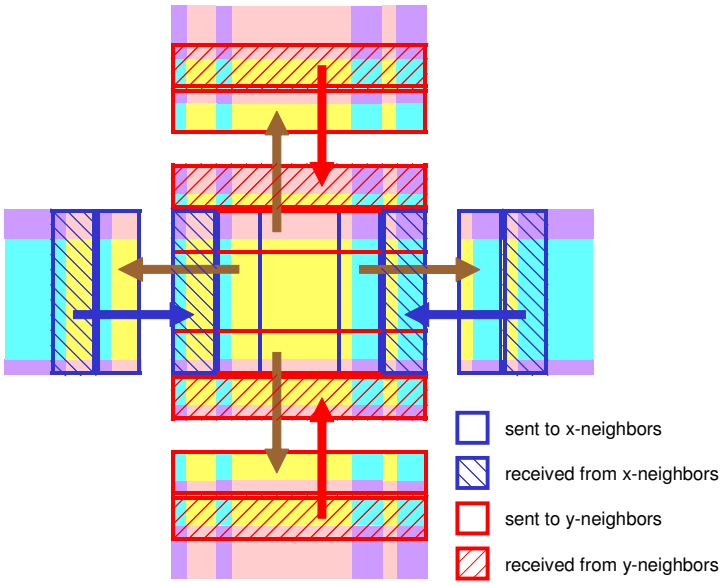


Figure 14: Field-array with downward communication $(e_f, e_t, s) = (-1, 2, 3)$ and upward communication $(e_f, e_t, s) = (-1, -4, 3)$.

`ctypes(3,2,`B`,`C`)` should be an array whose element `ctypes(1:3,`w`,`b`,`c`)=(/`e_f`,`e_t`,`s`/)` defines downward ($w = 1$) or upward ($w = 2$) boundary communication through the boundary of type $b \in [1, B]$ for a field-array $f =$ `cfields(`c`)` of the subdomain of $[0, \sigma_1 - 1] \times \ldots [0, \sigma_D - 1]$ as follows (Figure 13).

- Downward ($w = 1$) communication along d-th dimensional axis is the pair of sending s planes perpendicular to the axis to the lower neighbor and receiving the planes from the upper neighbor. The first plane to be sent has d-th dimensional coordinate e_f, while that to be received is at $\sigma_d + e_t$.
- Upward ($w = 2$) communication along d-th dimensional axis is the pair of sending s planes perpendicular to the axis to the upper neighbor and receiving the planes from the lower neighbor. The first plane to be sent has d-th dimensional coordinate $\sigma_d + e_f$, while that to be received is at e_t.

Therefore, when you just need s_l and s_u planes at the lower and upper boundaries surrouding a subdomain, $e_f = e_t = 0$ and $s = s_u$ for downward communication, while $e_f = e_t = -s_l$ and $s = s_l$ for upward communication as shown in Figure 13(b). On the other hand, if you need these planes keeping those calculated by the local node for, e.g., the addition of current densities at boundaries, $e_t = e_f + s_u$ and $s = s_u$ for downward communication, while $e_f = e_t + s_l$ for upward communication, as shown in Figure 14.

Note that if no data is transferred by downward and/or upward type c communication through a boundary of type b, the element `ctypes(3,`w`,`b`,`c`)`, i.e., s, should be set to 0.

`fsizes(2,`D`,`F`)` should be an array whose element `fsizes(`β`,`d`,`f`)` will have $\phi^l_d(f)$ ($\beta = 1$) or $\phi^u_d(f) - 1$ ($\beta = 2$) for the field-arrays of type f to notify you that the field-arrays

must have the shape (ε, $\phi_1^l(f)$:$\phi_1^u(f)$−1, ... ,$\phi_D^l(f)$:$\phi_D^u(f)$−1) for the leading D+1 dimensions, where ε = `ftypes(1,`f`)`. That is, if $D = 3$ and your field-array for electromagnetic field vectors `eb(6,:,:,:,2)` has type `feb`, you have to `allocate` the array by the following.

```
allocate(eb(6,fsizes(1,1,feb):fsizes(2,1,feb),
              fsizes(1,2,feb):fsizes(2,2,feb),
              fsizes(1,3,feb):fsizes(2,3,feb),2))
```

Note that the allocation above makes the origin of subdomains `eb(:,0,0,0,:)`. Therefore, if you like to define some other coordinates to the origin, for example `eb(:,1,2,3,:)`, you have to do the following keeping the number of elements in each dimension.

```
allocate(eb(6,fsizes(1,1,feb)+1:fsizes(2,1,feb)+1,
              fsizes(1,2,feb)+2:fsizes(2,2,feb)+2,
              fsizes(1,3,feb)+3:fsizes(2,3,feb)+3,2))
```

The value of $\phi_d^l(f)$ and $\phi_d^u(f)$ are calculated by the followings to obtain the maximum extensions at lower and upper boundaries from `ftypes(:,:)`, `cfields(:)` and `ctypes(:,:,:,:)`, and the maximum size of each subdmain edge from `sdoms(:,:,:)`.

$$
\begin{aligned}
\Gamma(f) &= \{c \,|\, \texttt{cfields(}c\texttt{)} = f\} \\
\lambda(e,s) &= \begin{cases} e & s \neq 0 \\ 0 & s = 0 \end{cases} \\
s^{\downarrow}(b,c) &= \texttt{ctypes(3,1,}b\texttt{,}c\texttt{)} \\
s^{\uparrow}(b,c) &= \texttt{ctypes(3,2,}b\texttt{,}c\texttt{)} \\
e_f^{\downarrow}(b,c) &= \lambda(\texttt{ctypes(1,1,}b\texttt{,}c\texttt{)}, s^{\downarrow}(b,c)) \\
e_t^{\downarrow}(b,c) &= \lambda(\texttt{ctypes(2,1,}b\texttt{,}c\texttt{)}, s^{\downarrow}(b,c)) \\
e_f^{\uparrow}(b,c) &= \lambda(\texttt{ctypes(1,2,}b\texttt{,}c\texttt{)}, s^{\uparrow}(b,c)) \\
e_t^{\uparrow}(b,c) &= \lambda(\texttt{ctypes(2,2,}b\texttt{,}c\texttt{)}, s^{\uparrow}(b,c)) \\
e_l^{\gamma}(f) &= \min_{b\in[1,B],c\in\Gamma(f)} (\{e_f^{\downarrow}(b,c)\} \cup \{e_t^{\uparrow}(b,c)\}) \\
e_u^{\gamma}(f) &= \max_{b\in[1,B],c\in\Gamma(f)} (\{e_t^{\downarrow}(b,c) + s^{\downarrow}(b,c)\} \cup \{e_f^{\uparrow}(b,c) + s^{\uparrow}(b,c)\}) \\
\phi_d^l(f) &= \min(e_l^{\gamma}(f), e_l(f), e_l^b(f), e_l^r(f)) \\
e_u^{\max}(f) &= \max(e_u^{\gamma}(f), e_u(f), e_u^b(f), e_u^r(f)) \\
\phi_d^{\max} &= \max_{m\in[0,N-1]} \{\delta_d^u(m) - \delta_d^l(m)\} \\
\phi_d^u(f) &= \phi_d^{\max} + e_u^{\max}(f)
\end{aligned}
$$

For example, suppose $D = 2$, the subdomain decomposition is done as shown in Figure 9 with fully periodic boundaries, and you specify the followings for your electromagnetic field array `eb(6,:,:,2)` with field-array type identifier `feb` and boundary communication type identifier `ceb`.

```
ftypes(:,feb)=(/6, 0,0, 0,1, 0,0/)
cfields(ceb)=feb
ctypes(:,:,1,ceb)=reshape((/0,0,2, -1,-1,1/), (/3,2)/)
```

Then you will have the followings in `fsizes(:,:,feb)` to allocate the array by `allocate(eb(6,-1:7,-1:8,2))`.

$$\texttt{fsizes(1,1,feb)} = \min(\min(0,-1),0,0,0) = -1$$
$$\texttt{fsizes(2,1,feb)} = 6 + \max(\max(2,0),0,1,0) = 6+2-1 = 7$$
$$\texttt{fsizes(1,2,feb)} = \min(\min(0,-1),0,0,0) = -1$$
$$\texttt{fsizes(2,2,feb)} = 7 + \max(\max(2,0),0,1,0) = 7+2-1 = 8$$

`stats`
`repiter`
`verbose`
See §3.4.1 because the arguments above are perfectly equivalent to those of `oh1_init()`.

C Interface

```
void oh3_init(int **sdid, int nspec, int maxfrac, int **nphgram,
              int **totalp, struct S_particle **pbuf, int **pbase,
              int maxlocalp, void *mycomm, int **nbor, int *pcoord,
              int **sdoms, int *scoord, int nbound, int *bcond, int **bounds,
              int *ftypes, int *cfields, int *ctypes, int **fsizes,
              int stats, int repiter, int verbose);
```

`sdid`
`nspec`
`maxfrac`
`nphgram`
See §3.4.1 because the arguments above are perfectly equivalent to those of `oh1_init()`.

`totalp`
`pbuf`
`pbase`
`maxlocalp`
See §3.5.2 because the arguments above are perfectly equivalent to those of `oh2_init()`.

`mycomm`
`nbor`
`pcoord`
See §3.4.1 because the arguments above are perfectly equivalent to those of `oh1_init()`.

`**sdoms` should be a double pointer to an array of $N \times D \times 2$ elements to form `sdoms`$[N][D][2]$ conceptually, or a pointer to `NULL` (not `NULL` itself) if you want the library to allocate and initialize the array and return the pointer to it through the argument. If you prepare the array, its element `sdoms`$[m][d][\beta]$ should have the d-th ($d \in [0, D-1]$) dimensional integer coordinate of the lower ($\beta = 0$) or upper ($\beta = 1$) boundary of the subdomain $m \in [0, N-1]$, namely $\delta_d^l(m)$ or $\delta_d^u(m)$ respectively. For example, for the 3-dimensional cuboid subdomain m whose grid

points at west-south-east and east-north-top corners are $(\delta^l_x(m), \delta^l_y(m), \delta^l_z(m))$ and $(\delta^u_x(m)-1, \delta^u_y(m)-1, \delta^u_z(m)-1)$, the subarray `sdoms`$[m][\,][\,]$ should have the followings (Figure 9).

`sdoms`$[m]$`[0][0]`$=\delta^l_x(m)$; `sdoms`$[m]$`[0][1]`$=\delta^u_x(m)$;
`sdoms`$[m]$`[1][0]`$=\delta^l_y(m)$; `sdoms`$[m]$`[1][1]`$=\delta^u_y(m)$;
`sdoms`$[m]$`[2][0]`$=\delta^l_z(m)$; `sdoms`$[m]$`[2][1]`$=\delta^u_z(m)$;

Note that if the subdomain m is the d-th dimensional lower (upper) neighbor of n sharing a $(D-1)$-dimensional plane perpendicular to d-th axis (e.g., a neighbor along x-axis sharing a yz-plane), n's lower (upper) boundary plane has to be m's upper (lower) boundary plane. For example, if $D = 3$ and m is n's lower neighbor along x-axis, the following must be satisfied.

$$\Delta^l_x = \min_{m\in[0,N-1]}\{\delta^l_x(m)\} \qquad \Delta^u_x = \max_{m\in[0,N-1]}\{\delta^u_x(m)\}$$
$$(\delta^l_x(n) = \delta^u_x(m) \ \vee\ \delta^l_x(n) = \delta^u_x(m) - (\Delta^u_x - \Delta^l_x) \ \vee\ \delta^l_x(n) = \delta^u_x(m) + (\Delta^u_x - \Delta^l_x)) \ \wedge$$
$$\delta^l_y(n) = \delta^l_y(m) \ \wedge\ \delta^u_y(n) = \delta^u_y(m) \ \wedge\ \delta^l_z(n) = \delta^l_z(m) \ \wedge\ \delta^u_z(n) = \delta^u_z(m)$$

Alternatively, if the work to define `sdoms` is bothersome for you, you may delegate it to `oh3_init()` by passing a pointer to `NULL` or by making `sdoms`$[0][0][0] >$ `sdoms`$[0][0][1]$ and gives the lower and upper boundaries of the whole space domain $[\Delta^l_0, \Delta^u_0-1] \times \ldots \times [\Delta^l_{D-1}, \Delta^u_{D-1}-1]$ through the argument array `scoord`$[D][2]$ as follows.

`int scoord[`D`][2] = {{`Δ^l_0`,`Δ^u_0`},...,{`Δ^l_{D-1}`,`Δ^u_{D-1}`}};`

In this case, `oh3_init()` also refers to the argument array `pcoord`$[D]$ = `{`$\Pi_0, \ldots, \Pi_{D-1}$`}` and defines `sdoms`$[m][d][\beta]$ for $m = rank(\pi_0, \ldots, \pi_{D-1})$ as follows.

$$\begin{aligned}
a_d &= \lfloor(\Delta^u_d - \Delta^l_d)/\Pi_d\rfloor \\
k_d &= \Pi_d - ((\Delta^u_d - \Delta^l_d) \bmod \Pi_d) \\
\texttt{sdoms}[m][d][0] &= \begin{cases} \Delta^l_d + \pi_d \cdot a_d & \pi_d \le k_d \\ \Delta^l_d + \pi_d \cdot a_d + (\pi_d - k_d) & \pi_d > k_d \end{cases} \\
m^+_d &= rank(\pi_0, \ldots, \pi_d + 1, \ldots, \pi_{D-1}) \\
\texttt{sdoms}[m][d][1] &= \begin{cases} \texttt{sdoms}[m^+_d][d][0] & \pi_d < \Pi_d - 1 \\ \Delta^u_d & \pi_d = \Pi_d - 1 \end{cases}
\end{aligned}$$

That is, if we have Π_x subdomains along x-axis and the lower and upper boundaries of the whole domain along x-axis are Δ^l_x and Δ^u_x, eastmost $[(\Delta^u_x - \Delta^l_x) \bmod \Pi_x]$ subdomains have x-edges of $\lceil(\Delta^u_x - \Delta^l_x)/\Pi_x\rceil$ while remaining western ones have x-edges of $\lfloor(\Delta^u_x - \Delta^l_x)/\Pi_x\rfloor$. Note that the delegation of setting `sdoms` also means that for the argument array `bounds`.

`*scoord` should be a pointer to an array of $D \times 2$ to form `scoord`$[D][2]$ conceptually, if you delegate the setting of the array `sdoms`$[N][D][2]$ to `oh3_init()`. If so, its element `scoord`$[d][\beta]$ should have the d-th ($d \in [0, D-1]$) dimensional integer coordinate of the lower ($\beta = 0$) or upper ($\beta = 1$) boundary of the whole space domain. That is, `scoord`$[D][2]$ should have the following for the space domain of $[\Delta^l_0, \Delta^u_0-1] \times \ldots \times [\Delta^l_{D-1}, \Delta^u_{D-1}-1]$.

`int scoord[`D`][2] = {{`Δ^l_0`,`Δ^u_0`},...,{`Δ^l_{D-1}`,`Δ^u_{D-1}`}};`

Otherwise, i.e., if you completely specify `sdoms` by yourself, `scoord` can be `NULL` or the array can have any values.

`nbound` should be a positive integer representing the number of boundary condition types B of the space domain. That is, you can specify a type of boundary condition $b \in [0, B-1]$ for each boundary of the whole space domain through the argument `bcond`$[D][2]$ or of each subdomain through the argument `bounds`$[N][D][2]$. Then also you can specify how the communication through a boundary of a specific type is performed through the argument `ctypes`$[C][B][2][3]$. Remember that the boundary condition type 0 is special and reserved for periodic boundaries.

`*bcond` should be a pointer to an array of $D \times 2$ to form `bcond`$[D][2]$ conecptually, if you delegate the setting of the array `sdoms`$[N][D][2]$ and `bounds`$[N][D][2]$ to `oh3_init()`. If so, its element `bcond`$[d][\beta]$ should have the type of boundary condition $b \in [0, B-1]$ for the lower ($\beta = 0$) or upper ($\beta = 1$) boundary plane of the whole space domain perpendicular to the d-th axis. Otherwise, `bcond` can be `NULL` or the array can have any values.

`**bounds` should be a double pointer to an array of $N \times D \times 2$ to form `bounds`$[N][D][2]$ conceptually, if you specify `sdoms` by yourself. If so, its element `bounds`$[m][d][\beta]$ should have the type of boundary condition $b \in [0, B-1]$ for the lower ($\beta = 0$) or upper ($\beta = 1$) boundary plane of the subdomain m perpendicular to the d-th axis. Remember that, for a pair of adjacent subdomains, the boundary condition of the boundary plane shared by them must have type 0, unless the plane is a special *wall*. Also remember that a subdomain boundary, which is also a boundary of the whole space domain with periodic boundary condition, should have type 0 too. See Figure 10 an example of complicated subdomain boundaries with walls and holes.

Otherwise, i.e., you delegate the setting of the array `sdoms[][][]` to `oh3_init()`, it is assumed that you also delegate the setting of `bounds`. In this case, `oh3_init()` allocate the array of $N \times D \times 2$ and set the pointer to it to `*bounds` if it was `NULL`, and then initialize `bounds` so that *internal* boundaries have the type 0, while *external* boundaries of the whole space domain have corresponding types specified by `bcond` as shown in Figure 11.

`*ftypes` should be a pointer to an array of $(F+1) \times 7$ to form `ftypes`$[F+1][7]$ conceptually. Its element `ftypes`$[f][\,]$ should have the followings to specify the field-array associated to grid points in a subdomain and identified by the integer $f \in [0, F-1]$, while `ftypes`$[F][0]$ should be 0 (or less) to tell `oh3_init()` that you have F types of arrays.

`ftypes[`f`][0]` is the number of elements associated to a grid point of a type f field-array. For example, if f is for electromagnetic field array namely `eb[2][][][]` of six `double` elements `struct` for three electric and three magnetic field vector components, `ftypes[`f`][0]` should be 6.

`ftypes[`f`][1]` and `ftypes[`f`][2]` defines lower (1) and upper (2) *extensions* $e_l(f)$ and $e_u(f)$ required for the type f field-arrays, besides extensions for communication. That is, for a subdomain of $[0, \sigma_0-1] \times \cdots \times [0, \sigma_{D-1}-1]$, the array for f is at least as large as to have grid points of $[e_l(f), \sigma_0+e_u(f)-1] \times \cdots \times [e_l(f), \sigma_{D-1}+e_u(f)-1]$. Note that if the field-arrays of type f do not need such non-communicational extensions, you should let $e_l(f) = e_u(f) = 0$.

`ftypes[`f`][3]` and `ftypes[`f`][4]` defines lower (3) and upper (4) extensions $e^b_l(f)$ and $e^b_u(f)$ for the broadcast of the type f field-arrays. For example, for your electromagnetic filed `eb[2][][][]` of type f for a subdomain of $[0,\sigma_x-1]\times[0,\sigma_y-1]\times[0,\sigma_z-1]$, `oh3_bcast_field()` sends structured elements in the range[10];

from `eb[0]`$[e^b_l(f)]$`[`$e^b_l(f)$`][`$e^b_l(f)$`]`
to `eb[0]`$[\sigma_z+e^b_u(f)-1]$`[`$\sigma_y+e^b_u(f)-1$`][`$\sigma_x+e^b_u(f)-1$`]`

to the helpers of the local node (Figure 12). Note that if the field-arrays of type f are never broadcasted, you should let $e^b_l(f)=e^b_u(f)=0$.

`ftypes[`f`][5]` and `ftypes[`f`][6]` defines lower (5) and upper (6) extensions $e^r_l(f)$ and $e^r_u(f)$ for the reduction of the type f field-array. For example, for your current density array of type f namely `cd[2][][][]` having structured elements of three vector components for a subdomain of $[0,\sigma_x-1]\times[0,\sigma_y-1]\times[0,\sigma_z-1]$, `oh3_allreduce_field()` or `oh3_reduce_field()` performs the reduction of the elements in the range[11];

from `cd[0]`$[e^r_l(f)]$`[`$e^r_l(f)$`][`$e^r_l(f)$`]`
to `cd[0]`$[\sigma_z+e^r_u(f)-1]$`[`$\sigma_y+e^r_u(f)-1$`][`$\sigma_x+e^r_u(f)-1$`]`

to have the sum in the primary family of the local node. Note that if you will never perform reductions on the field-arrays of type f, you should let $e^r_l(f)=e^r_u(f)=0$.

`*cfields` should be a pointer to an array of $C+1$ elements and its element `cfields`$[c]$ should have $f\in[0,F-1]$ to identify a field-array type for which a type of boundary communication identified by the integer $c\in[0,C-1]$ is defined, while `ctypes`$[C]$ should be -1 (or less) to tell `oh3_init()` that you have C types of boundary communications.

This array implies that a field-array may have two or more boundary communication types according to the timing of the communication, or no boundary communication may be taken for the field-array.

`*ctypes` should be a pointer to an array of $C\times B\times 2\times 3$ to form `ctypes`$[C][B][2][3]$ conceptually. Its elements `ctypes`$[c][b][w][\,]=(e_f,e_t,s)$ defines downward ($w=0$) or upward ($w=1$) boundary communication through the boundary of type $b\in[0,B-1]$ for a field-array $f=$ `cfields`$[c]$ of the subdomain of $[0,\sigma_0-1]\times\ldots[0,\sigma_{D-1}-1]$ as follows (Figure 13).

- Downward ($w=0$) communication along d-th dimensional axis is the pair of sending s planes perpendicular to the axis to the lower neighbor and receiving the planes from the upper neighbor. The first plane to be sent has d-th dimensional coordinate e_f, while that to be received is at σ_d+e_t.
- Upward ($w=1$) communication along d-th dimensional axis is the pair of sending s planes perpendicular to the axis to the upper neighbor and receiving the planes from the lower neighbor. The first plane to be sent has d-th dimensional coordinate σ_d+e_f, while that to be received is at e_t.

[10] Not the set of structured elements
$\{$`eb[0][z][y][x]` $\mid x\in[e^b_l(f),\sigma_x+e^b_u(f)-1], y\in[e^b_l(f),\sigma_y+e^b_u(f)-1], z\in[e^b_l(f),\sigma_z+e^b_u(f)-1]\}$

[11] Not the set of structured elements
$\{$`cd[0][z][y][x]` $\mid x\in[e^r_l(f),\sigma_x+e^r_u(f)-1], y\in[e^r_l(f),\sigma_y+e^r_u(f)-1], z\in[e^r_l(f),\sigma_z+e^r_u(f)-1]\}$

Therefore, when you just need s_l and s_u planes at the lower and upper boundaries surrouding a subdomain, $e_f = e_t = 0$ and $s = s_u$ for downward communication, while $e_f = e_t = -s_l$ and $s = s_l$ for upward communication as shown in Figure 13(b). On the other hand, if you need these planes keeping those calculated by the local node for, e.g., the addition of current densities at boundaries, $e_t = e_f + s_u$ and $s = s_u$ for downward communication, while $e_f = e_t + s_l$ for upward communication, as shown in Figure 14.

Note that if no data is transferred by downward and/or upward type c communication through a boundary of type b, the element `ctypes`$[c][b][w][2]$, i.e., s, should be set to 0.

`**fsizes` should be a double pointer to an array of $F \times D \times 2$ to form `fsizes`$[F][D][2]$ conceptually, or a pointer to `NULL` (not `NULL` itself) if you want the library to allocate the array and return the pointer to it through the argument. In both cases, its element `fsizes`$[f][d][\beta]$ will have $\phi^l_d(f)$ ($\beta = 0$) or $\phi^u_d(f)$ ($\beta = 1$) for the field-arrays of type f to notify you that the field-arrays must have the size of $(\phi^u_{D-1}(f) - \phi^l_{D-1}(f)) \times \cdots \times (\phi^u_0(f) - \phi^u_D(f)) \times \varepsilon$ for each of primary and secondary subdomains, where $\varepsilon =$ `ftypes`$[f][0]$. That is, if $D = 3$ and your field-array for electromagnetic field vectors `eb[2][][][]` of `struct` named `ebfield` has type `feb`, you have to allocate the array by the following.

```
int (*fs)[3][2]=(int(*)[3][2])(*fsizes);
int s[3]={fs[feb][0][1]-fs[feb][0][0],
          fs[feb][1][1]-fs[feb][1][0],
          fs[feb][2][1]-fs[feb][2][0]};
int lext=fs[feb][0][0]+s[0]*(fs[feb][1][0]+s[1]*fs[feb][2][0]);
eb[0] = (struct ebfield*)
        malloc(sizeof(struct ebfield)*s[0]*s[1]*s[2]*2) - lext;
eb[1] = eb[0] + s[0]*s[1]*s[2];
```

Note that the allocation above makes `eb[0]` and `eb[1]` points the origin of the subdomain at $(0, 0, 0)$ in its local integer coordinate system. Therefore, if you like to make `eb[]` point some other grid point, for example $(1, 2, 3)$, you have to modify `lext` above as follows.

```
int lext=(fs[feb][0][0]-1)+
         s[0]*((fs[feb][1][0]-2)+s[1]*(fs[feb][2][0]-3));
```

The value of $\phi^l_d(f)$ and $\phi^u_d(f)$ are calculated by the followings to obtain the maximum extensions at lower and upper boundaries from `ftypes[][]`, `cfields[]` and `ctypes[][][][]`, and the maximum size of each subdmain edge from `sdoms[][][]`.

$$
\begin{aligned}
\Gamma(f) &= \{c \,|\, \texttt{cfields}[c] = f\} \\
\lambda(e, s) &= \begin{cases} e & s \neq 0 \\ 0 & s = 0 \end{cases} \\
s^{\downarrow}(b, c) &= \texttt{ctypes}[c][b][0][2] \\
s^{\uparrow}(b, c) &= \texttt{ctypes}[c][b][1][2] \\
e^{\downarrow}_f(b, c) &= \lambda(\texttt{ctypes}[c][b][0][0], s^{\downarrow}(b, c)) \\
e^{\downarrow}_t(b, c) &= \lambda(\texttt{ctypes}[c][b][0][1], s^{\downarrow}(b, c)) \\
e^{\uparrow}_f(b, c) &= \lambda(\texttt{ctypes}[c][b][1][0], s^{\uparrow}(b, c))
\end{aligned}
$$

$$
\begin{aligned}
e_t^{\uparrow}(b,c) &= \lambda(\texttt{ctypes}[c][b][1][1], s^{\uparrow}(b,c)) \\
e_l^{\gamma}(f) &= \min_{b\in[0,B-1],c\in\Gamma(f)}(\{e_f^{\downarrow}(b,c)\}\cup\{e_t^{\uparrow}(b,c)\}) \\
e_u^{\gamma}(f) &= \max_{b\in[0,B-1],c\in\Gamma(f)}(\{e_t^{\downarrow}(b,c)+s^{\downarrow}(b,c)\}\cup\{e_f^{\uparrow}(b,c)+s^{\uparrow}(b,c)\}) \\
\phi_d^l(f) &= \min(e_l^{\gamma}(f), e_l(f), e_l^b(f), e_l^r(f)) \\
e_u^{\max}(f) &= \max(e_u^{\gamma}(f), e_u(f), e_u^b(f), e_u^r(f)) \\
\phi_d^{\max} &= \max_{m\in[0,N-1]}\{\delta_d^u(m)-\delta_d^l(m)\} \\
\phi_d^u(f) &= \phi_d^{\max}+e_u^{\max}(f)
\end{aligned}
$$

For example, suppose $D = 2$, the subdomain decomposition is done as shown in Figure 9 with fully periodic boundaries, and you specify the followings for your electromagnetic field array `eb[2][][]` with field-array type identifier `feb` and boundary communication type identifier `ceb`.

```
ftypes[feb][0]=6;
ftypes[feb][1]=0; ftypes[feb][2]=0;
ftypes[feb][3]=0; ftypes[feb][4]=1;
ftypes[feb][5]=0; ftypes[feb][6]=0;
cfields[ceb]=feb;
ctypes[ceb][0][0][0]=ctypes[ceb][0][0][1]=0;
ctypes[ceb][0][0][2]=2;
ctypes[ceb][0][1][0]=ctypes[ceb][0][1][1]=-1;
ctypes[ceb][0][1][2]=1;
```

Then you will have the followings in `fsizes[feb][][]` to allocate the array of six-element structures of $(1+8)\times(1+9)\times 2$.

$$
\begin{aligned}
\texttt{fsizes[feb][0][0]} &= \min(\min(0,-1),0,0,0) = -1 \\
\texttt{fsizes[feb][0][1]} &= 6+\max(\max(2,0),0,1,0) = 6+2 = 8 \\
\texttt{fsizes[feb][1][0]} &= \min(\min(0,-1),0,0,0) = -1 \\
\texttt{fsizes[feb][1][1]} &= 7+\max(\max(2,0),0,1,0) = 7+2 = 9
\end{aligned}
$$

`stats`
`repiter`
`verbose`
: See §3.4.1 because the arguments above are perfectly equivalent to those of `oh1_init()`.

3.6.2 `oh13_init()`

The function (subroutine) `oh13_init()` performs what `oh3_init()` does excluding the initialization of `oh2_init()` but including that of `oh1_init()`. More specifically, let I_1, I_2 and I_3 be the set of initializing operations performed in `oh1_init()`, `oh2_init()` and `oh3_init()` respectively, and thus $I_1 \subset I_2 \subset I_3$. The funtion `oh13_init()` performs $I_3 - (I_2 - I_1)$ for those who want to have functions provided by level-3 library but to transfer and manage particles by themselves. Therefore `oh13_init()` does not allocate the large buffer for particle transfer. It also inhibits perticle transfer operations in `oh3_`

`transbound()` to make it almost equivalent to `oh1_transbound()` besides a few necessary operations for field-arrays.

The definition $I_3 - (I_2 - I_1)$ of the initialization by `oh13_init()` is similarly applicable to its arguments. That is, its set of arguments is $A_3 - (A_2 - A_1) \cup A_1$ where A_k is the set of arguments of `oh`k`_init()`. Note that two arguments `rcounts` and `scounts` of `oh1_init()`, which is excluded from `oh2_init()` and thus also from `oh3_init()`, is in the set of `oh13_init()`.

Fortran Interface

```
      subroutine oh13_init(sdid, nspec, maxfrac, nphgram, totalp, &
                           rcounts, scounts, mycomm, nbor, pcoord, &
                           sdoms, scoord, nbound, bcond, bounds, ftypes, &
                           cfields, ctypes, fsizes, &
                           stats, repiter, verbose)
        use oh_type
        implicit none
        integer,intent(out)   :: sdid(2)
        integer,intent(in)    :: nspec
        integer,intent(in)    :: maxfrac
        integer,intent(inout) :: nphgram(:,:,:)
        integer,intent(out)   :: totalp(:,:)
        integer,intent(out)   :: rcounts(:,:,:)
        integer,intent(out)   :: scounts(:,:,:)
        type(oh_mycomm),intent(out) :: mycomm
        integer,intent(inout) :: nbor(3,3,3)        ! for 3D codes.
        integer,intent(in)    :: pcoord(OH_DIMENSION)
        integer,intent(inout) :: sdoms(:,:,:)
        integer,intent(in)    :: scoord(2,OH_DIMENSION)
        integer,intent(in)    :: nbound
        integer,intent(in)    :: bcond(2,OH_DIMENSION)
        integer,intent(inout) :: bounds(:,:,:)
        integer,intent(in)    :: ftypes(:,:)
        integer,intent(in)    :: cfields(:)
        integer,intent(in)    :: ctypes(:,:,:,:)
        integer,intent(out)   :: fsizes(:,:,:)
        integer,intent(in)    :: stats
        integer,intent(in)    :: repiter
        integer,intent(in)    :: verbose
      end subroutine
```

C Interface

```
 void oh13_init(int **sdid, int nspec, int maxfrac, int **nphgram,
                int **totalp, int **rcounts, int **scounts,
                void *mycomm, int **nbor, int *pcoord,
                int **sdoms, int *scoord, int nbound, int *bcond,
                int **bounds, int *ftypes, int *cfields, int *ctypes,
                int **fsizes,
                int stats, int repiter, int verbose);
```

`sdid`
`nspec`

`maxfrac`
`nphgram`
`totalp`
`rcounts`
`scounts`
`mycomm`
`nbor`
`pcoord`
: See §3.4.1 because the arguments above are perfectly equivalent to those of `oh1_init()`.

`sdoms`
`scoord`
`nbound`
`bcond`
`bounds`
`ftypes`
`cfields`
`ctypes`
`fsizes`
: See §3.6.1 because the arguments above are perfectly equivalent to those of `oh3_init()`.

`stats`
`repiter`
`verbose`
: See §3.4.1 because the arguments above are perfectly equivalent to those of `oh1_init()`.

3.6.3 `oh3_grid_size()`

The function (subroutine) `oh3_grid_size()` is to specify the *grid size* of each dimension if the *real* coordinate for particle locations is different from the *integer* coordinate for subdomains and field-arrays of them. Specifically, the d-th element ($d \in [1, D]$ for Fortran and $d \in [0, D-1]$ for C) of its sole argument `size` being 1-dimensional array of D elemnets should have the scale factor γ_d to map integer coordinate $(x^i_1, \cdots, x^i_D)$ to $(x^i_1 \cdot \gamma_1, \cdots, x^i_D \cdot \gamma_D)$.

Fortran Interface

```
subroutine oh3_grid_size(size)
  implicit none
  real*8,intent(in)     :: size(OH_DIMENSION)
end subroutine
```

C Interface

```
void oh3_grid_size(double size[OH_DIMENSION]);
```

The grid size γ_d will only affect the result of `oh3_map_particle_to_neighbor()` or `oh3_map_particle_to_subdomain()` whose return value will be m iff $x_d \in [\delta^l_d(m) \cdot \gamma_d,\ \delta^u_d(m) \cdot \gamma_d)$ for all $d \in [1, D]$, where x_d is the argument `x`, `y` or `z` of the functions. Note that this function should be called just once, if necessary, *after* `oh3_init()` (or `oh13_init()`) is called

and *before* the first call of `oh3_map_particle_to_neighbor()` or `oh3_map_particle_to_subdomain()`.

3.6.4 `oh3_transbound()`

If you initialize the library by `oh3_init()`, the function `oh3_transbound()` at first performs the same operetaions as `oh2_transbound()` does; that is, examination of the balancing and (re)building of helpand-helper configuration if necessary, followed by particle transfer. Otherwise, i.e., if you have called `oh13_init()`, `oh3_transbound()` acts as `oh1_transbound()` to make particle transfer schedule. Finally, in both cases, `oh3_transbound()` maintains library's internal data structures for field-arrays of the secondary subdomain, if helpand-helper configuration has been (re)built. For this maintenance, the function refers to the information given to `oh3_init()` but not the argument arrays themselves.

Since the arguments of `oh3_transbound()` and its return value are perfectly equivalent to those of `oh1_transbound()` (and `oh2_transbound()`), see §3.4.4 for their definitions.

Fortran Interface

```
integer function oh3_transbound(currmode, stats)
  implicit none
  integer,intent(in) :: currmode
  integer,intent(in) :: stats
end function
```

C Interface

```
int oh3_transbound(int currmode, int stats);
```

3.6.5 `oh3_map_particle_to_neighbor()`

The function `oh3_map_particle_to_neighbor()` returns the identifier of the subdomain in which the particle at given position will reside and to which the primary or secondary subdomain of the local node adjoins. Therefore, if the particle may be in a non-neighboring subdomain due to, for example, initial particle distribution, particle injection or particle warp, the relative function `oh3_map_particle_to_subdomain()` should be used.

Although the function is faster than `oh3_map_particle_to_subdomain()`, it is not good idea to use it to examine whether the particle is in the primary/secondary subdomain of the local node, because the calling cost is not negligible. That is, it is strongly recommended to do the examination by yourself and then call this function if you find the particle has gone.

This function has three instances with two, three and four arguments according to the dimension of the simulated space domain defined by $D =$ `OH_DIMENSION`.

Fortran Interface

```
integer function oh3_map_particle_to_neighbor(x, ps)
  implicit none
  real*8,intent(inout) :: x
  integer,intent(in)   :: ps
end function
```

```
integer function oh3_map_particle_to_neighbor(x, y, ps)
  implicit none
  real*8,intent(inout) :: x
  real*8,intent(inout) :: y
  integer,intent(in)   :: ps
end function
integer function oh3_map_particle_to_neighbor(x, y, z, ps)
  implicit none
  real*8,intent(inout) :: x
  real*8,intent(inout) :: y
  real*8,intent(inout) :: z
  integer,intent(in)   :: ps
end function
```

C Interface

```
int  oh3_map_particle_to_neighbor(double *x, int ps);
int  oh3_map_particle_to_neighbor(double *x, double *y, int ps);
int  oh3_map_particle_to_neighbor(double *x, double *y, double *z, int ps);
```

`x`,`y`,`z` (for Fortran)
`*x`,`*y`,`*z` (for C)
: These three (if $D = 3$) arguments should be the coordinates at which a particle is located in Fortran, or the pointers to the variables having the coordinates in C. In both cases, the actual argument variables may be updated as discussed later.

`ps` should be 0 for a primary particle, or 1 for a secondary particle.

return value is the identifier of the subdomain in which the particle will reside, or -1 if such a subdomain is not found as discussed later.

The function at first examines whether the particle is in the primary (`ps` = 0) or secondary (`ps` = 1) subdomain of the local node and returns its identifier if the particle is in it, referring to the subdomain boundaries given by or set to the argument `sdoms` of `oh3_init()`. Otherwise, it assumes that the particle has moved into a subdomain adjoining to the primary/secondary subdomain and returns the identifier of the subdomain into which the particle has moved, referring to the neighboring infomation given by or set to the argument `nbor` of `oh3_init()`, or that in the helpand.

In the latter case of the boundary crossing, the periodic boundary condition of the whole space domain is taken care of by the function. Therefore, the coordinates given by `x`, `y` and `z` should be *raw* ones without wraparound. Moreover, the actual argument variables are updated by the function if the particle has crossed a periodic boundary. For example, if the particle has crossed the periodic boundary plane perpendicular to x-axis, the actual argument variable x is updated as follows.

$$x \leftarrow \begin{cases} x + (\Delta_x^u - \Delta_x^l) & x < \Delta_x^l \\ x - (\Delta_x^u - \Delta_x^l) & x \geq \Delta_x^u \end{cases}$$

On the other hand, if the particle has crossed a non-periodic boundary of the whole space domain, the function returns -1 to indicate that the particle is out of bounds[12]. To

[12]The values in the actual argument variables are kept unless the particle has crossed two or more contacting space domain boundaries including periodic ones at once. More specifically, the function examines boundary crossing in the order of yz, xz and then xy planes if $D = 3$, and updates actual argument variables `x`, `y` and `z` in this order if the corresponding boundary planes are periodic.

examine the boundary condition, the function refers to the conditions given through the argument `bcond` or `bounds` of `oh3_init()`. The function also returns -1 if the particle has moved into a non-exsistent neighbor, which may be defined by `nbor`.

3.6.6 `oh3_map_particle_to_subdomain()`

The function `oh3_map_particle_to_subdomain()` returns the identifier of the subdomain in which the particle at given position will reside. Unlike the relative function `oh3_map_particle_to_neighbor()`, this function can find the identifier of *any* subdomain and thus should be used for, e.g., initial particle disribution, particle injection, particle warp, and so on. Of course you may use this function always but have to remember that it is slower than `oh3_map_particle_to_neighbor()` especially if you specify `sdoms` argument of `oh3_init()` by yourself.

This function has three instances with one, two and three arguments according to the dimension of the simulated space domain defined by $D =$ `OH_DIMENSION`.

Fortran Interface

```
integer function oh3_map_particle_to_subdomain(x)
  implicit none
  real*8,intent(in)  :: x
end function
integer function oh3_map_particle_to_subdomain(x, y)
  implicit none
  real*8,intent(in)  :: x
  real*8,intent(in)  :: y
end function
integer function oh3_map_particle_to_subdomain(x, y, z)
  implicit none
  real*8,intent(in)  :: x
  real*8,intent(in)  :: y
  real*8,intent(in)  :: z
end function
```

C Interface

```
int  oh3_map_particle_to_subdomain(double x);
int  oh3_map_particle_to_subdomain(double x, double y;
int  oh3_map_particle_to_subdomain(double x, double y, double z);
```

`x`,`y` and `z` should be the coordinates at which a particle is located.

return value is the identifier of the subdomain in which the particle will reside, or -1 if such a subdomain is not found as discussed later.

If you delegated the setting of `sdoms` array of `oh3_init()`, the function finds the subdomain by a simple calculation taking $O(1)$ time which should be, however, longer than that taken by `oh3_map_particle_to_neighbor()` due to an integer division. Therefore, it is not good idea to call this function to examine whether the particle is in the primary/secondary subdomain of the local node. That is, you should examine it by yourself and then, if the particle has gone outside, call this function. Also note that the calculation does not take care of the periodic boundary condition of the whole space domain, and thus you

have to perform wraparound calculation *before* calling this function if necessary, or you will get the return value -1 to indicate that particle is out of bounds.

On the other hand, if you specify the array `sdoms` by yourself, this function *searches* the target subdomain. If your space domain is a cuboid (or a rectangler or a line segment) without any holes, the cost of search is $O(\log N)$. Otherwise, for a complicatedly shaped domain, the cost could be $O(N)$ although the function does its best to reduce it to $O(\log N)$. The search may fail if there is no subdomain including the given particle coordinates due to, for example, going outside the whole space domain, dropping into a hole, and so on, to make the function return -1.

3.6.7 `oh3_bcast_field()`

The function (subroutine) `oh3_bcast_field()` performs red-black broadcast communications of a field-array whose type is specified by its argument `ftype` in the families the local node belongs to. The argument `pfld` specifies the field-array to be broadcasted in the primary family, while `sfld` is for the data to be broadcasted in the secondary family. You may be unaware that the local node really has its primary or secondary family, because the function will skip the primary broadcast if it is a leaf and the secondary one if it is the root. It is neither necessary to specify the data count because it is calculated by the library, nor to give MPI data-type to the function because `MPI_DOUBLE_PRECISION` for Fortran or `MPI_DOUBLE` for C is assumed[13].

Fortran Interface

```
subroutine oh3_bcast_field(pfld, sfld, ftype)
  implicit none
  real*8,intent(in)  :: pfld
  real*8,intent(out) :: sfld
  integer,intent(in) :: ftype
end subroutine
```

C Interface

```
void oh3_bcast_field(void *pfld, void *sfld, int ftype);
```

`pfld` should be (the pointer to) the first field-array element at the origin of the primary subodmain. The contents of the field-array are broadcasted from the local node to its helpers in its primary family.

`sfld` should be (the pointer to) the first field-array element at the origin of the secondary subodmain. The broadcasted data in the secondary family is received to the field-array.

`ftype` should be the identifier to specify the type of the field-array.

For example, to broadcast your electromagnetic field-array `eb(6,:,:,:,2)` of type `feb`, you can simply do the following in your Fortran code providing the origins are `eb(:,0,0,0,:)`.

```
call oh3_bcast_field(eb(1,0,0,0,1),eb(1,0,0,0,2),feb)
```

[13] Therefore, your field-arrays should have elements only of double presision floating point data or structures only of them.

As for C field-array of `struct` whose origins are pointed by `eb[0]` and `eb[1]`, what you have to do is simply the following.

```
oh3_bcast_field(eb[0],eb[1],feb);
```

In order to make the interfaces simple as shown above, the function refers to $e_l^b(f)$ and $e_u^b(f)$ for $f =$ `ftype` given in the argument `ftypes` of `oh3_init()`, and the size of primary/secondary subdomain given in `sdoms` and that of the field-array itself set to `fsizes`. Note that the elements to be broadcasted are not only in the subarray defined by $e_l^b(f)$ and $e_u^b(f)$ but also some of outside the subarray as shown in Figure 12 in §3.6.1 for the sake of efficiency. This overrun should not be harmful to the logical correctness of the simulation.

3.6.8 `oh3_allreduce_field()`

The function (subroutine) `oh3_allreduce_field()` performs red-black all-reduce summation of a field-array whose type is specified by its argument `ftype` in the families the local node belongs to. The argument `pfld` specifies the field-array to be reduced in the primary family, while `sfld` is for the data to be reduced in the secondary family. You may be unaware that the local node really has its primary or secondary family, because the function will skip the primary reduction if it is a leaf and the secondary one if it is the root. It is neither necessary to specify the data count because it is calculated by the library, to give MPI data-type to the function because `MPI_DOUBLE_PRECISION` for Fortran or `MPI_DOUBLE` is assumed, nor to tells it how the reduction is done because `MPI_SUM` is assumed[14].

Fortran Interface

```
subroutine oh3_allreduce_field(pfld, sfld, ftype)
  implicit none
  real*8,intent(inout) :: pfld
  real*8,intent(inout) :: sfld
  integer,intent(in)   :: ftype
end subroutine
```

C Interface

```
void oh3_allreduce_field(void *pfld, void *sfld, int ftype);
```

`pfld` should be (the pointer to) the first field-array element at the origin of the primary subodmain. The contents of the field-array are replaced with the sum in the primary family.

`sfld` should be (the pointer to) the first field-array element at the origin of the secondary subodmain. The contents of the field-array are replaced with the sum in the secondary family.

`ftype` should be the identifier to specify the type of the field-array.

For example, to have the sum of your current density field-array `cd(3,:,:,:,2)` of type `fcd`, you can simply do the following in your Fortran code providing the origins are `cd(:,0,0,0,:)`.

[14] Therefore, the function cannot be used for any other reductions than summing up.

```
call oh3_allreduce_field(cd(1,0,0,0,1),cd(1,0,0,0,2),fcd)
```

As for C field-array of `struct` whose origins are pointed by `cd[0]` and `cd[1]`, what you have to do is simply the following.

```
oh3_allreduce_field(cd[0],cd[1],fcd);
```

In order to make the interfaces simple as shown above, the function refers to $e_l^r(f)$ and $e_u^r(f)$ for $f =$ `ftype` given in the argument `ftypes` of `oh3_init()`, and the size of primary/secondary subdomain given in `sdoms` and that of the field-array itself set to `fsizes`. Note that the elements to be reduced are not only in the subarray defined by $e_l^r(f)$ and $e_u^r(f)$ but also some of outside the subarray as shown in Figure 12 in §3.6.1 for the sake of efficiency. This overrun should not be harmful to the logical correctness of the simulation.

3.6.9 `oh3_reduce_field()`

The function (subroutine) `oh3_reduce_field()` performs red-black one-way counterpart of the function `oh3_allreduce_field()`.

Fortran Interface

```
subroutine oh3_reduce_field(pfld, sfld, ftype)
  implicit none
  real*8,intent(inout) :: pfld
  real*8,intent(in)    :: sfld
  integer,intent(in)   :: ftype
end subroutine
```

C Interface

```
void oh3_reduce_field(void *pfld, void *sfld, int ftype);
```

`pfld` should be (the pointer to) the first field-array element at the origin of the primary subodmain. The contents of the field-array are replaced with the sum in the primary family.

`sfld` should be (the pointer to) the first field-array element at the origin of the secondary subodmain. The contents of the field-array remain unchanged.

`ftype` should be the identifier to specify the type of the field-array.

3.6.10 `oh3_exchange_borders()`

The function (subroutine) `oh3_exchange_borders()` exchanges boundary planes of a field-array between adjacent primary subdomains. Then, if specified to do, the boundary planes are broadcasted from the local node to its helpers.

Fortran Interface

```
subroutine oh3_exchange_borders(pfld, sfld, ctype, bcast)
  implicit none
  real*8,intent(inout) :: pfld
  real*8,intent(out)   :: sfld
  integer,intent(in)   :: ctype
  integer,intent(in)   :: bcast
end subroutine
```

C Interface

```
void oh3_reduce_field(void *pfld, void *sfld, int ctype, int bcast);
```

`pfld` should be (the pointer to) the first field-array element at the origin of the primary subodmain. The boundary planes (or line segments) of the field-array are sent/ received to/from the nodes which are responsible for the subdomains adjoining to the primary subdomain of the local node as their primary ones.

`sfld` should be (the pointer to) the first field-array element at the origin of the secondary subodmain. The boundary planes of the field-array are replaced with that in the helpand of the local node, if `bcast` is non-zero and we are in secondary mode.

`ctype` should be the identifier to specify the type of the field-array comunication, which is an index of `ctypes` of `oh3_init()`.

`bcast` should be non-zero to broadcast obtained boundary planes to the helpers. If it is 0, only the boundary exchange of the primary subdomain is performed. Note that if we are in primary mode, the broadcast is not performed even if `bcast` $\neq 0$.

For example, you can simply do the following in your Fortran code to exchange boundary data of your electromagnetic field-array `eb(6,:,:,:,2)` of communication type `ceb`, providing the origins are `eb(:,0,0,0,:)` and you do not want to broadcast the received boundary planes.

```
call oh3_exchange_borders(eb(1,0,0,0,1),eb(1,0,0,0,2),ceb,0)
```

As for C field-array of `struct` whose origins are pointed by `eb[0]` and `eb[1]`, what you have to do is simply the following.

```
oh3_exchange_borders(eb[0],eb[1],ceb,0);
```

By these simple statements, you can achieve fairly complicated communications as shown in Figure 13 of Sectin 3.6.1 because `oh3_exchange_borders()` takes care of various matters. First, it of course follows the specifications of the number of planes and their sources and destinations in the field-array given through the argument `ctypes` of `oh3_init()`. The specifiations are also used to determine the size of a plane depending on the axis along which a communication is taken place. That is, the function enlarges the planes to be exchanged as it proceeds the communication from along x-axis then y and to z-axis, so that the local node obtains boundary data not only from the subdomains contacted with planes but also with edges and vertices as shown in Figure 13. Finally, to have the shape of the set of planes to be transferred and to represent them with a derivative data type of MPI, the function consults the size of primary/secondary subdomain given in `sdoms` and that of the field-array itself set to `fsizes`.

The finely designed boundary communication above is especially helpful for more complicated communications required to have the sum of current densities of a grid point around a vertices connecting subdomains. As shown in Figure 14 of §3.6.1, you can have 3^D partial sums calculated by 3^D families by a simple definition in `ctypes` and the following simple call in Fortran, providing your current density field-array is `cd(3,:,:,:,2)` and its type is `ccd`.

```
call oh3_exchange_borders(cd(1,0,0,0,1),cd(1,0,0,0,2),ccd,1)
```

Note that the boundary planes obtained by the communication between adjoined primary subdomains are broadcasted to the helpers of the local node if necessary in the example above. The C counterpart of the example is also simple as follows.

```
oh3_exchange_borders(cd[0],cd[1],ccd,1);
```

3.7 Level-4p Extension and Its Functions

3.7.1 Position-Aware Particle Management

The level-4p extension is for *position-aware* particle management for which the load balancing particle transfer mechanism provided by `oh4p_transbound()` takes care that (almost) all particles in a *grid-voxel* are accommodated by a particular node. In addition, the function gives you a *per-grid histogram* in an array, say

$$\texttt{pghgram}(\phi_x^l\texttt{:}\phi_x^u\texttt{−1, }\ \phi_x^l\texttt{:}\phi_x^u\texttt{−1, }\ \phi_x^l\texttt{:}\phi_x^u\texttt{−1, }\ S\texttt{, 2)}$$

for Fortran where ϕ_d^l and ϕ_d^u ($d \in \{x, y, z\}$) are given by an API function `oh4p_init()` based on the shape of the largest subdomain. By referring to $\texttt{pghgram}(x, y, z, s, c)$ you can know the number of primary ($c = 1$) or secondary($c = 2$) particles of species s resinding in a grid-voxel whose integer coordinates local to its residing subdomain are (x, y, z) where $(0, 0, 0)$ is at the bottom-south-west corner of the primary/secondary subdomain. For C, the array is

$$\texttt{pghgram[}\phi_x \times \phi_y \times \phi_z \times S \times 2\texttt{]}$$

where $\phi_d = \phi_d^u - \phi_d^l$, and the particle population in a grid-voxel at (x, y, z) is $\texttt{pghgram}[c][s][z][y][x]$ conceputually, where $c \in \{0, 1\}$ and $s \in [0, S)$.

Moreover, the primary/secondary particles of a species accommodated by a node is *sorted* in its particle buffer, say `pbuf`, according to the coordinates of their resident grid-voxels as follows. Unlike the lower level couterpart, `pbuf` should accommodate $2P_{lim}$ particles where P_{lim} is given to the library as the argument `maxlocalp` of `oh4p_init()`. Then on the t-th ($t \geq 1$) call of `oh4p_transbound()`, the first half `pbuf(:,1)` or `pbuf[0][]` should have the *input* particles to the function which *outputs* the result of particle transfer and sorting to the second half `pbuf(:,2)` or `pbuf[1][]` if t is odd, while the roles of first and second half are switched if t is even.

Let $base(c, s)$ be the index of the first primary ($c = 0$) or secondary ($c = 1$) particle of species s, i.e.,

$$base(c, s) = \sum_{i=0}^{c-1} \sum_{s'=1}^{S} \texttt{totalp}(s', i{+}1) + \sum_{s'=1}^{s-1} \texttt{totalp}(s', c{+}1) + 1$$

for Fortran, while

$$base(c, s) = \sum_{i=0}^{c-1} \sum_{s'=0}^{S-1} \texttt{totalp}[i][s'] + \sum_{s'=0}^{s-1} \texttt{totalp}[c][s']$$

for C. Then the particles after $(2t{+}k)$-th ($k \in \{0, 1\}$) call of `oh4p_transbound()` and in $(0, 0, 0)$ are in

$$\texttt{pbuf}(base(c, s)\texttt{:}\, base(c, s)\texttt{+pghram}(x, y, z, s, c{+}1)\texttt{−1, }\ k{+}1) \qquad \text{(Fortran)}$$
$$\texttt{pbuf}[k][base(c, s)]\texttt{, }\ \ldots\texttt{, }\ \texttt{pbuf}[k][base(c, s)\texttt{+pghgram}[c][s][z][y][x]\texttt{−1}] \qquad \text{(C)}$$

followed by those in $(0, 0, 1)$, then $(0, 0, 2)$ and so on. For example, a Fortran code snip to visit all particles in each grid-voxel is as follows.

```
if (has_secondary_subdomain()) then;  cc=2;  else;  cc=1;  end if
do c=1, cc
  b = pbase(c)
  do s=1, nspec
    base(s) = b;  b = b + totalp(s,c)
  end do
  do z=0, sdoms(2,3,sdid(c))-sdoms(1,3,sdid(c))-1
    do y=0, sdoms(2,2,sdid(c))-sdoms(1,2,sdid(c))-1
      do x=0, sdoms(2,1,sdid(c))-sdoms(1,1,sdid(c))-1
        do s=1, nspec
          do i=1, pghgram(x,y,z,s,c)
            call do_something(pbuf(base(s)+i,k+1))
          end do
          base(s) = base(s) + pghgram(x,y,z,s,c)
end do;  end do;  end do;  end do;  end do;
```

The C's counterpart of the code above will be as follows.

```
for (c=0; c<has_secondary_subodmain() ? 2 : 1; c++) {
  b = pbase[c];
  for (s=0; s<nspec; s++) {
    base[s] = b;  b += totalp[c][s];
  }
  for (z=0; z<sdoms[sdid[c]][2][1]-sdoms[sdid[c]][2][0]; z++) {
    for (y=0; y<sdoms[sdid[c]][1][1]-sdoms[sdid[c]][1][0]; y++) {
      for (x=0; x<sdoms[sdid[c]][0][1]-sdoms[sdid[c]][0][0]; x++) {
        for (s=0; s<nspec; s++) {
          for (i=0; i<pghgram[c][s][z][y][x]; i++)
            do_something(pbuf[k][base[s]+i]);
          base[s] += pghgram[c][s][z][y][x];
} } } } }
```

An important notice is that the maintenance of per-grid histogram is up to library as well as the per-subdomain counterpart which is referred to as `nphgram` in lower levels. Therefore, you have to call `oh4p_map_particle_to_neighbor()` or `oh4p_map_particle_to_subdomain()` once for each and every particle[15], before each call of `oh4p_transbound()`, in order to let the library know the particle position. Since these functions have to examine the position of a particle, the structure of `oh_particle` for Fortran or `S_particle structure` for C must have double-precision floating-point elements `x`, `y` and `z` if your simulator is three-dimensional.

You also have to remember that `nid` element is (almost) meaningless for you because the mapping functions encode the information to identify the subdomain and grid-voxel in which the particle resides in the element. Moreover, if both of the number of nodes and the size of each subdomain are large, i.e., your whole space domain is large having grid-voxels more than about 10^9, you have to `#define` the macro `OH_BIG_SPACE` in `oh_config.h` to let `nid` element be `long␣long␣int`. More specifically, you have to do `#define` the macro if

[15] Except for those eliminated by setting their `sid` elements to -1 as discussed in §3.9.

the following holds.

$$G = \left\lceil \prod_{d=0}^{D-1} \phi_d \right\rceil \qquad (N + 3^D)2^G \geq 2^{31}$$

Due to the encoding of `nid` element and the delegation of the histogram management to the library, it might become tough for you to find and fix problems caused by some improper usage of API functions, especially those for particle mapping, injection and removal. Therefore, the following functions check the consistency of their arguments you give, unless you `#define` the macro `OH_NO_CHECK` in `oh_config.h` to mean your code is well debugged and thus the consistency check should be omitted to eliminate a few percent overhead.

```
oh4p_map_particle_to_neighbor()  oh4p_map_particle_to_subdomain()
oh4p_inject_particle()  oh4p_remove_mapped_particle()
oh4p_remap_particle_to_neighbor()  oh4p_remap_particle_to_subdomain()
```

Another important notice is that `oh4p_transbound()` does its best to make all particles in a grid-voxel accommodated by a node but cannot do it if the grid-voxel has too many particles. That is, we could have an extreme case in which all particles in the simulated system are concentrated in a grid-voxel and thus we cannot let a node accommodate all of them. To cope with such concentration, you have to define a threshold P_{hot} to allow `oh4p_transbound()` to split the set of particles in a grid-voxel into subsets each of which has a cardinality not less than P_{hot}. In other words, `oh4p_transbound()` may split a set of particles in a *hot-spot* grid-voxel having cardinality of $2P_{hot}$ or greater if otherwise a node should have primary/secondary particles more than ordered by the load balancing algorithm by $2P_{hot}$ or more. Therefore, a node may have $P_{\max} + 4P_{hot}$ particles and thus the particle buffer should be large enough to accommodate them.

The specific value of P_{hot} should be determined trading off two factors; greater value will satisfy the law of large numbers better when you pick a set of particles from those in a grid-voxel (e.g., a pair of colliding particles) while load imbalance will be severer and required particle buffer size will be larger. A compromization will be found at around 10-times of the average number of particles in a grid-voxel, but of course the decision is up to you. The value of P_{hot} should be passed to `oh4p_max_local_particles()` which will tell you the (minimum) size of the particle buffer taking $4P_{hot}$ margin into account.

3.7.2 Level-4p Functions

Level-4p extension provides the following functions.

`oh4p_init()` performs initialization similar to what `oh3_init()` and lower level counterparts do and that of level-4p's own for position-aware particle management.

`oh4p_max_local_particles()` calculates the size of particle buffers taking the hot-spot threshold P_{hot} into account.

`oh4p_per_grid_histogram()` tells other library functions where the per-grid histogram is located in your code.

`oh4p_transbound()` performs position-aware load balancing and particle transfer.

`oh4p_map_particle_to_neighbor()` finds the subdomain and grid-voxel which will be the residence of a particle that stays in the original subdomain or travel to its neighbor.

`oh4p_map_particle_to_subdomain()` finds the subdomain and grid-voxel which will be the residence of a particle that may go to any subdomains.

`oh4p_inject_particle()` injects a particle to the bottom of the particle buffer.

`oh4p_remove_mapped_particle()` removes a particle which you have mapped by `oh4p_map_particle_to_neighbor()` or `oh4p_map_particle_to_subdomain()`, or injected by `oh4p_inject_particle()` after the last call of `oh4p_transbound()`.

`oh4p_remap_particle_to_neighbor()` does what `oh4p_remove_mapped_particle()` and `oh4p_map_particle_to_neighbor()` do.

`oh4p_remap_particle_to_subdomain()` does what `oh4p_remove_mapped_particle()` and `oh4p_map_particle_to_subdomain()` do.

The function API for Fortran programs is given by the module named `ohhelp4p` in the file oh_mod4p.F90, while API for C is embedded in ohhelp_c.h.

3.7.3 `oh4p_init()`

The function (subroutine) `oh4p_init()` receives a number of fundamental parameters and arrays through which `oh4p_transbound()` interacts with your simulator body. It also initializes internal data structures used in level-4p and lower level libraries. Among its twenty-two arguments, other library functions directly refer to only the bodies of the argument `pbuf` as their implicit inputs. Therefore, after the call of `oh4p_init()`, modifying the bodies of other arguments has no effect to library functions.

Fortran Interface

```
subroutine oh4p_init(sdid, nspec, maxfrac, totalp, pbuf, pbase, &
                     maxlocalp, mycomm, nbor, pcoord, sdoms, scoord, &
                     nbound, bcond, bounds, ftypes, cfields, ctypes, &
                     fsizes, &
                     stats, repiter, verbose)
  use oh_type
  implicit none
  integer,intent(out)   :: sdid(2)
  integer,intent(in)    :: nspec
  integer,intent(in)    :: maxfrac
  integer,intent(out)   :: totalp(:,:)
  type(oh_particle),intent(inout) :: pbuf(:)
  integer,intent(out)   :: pbase(3)
  integer,intent(in)    :: maxlocalp
  type(oh_mycomm),intent(out) :: mycomm
  integer,intent(inout) :: nbor(3,3,3)        ! for 3D codes.
  integer,intent(in)    :: pcoord(OH_DIMENSION)
  integer,intent(inout) :: sdoms(:,:,:)
  integer,intent(in)    :: scoord(2,OH_DIMENSION)
  integer,intent(in)    :: nbound
  integer,intent(in)    :: bcond(2,OH_DIMENSION)
  integer,intent(inout) :: bounds(:,:,:)
  integer,intent(in)    :: ftypes(:,:)
  integer,intent(in)    :: cfields(:)
  integer,intent(in)    :: ctypes(:,:,:,:)
```

```
  integer,intent(out)   :: fsizes(:,:,:)
  integer,intent(in)    :: stats
  integer,intent(in)    :: repiter
  integer,intent(in)    :: verbose
end subroutine
```

C Interface

```
void oh4p_init(int **sdid, const int nspec, const int maxfrac, int **totalp,
               struct S_particle **pbuf, int **pbase, const int maxlocalp,
               void *mycomm, int **nbor, int *pcoord, int **sdoms,
               int *scoord, const int nbound, int *bcond, int **bounds,
               int *ftypes, int *cfields, int *ctypes, int **fsizes,
               const int stats, const int repiter, const int verbose);
```

`sdid`
`nspec`
`maxfrac`
`totalp`
: See §3.4.1 because the arguments above are perfectly equivalent to those of `oh1_init()`. Note that `nphgram` which the level-1 to level-3 counterparts have is not a member of the arguments of `oh4p_init()` because maintaining the per-subdomain histogram is perfectly up to the level-4p library functions.

`pbuf(`P_{lim}`)` (for Fortran)
`**pbuf` (for C)
: The argument `pbuf` should be an one-dimensional array of `oh_particle` type structure elements in Fortran, while it should be a double pointer to an array of `S_particle` structure in C. Unlike the level-2 (and level-3) counterpart, the array should be large enough to accommodate $2P_{lim}$ particles, where P_{lim} is given through the argument `maxlocalp` and should not be less than $P_{\max}$ at any time. The buffer is conceptually split into two portions of equal size, i.e., P_{lim}. At the first call of `oh4p_transbound()`, the first half should have the particles which the node accommmodates at initial, and the second half will have the primary/secondary particles for the node in the next (usually first) simulation step. Then you will update velocities and positions of the particles in the second half and call `oh4p_transbound()` again to have the particles for the next step in the first half. This buffer switching continues alternating the role of first and second halves each time you call `oh4p_transbound()`.

  Note that this double buffering does *not* increase the required memory size for particles from simulations with lower level libraries. That is, when you use the level-2 or level-3 libraries the second half is hidden from you but the library functions keep it for particle transfer. Also note that C coded simulator body can pass `pbuf` having a pointer to `NULL` (not `NULL` itself) to make `oh4p_init()` allocate the buffer for you and return the pointer to it through the argument.

`pbase`
`maxlocalp`
: See §3.5.2 because the arguments above are perfectly equivalent to those of `oh2_init()`.

`mycomm`

`nbor`
`pcoord`
: See §3.4.1 because the arguments above are perfectly equivalent to those of `oh1_init()`.

`sdoms`
`scoord`
`nbound`
`bcond`
`bounds`
`ftypes`
`cfields`
`ctypes`
: See §3.6.1 because the arguments above are perfectly equivalent to those of `oh3_init()`.

`fsizes(2,`D`,`F`+1)` (for Fortran)
`**fsizes()` (for C)
: The argument `fsizes` should be a three-diemsional array of integers in Fortran where F is the number of field-arrays defined by `ftypes`. In C, it should be a double pointer to such an array of $(F+1) \times D \times 2$ to form `fsizes`$[F+1][D][2]$ conceptually, or a pointer to `NULL` (not `NULL` itself) if you want the library to allocate the array and return the pointer to it through the argument. In any cases, the array element `fsizes`(β, d, f) $(f \in [1, F])$ or `fsizes`$[f][d][\beta]$ $(f \in [0, F))$ will have $\phi_d^l(f)$ $(\beta = 0)$ or $\phi_d^u(f)$ $(\beta = 1)$ for the field-arrays of type f to notify you that the required size of field-arrays as the counterpart of `oh3_init()` does. The difference is that `oh4p_init()`'s has one additional element set of `fsizes`$(\beta, d, F+1)$ or `fsizes`$[F][d][\beta]$ for the per-grid histogram you must (or may) allocate.

`stats`
`repiter`
`verbose`
: See §3.4.1 because the arguments above are perfectly equivalent to those of `oh1_init()`.

3.7.4 `oh4p_max_local_particles()`

The function `oh4p_max_local_particles()` calculates the absolute maximum number of particles which a node can accommodate and returns it to its caller, as the level-2 counterpart `oh2_max_local_particles()` shown in §3.5.3 does. The difference is that this function has one additional argument `hsthresh` for the hot-spot threshold P_{hot} and takes it into account for the calculation. The return value can be directly passed to the argument `maxlocalp` of `oh4p_init()`.

Fortran Interface

```
integer function oh4p_max_local_particles(npmax, maxfrac, minmargin, &
                                          hsthresh)
  implicit none
  integer*8,intent(in) :: npmax
  integer,intent(in)   :: maxfrac
  integer,intent(in)   :: minmargin
```

```
  integer,intent(in)   :: hsthresh
end function
```

C Interface

```
int  oh4p_max_local_particles(const dint npmax, const int maxfrac,
                              const int minmargin, const int hsthresh);
```

`npmax` should be the absolute maximum number of particles which your simulator is capable of as a whole.

`maxfrac` should have the tolerance factor percentage of load imbalance α and should be same as the argument `maxfrac` of `oh4p_init()`.

`minmargin` should be the minimum margin by which the return value P_{lim} has to clear over the per node average of `npmax`.

`hsthresh` should be the hot-spot threshold P_{hot} to define the minimum cardinality of a subset split from the set of a concentrated grid-voxel when the particles in it are assigned to two or more nodes.

return value is the number of particles P_{lim} given by the following.

$$\overline{P} = \lceil \texttt{npmax}/N \rceil \qquad P_{lim} = \max(\lceil \overline{P}(100+\alpha)/100 \rceil\ ,\ \overline{P} + \texttt{minmargin}) + 4P_{hot}$$

Note that `minmargin` is the margin over $\overline{P}$ to be kept besides the tolerance factor α for, e.g., initial particle accommodation in each node. Therefore it does not assure that a node has a room for `minmargin` particles in simulation. If you need such a room for, e.g., particle injection, add the room to P_{lim} to give it the argument `maxlocalp` of `oh4p_init()`. Also note that `oh4p_init()` confirms that this function has been called prior to the call of it and its `maxlocalp` argument is not less than the return value of this function, or abort the execution if both or either of them don't hold.

3.7.5 `oh4p_per_grid_histogram()`

The function (subroutine) `oh4p_per_grid_histogram()` is to let level-4p library functions know where the array of per-grid histogram is located in your simulator body, or to allocate the array for you.

Fortran Interface

```
subroutine oh4p_per_grid_histogram(pghgram)
  implicit none
  integer,intent(inout) :: pghgram
end subroutine
```

C Interface

```
void oh4p_per_grid_histogram(int **pghgram);
```

`pghgram` (for Fortran)

`**pghgram` (for C)

The argument `pghgram` for Fortran should be the origin of $(D+2)$-dimensional array for the per-grid histogram, say `h(0,0,0,1,1)` for the particles of the first species in the grid-voxel at $(0,0,0)$ (if three-dimensional simulation) of the primary subdomain. In C, it should be a double pointer to such an array element, say `&&h[0][0][0][0][0]`, or a pointer to `NULL` (not `NULL` itself) if you want the library to allocate the array and return the pointer to its origin element through the argument. Note that if you give the origin element to the function, the array must have the shape, if three-dimensional, $\phi_x \times \phi_y \times \phi_z \times S \times 2$ where $\phi_d = \phi_d^u - \phi_d^l$ and $\phi_d^\beta =$ `fsizes`$(\beta, d, F+1)$ or $\phi_d^\beta =$ `fsizes`$[F][d][\beta]$ obtained through the `fsizes` argument of `oh4p_init()`.

3.7.6 `oh4p_transbound()`

The function `oh4p_transbound()` at first performs operetaions for load balancing as same as that `oh1_transbound()` does; examination of the per-subdomain particle population histogram to check the balancing and (re)building of helpand-helper configuration if necessary. Then for each grid-voxel, it determines the node to accommodate particles in the grid-voxel or a set of nodes to do that if the grid-voxel is a hot-spot. After that particles in the first/second half of the particle buffer, `pbuf` argument of `oh4p_init()`, are transferred to satisfy load balancing and position-awareness. Finally particles in each node are sorted according to the coordinates of grid-voxels in which they reside and the per-grid histogram in the node presented to `oh4p_per_grid_histogram()` is updated to show the number of particles in each grid-voxel that the node accommodates. The sorted result is stored in the second/first half of `pbuf`.

Since the arguments of `oh4p_transbound()` and its return value are perfectly equivalent to those of `oh1_transbound()` (and `oh2_transbound()` and `oh3_transbound()`), see §3.4.4 for their definitions.

Fortran Interface

```
integer function oh4p_transbound(currmode, stats)
  implicit none
  integer,intent(in) :: currmode
  integer,intent(in) :: stats
end function
```

C Interface

```
int oh4p_transbound(const int currmode, const int stats);
```

3.7.7 `oh4p_map_particle_to_neighbor()`

The function `oh4p_map_particle_to_neighbor()` returns the identifier of the subdomain in which the primary (`ps` $= 0$) or secondary (`ps` $= 1$) particle `part` of spec `s` will reside and to which the primary or secondary subdomain of the local node likely adjoins. Alghough the function, unlike the level-3 couterpart `oh3_map_particle_to_neighbor()`, accepts particles traveling a non-neighboring subdomain due to, for example, initial particle distribution or particle warp, using the relative function `oh4p_map_particle_to_subdomain()` is recommended because it is faster for such particles.

Also unlike the level-3 counterpart `oh3_map_particle_to_neighbor()`, you have to call this function or `oh4p_map_particle_to_subdomain()` for *all* particles which the local node

accommodates so that the library maintains the per-subdomain and per-grid histograms. Another differences from the level-3 function are that the particle itself is passed through the first argument rather than its position, and its species `s` has to be given as the third argument.

Fortran Interface

```
integer function oh4p_map_particle_to_neighbor(part, ps, s)
  use oh_type
  implicit none
  type(oh_particle),intent(inout) :: part
  integer,intent(in)   :: ps
  integer,intent(in)   :: s
end function
```

C Interface

```
int  oh4p_map_particle_to_neighbor(struct S_particle *part, const int ps,
                                   const int s);
```

`part` (for Fortran)
`*part` (for C)
: The first argument `part` should be a `oh_particle` type structured data in Fortran, while it should be a pointer to `S_particle` structure in C. In both cases, the actual argument structure may be updated as discussed later.

`ps` should be 0 for a primary particle, or 1 for a secondary particle.

`s` should be the species identifier of the particle in $[1, S]$ in Fortran while in $[0, S)$ in C. Note that if the particle structure has the `spec` element, `s` must be equal to the value of the element of `part`.

return value is the identifier of the subdomain in which the particle will reside, or -1 if such a subdomain is not found as discussed later.

The function at first examines whether the particle is in the primary (`ps` = 0) or secondary (`ps` = 1) subdomain of the local node and returns its identifier if the particle is in it, referring to the subdomain boundaries given by or set to the argument `sdoms` of `oh4p_init()`. Otherwise, it assumes that the particle has moved into a subdomain adjoining to the primary/secondary subdomain and returns the identifier of the subdomain into which the particle has moved, referring to the neighboring infomation given by or set to the argument `nbor` of `oh4p_init()`, or that in the helpand.

In the latter case of the boundary crossing, the periodic boundary condition of the whole space domain is taken care of by the function. Therefore, the coordinates given by `x`, `y` and `z` elements of the argument `part` should be *raw* ones without wraparound. Moreover, the elements in the actual argument are updated by the function if the particle has crossed a periodic boundary. For example, if the particle has crossed the periodic boundary plane perpendicular to x-axis, the actual argument variable x is updated as follows.

$$x \leftarrow \begin{cases} x + (\Delta_x^u - \Delta_x^l)\gamma_x & x < \Delta_x^l \cdot \gamma_x \\ x - (\Delta_x^u - \Delta_x^l)\gamma_x & x \geq \Delta_x^u \cdot \gamma_x \end{cases}$$

On the other hand, if the particle has crossed a non-periodic boundary of the whole space domain, the function returns -1 to indicate that the particle is out of bounds[16]. To examine the boundary condition, the function refers to the conditions given through the argument `bcond` or `bounds` of `oh4p_init()`. The function also returns -1 if the particle has moved into a non-exsistent neighbor, which may be defined by `nbor`.

3.7.8 `oh4p_map_particle_to_subdomain()`

The function `oh4p_map_particle_to_subdomain()` returns the identifier of the subdomain in which the primary (`ps` $= 0$) or secondary (`ps` $= 1$) particle `part` of spec `s` will reside. The difference between this and the relative function `oh4p_map_particle_to_neighbor()` is that this function more quickly find the identifier of the non-neighboring resident subdomain for the particle and thus is desgined to be used for, e.g., initial particle disribution, particle warp, and so on. Of course you may use this function always but have to remember that it is much slower than `oh4p_map_particle_to_neighbor()` for particles staying in the primary/secondary subdomain or just crossing a subdomain boundary.

Unlike the level-3 counterpart `oh3_map_particle_to_subdomain()`, you have to call this function or `oh4p_map_particle_to_neighbor()` for *all* particles which the local node accommodates so that the library maintains the per-subdomain and per-grid histograms. The other differences from the level-3 function are that the particle itself is passed through the first argument rather than its position, its primariness/secondariness has to be given as the second argument `ps`, and its species `s` has to be given as the third argument. In addition, this function takes care of crossing periodic boundaries of the whole system.

Since the arguments of `oh4p_map_particle_to_subdomain()` and its return value are perfectly equivalent to those of `oh4p_map_particle_to_neighbor()`, though this function is much slower and thus you are discouraged to use it in usual cases, see §3.7.7 for their definitions.

Fortran Interface

```
integer function oh4p_map_particle_to_subdomain(part, ps, s)
  use oh_type
  implicit none
  type(oh_particle),intent(inout) :: part
  integer,intent(in)   :: ps
  integer,intent(in)   :: s
end function
```

C Interface

```
int  oh4p_map_particle_to_subdomain(struct S_particle *part, const int ps,
                                    const int s);
```

3.7.9 `oh4p_inject_particle()`

The function `oh4p_inject_particle()` injects a given particle at the bottom of `pbuf` and maintains per-subdomain and per-grid histograms according to its residence subdomain,

[16] The values in elements of `part` are kept unless the particle has crossed two or more contacting space domain boundaries including periodic ones at once. More specifically, the function examines boundary crossing in the order of yz, xz and then xy planes if $D = 3$, and updates `part`'s elements `x`, `y` and `z` in this order if the corresponding boundary planes are periodic.

grid-voxel, primariness and species. Note that the number of particles injected in a simulation step should not be greater than $P_{lim} - Q_n$.

Fortran Interface

```
integer function oh4p_inject_particle(part, ps)
  use oh_type
  implicit none
  type(oh_particle),intent(inout) :: part
  integer,intent(in)   :: ps
end function
```

C Interface

```
int  oh4p_inject_particle(const struct S_particle *part, const int ps);
```

`part` (for Fortran)
`*part` (for C)
: The argument `part` should be a `oh_particle` structure in Fortran, or a pointer to `S_particle` structure in C, to be injected. Elements except for `nid` in the given particle structure should be completely set with significant values in advance, especially if $S \neq 1$, `spec` elements which are referred to by the function to update histograms.

`ps` should be 0 for a primary particle, or 1 for a secondary particle. Sepcifying primariness/secondariness is important for good performace if the particle is injected into (or around) primary/secondary subdomain of the local node.

return value is the identifier of the subdomain in which the particle will reside, or -1 if such a subdomain is not found.

3.7.10 `oh4p_remove_mapped_particle()`

The function (subroutine) `oh4p_remove_mapped_particle()` removes a particle which you have mapped by `oh4p_map_particle_to_neighbor()` or `oh4p_map_particle_to_subdomain()`, or injected by `oh4p_inject_particle()` after the last call of `oh4p_transbound()`. Since the mapping or injection incremented counter elements in the per-subdomain and per-grid histograms, you have to call this function to cancel the increment when you discard the particle, instead of setting its `nid` element to -1.

Fortran Interface

```
subroutine oh4p_remove_mapped_particle(part, ps, s)
  use oh_type
  implicit none
  type(oh_particle),intent(inout) :: part
  integer,intent(in)   :: ps
  integer,intent(in)   :: s
end subroutine
```

C Interface

```
void oh4p_remove_mapped_particle(struct S_particle *part, const int ps,
                                 const int s);
```

`part` (for Fortran)
`*part` (for C)
: The argument `part` should be a `oh_particle` structure in Fortran, or a pointer to `S_particle` structure in C, to be removed.

`ps` should be 0 for a primary particle, or 1 for a secondary particle.

`s` should be the species identifier of the particle in $[1, S]$ in Fortran while in $[0, S)$ in C.

Note that the `nid` element of the particle `part` is set to -1 by the function.

3.7.11 `oh4p_remap_particle_to_neighbor()`

The function `oh4p_remap_particle_to_neighbor()` cancels the mapping of the primary (`ps` = 0) or secondary (`ps` = 1) particle `part` of spec `s` done by functions such as `oh4p_map_particle_to_neighbor()` and then find the subdomain in which the particle will reside to return its identifer. That is, this function does in series what `oh4p_remove_mapped_particle()` and `oh4p_map_particle_to_neighbor()` do.

Fortran Interface

```
integer function oh4p_remap_particle_to_neighbor(part, ps, s)
  use oh_type
  implicit none
  type(oh_particle),intent(inout) :: part
  integer,intent(in)   :: ps
  integer,intent(in)   :: s
end function
```

C Interface

```
int  oh4p_remap_particle_to_neighbor(struct S_particle *part, const int ps,
                                     const int s);
```

`part` (for Fortran)
`*part` (for C)
: The argument `part` should be a `oh_particle` structure in Fortran, or a pointer to `S_particle` structure in C, to be remapped.

`ps` should be 0 for a primary particle, or 1 for a secondary particle.

`s` should be the species identifier of the particle in $[1, S]$ in Fortran while in $[0, S)$ in C.

3.7.12 `oh4p_remap_particle_to_subdomain()`

The function `oh4p_remap_particle_to_subdomain()` cancels the mapping of the primary (`ps` = 0) or secondary (`ps` = 1) particle `part` of spec `s` done by functions such as `oh4p_map_particle_to_neighbor()` and then find the subdomain in which the particle will reside to return its identifer. That is, this function does in series what `oh4p_remove_mapped_particle()` and `oh4p_map_particle_to_subdomain()` do.

Fortran Interface

```
integer function oh4p_remap_particle_to_subdomain(part, ps, s)
  use oh_type
  implicit none
  type(oh_particle),intent(inout) :: part
  integer,intent(in)   :: ps
  integer,intent(in)   :: s
end function
```

C Interface

```
int  oh4p_remap_particle_to_subdomain(struct S_particle *part, const int ps,
                                      const int s);
```

`part` (for Fortran)
`*part` (for C)
: The argument `part` should be a `oh_particle` structure in Fortran, or a pointer to `S_particle` structure in C, to be remapped.

`ps` should be 0 for a primary particle, or 1 for a secondary particle.

`s` should be the species identifier of the particle in $[1, S]$ in Fortran while in $[0, S)$ in C.

3.8 Level-4s Extension and Its Functions

3.8.1 Position-Aware Particle Management in Level-4s

The level-4s extension is similar to the level-4p counterpart to provide you of position-aware particle management, but the load balancing particle transfer mechanism given by `oh4s_transbound()` has the following features different from the level-4p counterpart.

- A node n responsible of a subdomain n^p ($p \in \{0, 1\}$) as its primary ($n^0 = n$) or secondary ($n^1 = \mathit{parent}(n)$) subdomain accommodates all particles in the *subcuboid*;

  $$[\delta_x^l(n^p),\, \delta_x^u(n^p)) \times [\delta_y^l(n^p),\, \delta_y^u(n^p)) \times [\delta_z^l(n^p)+\zeta_p^l(n),\, \delta_z^l(n^p)+\zeta_p^u(n))$$

  where $0 \le \zeta_p^l(n) \le \zeta_p^u(n) \le \delta_z^u(m) - \delta_z^l(m)$. That is, the subcuboid consists of grid-voxels in the subdomain n^p whose local z-coordinates are in $[\zeta_p^l(n), \zeta_p^u(n))$. The function (subroutine) `oh4s_transbound()` determine $\zeta_p^\beta(n)$ ($\beta \in \{l, u\}$) and *returns* them through `oh4s_init()`'s argument array `zbound`. Unlike the level-4p counterpart, it is assured that all particles in a subcuboid is accommodated by a particular node, but this requires that the particle population in a grid-voxel, or the *density*, should have a certain upper bound. Therefore, you have to determine this *maximum density* $\mathcal{D}$ and show it to the library through `maxdensity` argument of `oh4s_init()`.

- In addition to the particles in the subdomain n^p's subcuboid responsible of, the node n above also accommodates *halo* particles residing in grid-voxels just outside the surface of the subcuboid. That is, halo particles are those residing in the set of grid-voxels whose coordinates local to the subdomain n^p are in the following where $\delta_d(m) = \delta_d^u(m) - \delta_d^l(m)$.

  $$[-1,\, \delta_x(n^p)+1) \times [-1,\, \delta_y(n^p)+1) \times [\zeta_p^l(n)-1,\, \zeta_p^u(n)+1) -$$
  $$[0,\, \delta_x(n^p)) \times [0,\, \delta_y(n^p)) \times [\zeta_p^l(n),\, \zeta_p^u(n))$$

These halo particles assure that, for every particle residing at the position (x, y, z) in the subcuboid of the node n, all particles in the sphere with center (x, y, z) and radius $\min(\gamma_x, \gamma_y, \gamma_z)$ are accommodated by the node n.

- In addition to the per-grid histogram whose element `pghgram`(c, s, x, y, z) for Fortran or `pghgram`$[z][y][x][s][c]$ for C having the number of primary ($c = 1$ in Fortran while $c = 0$ in C) or secondary ($c = 2$ in Fortran while $c = 1$ in C) particles of species s in the grid-voxel (x, y, z), `oh4s_transbound()` gives you the index of the first particle in it through the second argument *per-grid index* array, say `pgindex`(c, s, x, y, z) or `pgindex`$[z][y][x][s][c]$ of `oh4s_per_grid_histogram()`. With this index array, particles in all grid-voxels the local node is responsible of after $(2t+k)$-th ($k \in \{0, 1\}$) can be visited by the following Fortran code snip.

```
if (has_secondary_subdomain()) then;  cc=2;  else;  cc=1;  end if
do c=1, cc
  do z=zbound(1,c), zbound(2,c)-1
    do y=0, sdoms(2,2,sdid(c))-sdoms(1,2,sdid(c))-1
      do x=0, sdoms(2,1,sdid(c))-sdoms(1,1,sdid(c))-1
        do s=1, nspec
          do i=0, pghgram(x,y,z,s,c)-1
            call do_something(pbuf(pgindex(x,y,z,s,c)+i,k+1))
          end do
end do;  end do;  end do;  end do;  end do;
```

In C, the code snip corresponding to above is as follows.

```
for (c=0; c<has_secondary_subodmain() ? 2 : 1; c++) {
  for (z=zbound[c][0]; z<zbound[c][1]; z++) {
    for (y=0; y<sdoms[sdid[c]][1][1]-sdoms[sdid[c]][1][0]; y++) {
      for (x=0; x<sdoms[sdid[c]][0][1]-sdoms[sdid[c]][0][0]; x++) {
        for (s=0; s<nspec; s++) {
          for (i=0; i<pghgram[c][s][z][y][x]; i++)
            do_something(pbuf[k][pgindex[c][s][z][y][x]+i]);
} } } } }
```

Moreover, for a particle p in the grid-voxel (x, y, z), all particles whose distance from p can be less than $\min(\gamma_x, \gamma_y, \gamma_x)$ can be found by the following Fortran code snip.

```
do dz=-1,1;  do dy=-1,1;  do dx=-1,1
 do i=0, pghgram(x+dx,y+dy,z+dz,s,c)-1
   call do_something(pbuf(pgindex(x+dx,y+dy,z+dz,s,c)+i,k+1))
 end do
end do;  end do;  end do;
```

The C version of the code above is as follows.

```
for (dz=-1;dz<2;dz++)  for (dy=-1;dy<2;dy++)  for (dx=-1;dx<2;dx++) {
  for (i=0; i<pghgram[c][s][z+dz][y+dy][x+dx]; i++)
    do_something(pbuf[k][pgindex[c][s][z+dz][y+dy][x+dx]+i]);
}
```

Note that `pghgram` and `pgindex` are meaningful for halo region with $x = -1$, $x = \delta_x(n^p)$, etc, so that you may access halo particles in the code snip shown above.

- Besides the particle transfer mechanism provided by `oh4s_transbound()`, the level-4s library provides you of inter-node transfer of the halo part of any one-dimensional particle-associated array whose layout is *similar* to the particle buffer. For example, suppose your simulation code has a vector v in each node and its i-th element corresponds to the i-th particle in the particle buffer of the node. The function (subroutine) `oh4s_exchange_border_data()` takes the vector v (and send/receive buffers and data-type as discussed in §3.8.7) to send v's elements in grid-voxels whose local coordinate (x_s, y_s, z_s) of local subdomain m satisfies;

  $$x_s = 0 \ \vee \ x_s = \delta_x(m) - 1 \ \vee \ y_s = 0 \ \vee \ y_s = \delta_y(m) - 1 \ \vee \ z_s = 0 \ \vee \ z_s = \delta_z(m) - 1$$

  to the nodes responsible of m's neighbors, and to receives elements for m's local coordinate (x_r, y_r, z_r) satisfying;

  $$x_s = -1 \ \vee \ x_s = \delta_x(m) \ \vee \ y_s = -1 \ \vee \ y_s = \delta_y(m) \ \vee \ z_s = -1 \ \vee \ z_s = \delta_z(m)$$

  from the neighbor nodes. This function will be convenient to implement, e.g., an iterative linear solver of unknowns corresponding to particles.

3.8.2 Level-4s Functions

Level-4s extension provides the following functions.

`oh4s_init()` performs initialization similar to what `oh4p_init()` does with a few modifications for level-4s's own features.

`oh4s_particle_buffer()` tells other library functions where the particle buffer is located in your code.

`oh4s_per_grid_histogram()` tells other library functions where the per-grid histogram and index arrays are located in your code.

`oh4s_transbound()` performs position-aware load balancing and particle transfer.

`oh4s_exchange_border_data()` transfers one-dimensional array elements corresponding to halo particles.

`oh4s_map_particle_to_neighbor()` finds the subdomain and grid-voxel which will be the residence of a particle that stays in the original subdomain or travel to its neighbor.

`oh4s_map_particle_to_subdomain()` finds the subdomain and grid-voxel which will be the residence of a particle that may go to any subdomains.

`oh4s_inject_particle()` injects a particle to the bottom of the particle buffer.

`oh4s_remove_mapped_particle()` removes a particle which you have mapped by `oh4s_map_particle_to_neighbor()` or `oh4s_map_particle_to_subdomain()`, or injected by `oh4s_inject_particle()` after the last call of `oh4s_transbound()`.

`oh4s_map_particle_to_neighbor()` does what `oh4s_remove_mapped_particle()` and `oh4s_map_particle_to_neighbor()` do.

`oh4s_map_particle_to_subdomain()` does what `oh4s_remove_mapped_particle()` and `oh4s_map_particle_to_subdomain()` do.

The function API for Fortran programs is given by the module named `ohhelp4s` in the file oh_mod4s.F90, while API for C is embedded in ohhelp_c.h.

3.8.3 `oh4s_init()`

The function (subroutine) `oh4s_init()` receives a number of fundamental parameters and arrays through which `oh4s_transbound()` and other library functions interacts with your simulator body. It also initializes internal data structures used in level-4s and lower level libraries. Though some of 26 arguments are modified by `oh4s_transbound()`, it and other library functions will not directly refer to any of them. Therefore, after the call of `oh4s_init()`, modifying the bodies of arguments has no effect to library functions.

Fortran Interface

```
subroutine oh4s_init(sdid, nspec, maxfrac, npmax, minmargin, maxdensity, &
                     totalp, pbase, maxlocalp, cbufsize, mycomm, nbor, &
                     pcoord, sdoms, scoord, nbound, bcond, bounds, &
                     ftypes, cfields, ctypes, fsizes, zbound, &
                     stats, repiter, verbose)
  use oh_type
  implicit none
  integer,intent(out)   :: sdid(2)
  integer,intent(in)    :: nspec
  integer,intent(in)    :: maxfrac
  integer*8,intent(in)  :: npmax
  integer,intent(in)    :: minmargin
  integer,intent(in)    :: maxdensity
  integer,intent(out)   :: totalp(:,:)
  integer,intent(out)   :: pbase(3)
  integer,intent(out)   :: maxlocalp
  integer,intent(out)   :: cbfsize
  type(oh_mycomm),intent(out) :: mycomm
  integer,intent(inout) :: nbor(3,3,3)
  integer,intent(in)    :: pcoord(OH_DIMENSION)
  integer,intent(inout) :: sdoms(:,:,:)
  integer,intent(in)    :: scoord(2,OH_DIMENSION)
  integer,intent(in)    :: nbound
  integer,intent(in)    :: bcond(2,OH_DIMENSION)
  integer,intent(inout) :: bounds(:,:,:)
  integer,intent(in)    :: ftypes(:,:)
  integer,intent(in)    :: cfields(:)
  integer,intent(in)    :: ctypes(:,:,:,:)
  integer,intent(out)   :: fsizes(:,:,:)
  integer,intent(out)   :: zbound(2,2)
  integer,intent(in)    :: stats
  integer,intent(in)    :: repiter
  integer,intent(in)    :: verbose
end subroutine
```

C Interface

```
void oh4s_init(int **sdid, const int nspec, const int maxfrac,
               const long long int npmax, const int minmargin,
               const int maxdensity, int **totalp, int **pbase,
               int *maxlocalp, int *cbufsize, void *mycomm, int **nbor,
               int *pcoord, int **sdoms, int *scoord, const int nbound,
               int *bcond, int **bounds, int *ftypes, int *cfields,
               int *ctypes, int **fsizes, int **zbound,
               const int stats, const int repiter, const int verbose);
```

`sdid`
`nspec`
: See §3.4.1 because two arguments above are perfectly equivalent to those of `oh1_init()`.

`maxfrac` is perfectly equivalent to that of `oh1_init()` and thus should have the tolerance factor percentage of load imbalance α greater than 0 and less than 100, as discussed in §3.4.1. This argument is used to calculate the *base value* of the particle buffer size $P'_{lim} =$ `maxlocalp`.

`npmax` should be the absolute maximum number of particles which your simulator is capable of as a whole. Unlike the level-2/3/4p libraries, this argument is given to `oh4s_init()` for the calcuation of $P'_{lim} =$ `maxlocalp` rather than to `oh2_max_local_particles()` or `oh4p_max_local_particles()`.

`minmargin` should be the minimum margin by which $P'_{lim} =$ `maxlocalp` has to clear over the per node average of `npmax`. Unlike the level-2/3/4p libraries, this argument is given to `oh4s_init()` for the calcuation of $P'_{lim} =$ `maxlocalp` rather than to `oh2_max_local_particles()` or `oh4p_max_local_particles()`.

`maxdensity` should be the maximum density $\mathcal{D}$ being the maximum particle population in a grid-voxel to be used for the calculation of $P'_{lim} =$ `maxlocalp`.

`totalp` is perfectly equivalent to that of `oh1_init()` shown in §3.4.1. Note that `nphgram` which the level-1 to level-3 counterparts have is not a member of the arguments of `oh4s_init()` because maintaining the per-subdomain histogram is perfectly up to the level-4s library functions, as in level-4p.

`pbase` is perfectly equivalent to that of `oh2_init()` shown in §3.5.2. Note that `pbuf` which the level-2/3/4p counterparts have is not a member of the arguments of `oh4s_init()` because the particle buffer should be allocated referring to $P'_{lim} =$ `maxlocalp` calculated by this function and then be given to the level-4s library through `oh4s_particle_buffer()`.

`maxlocalp` (for Fortran)
`*maxlocalp` (for C)
: The (variable pointed by this) argument will have the *base value* of the absolute limit of the particle buffer, P'_{lim}, given by the following.

$$\overline{P} = \lceil \texttt{npmax}/N \rceil$$
$$\delta_d^{\max} = \max_{0 \le m < N} \{\delta_d(m)\}$$
$$P_{halo} = \mathcal{D}((\delta_x^{\max}+2)(\delta_y^{\max}+2)(\delta_z^{\max}+2) - \delta_x^{\max}\delta_y^{\max}\delta_z^{\max})$$

$$P_{mgn} = \mathcal{D}\delta_x^{\max}\delta_y^{\max}$$
$$P'_{lim} = \max(\left\lceil \overline{P}(100+\alpha)/100 \right\rceil,\ \overline{P} + \texttt{minmargin}) + 2(P_{halo} + P_{mgn})$$

Note that P_{halo} represents the maximum number of halo particle in each of primary and secondary subcuboids, while P_{mgn} means we have to allow the excess of this amount from the particle population which OhHelp load balancer suggests for each of primary and secondary because particles in a xy-plane of subdomain is the unit of balancing. Also note that this argument `maxlocalp` is *output* one for `oh4s_init()` rather than input in level-2/3/4p counterparts.

`cbufsize` (for Fortran)
`*cbufsize` (for C)
: The (variable pointed by this) argument will have the size P_{comm} of send and receive buffers required by the halo-part communication of a particle-associated one-dimensional array by `oh4s_exchange_border_data()`, given by the following.

$$P_{comm} = 2\mathcal{D}\delta_z^{\max}\max(\delta_x^{\max}+2,\ \delta_y^{\max})$$

Note that $P_{comm} < 2P_{halo}$ because of the followings two reasons. First, elements in the bottom/top surfaces grid-voxels of a subcuboid are directly sent from the particle-associated array and those in just below/above the bottom/top surfaces are directly received into the array, without buffering. Second, the communication for *vertical* surfaces takes place in two phases, at first yz-surfaces (or west/east ones) and then xz-surfaces (or south/north ones) including their intersections of yz-surfaces, so that the buffers are not necessary to keep elements of both at the same time but are sufficient to accommodate larger one of them.

`mycomm`
`nbor`
`pcoord`
: See §3.4.1 because the arguments above are perfectly equivalent to those of `oh1_init()`.

`sdoms`
`scoord`
`nbound`
`bcond`
`bounds`
`ftypes`
`cfields`
`ctypes`
: See §3.6.1 because the arguments above are perfectly equivalent to those of `oh3_init()`.

`fsizes` is perfectly equivalent to that of `oh4p_init()` shown in §3.7.3. Therefore, `fsizes`$(\beta, d, F+1)$ or `fsizes`$[F][d][\beta]$ is for the per-grid histogram (and per-grid index) you must (or may) allocate.

`zbound(2,2)` (for Fortran)
`**zbound` (for C)
: The argument `zbound` should be an two-dimensional integer array of $(2, 2)$ in Fortran, while in C it should be a double pointer to an integer array of $[2 \times 2]$ or a

pointer to `NULL` (not `NULL` itself) to make `oh4s_init()` allocate the array for you and return the pointer to it through the argument. After a call of `oh4s_transbound()`, `zbound`(β+1, p+1) or `zbound`[p][b] ($p, \beta \in \{0, 1\}$) will have the local z-coordinate of the lower ($\beta = 0$) or upper ($\beta = 1$) surface of the primary ($p = 0$) or secondary ($p = 1$) subcuboid of the local node n, i.e., $\zeta_p^\beta(n)$.

`stats`
`repiter`
`verbose`
See §3.4.1 because the arguments above are perfectly equivalent to those of `oh1_init()`.

3.8.4 `oh4s_particle_buffer()`

The function (subroutine) `oh4s_particle_buffer()` is to let level-4s library functions know where the particle buffer is located in your simulator body, or to allocate the buffer for you. Unlike level-2/3/4p libraries, the particle buffer is not given to (or by) `oh4s_init()` because its mininum size P'_{lim} is calculated by `oh4s_init()` and is reported through its argument `maxlocalp`. Therefore, if your simulator is coded in Fortran, you must allocate the buffer for $2P'_{lim}$ or more particles and give the buffer to this function through `pbuf` argument, together with the real buffer size P_{lim} through the argument `maxlocalp` of this function. As for C coded simulators, you may allocate the buffer and give the double pointer to it, or let this function allocate the buffer of $2P_{lim} = 2 \times$ `maxlocalp` elements.

As in the level-4p library, the buffer `pbuf` is conceptually split into two portions of equal size P_{lim}. At the first call of `oh4s_transbound()`, the first half should have the particles which the node accommmodates at initial, and the second half will have the primary/secondary particles for the node in the next (usually first) simulation step. Then you will update velocities and positions of the particles in the second half and call `oh4s_transbound()` again to have the particles for the next step in the first half. This buffer switching continues alternating the role of first and second halves each time you call `oh4s_transbound()`.

Fortran Interface

```
integer subroutine oh4s_particle_buffer(maxlocalp, pbuf)
  use oh_type
  implicit none
  integer,intent(in)   :: maxlocalp
  type(oh_particle),intent(inout) :: pbuf(:)
end subroutine
```

C Interface

```
void oh4s_particle_buffer(const int maxlocalp, struct S_particle **pbuf);
```

`maxlocalp` should have the absolute limit of each portion of the particle buffer `pbuf` and thus defines P_{lim}. That is, the particle buffer `pbuf` should have (or will have) $2P_{lim}$ elements. The value of P_{lim} must not be less than P'_{lim} calculated by `oh4s_init()` and reported through its argument of the same name, or this function aborts the execution. On the other hand, you may (or must) specify $P_{lim} > P'_{lim}$ to ensure that each portion of the buffer can accommodate $P_{lim} - P'_{lim}$ particles to be injected, for example.

`pbuf(`P_{lim}`)` (for Fortran)
`**pbuf` (for C) The argument `pbuf` should be an one-dimensional array of `oh_particle` type structure and have $2P_{lim}$ elements in Fortran. As for C coded simulators, it should be a double pointer to an array of `S_particle` structure having $2P_{lim}$ elements, or a pointer to `NULL` (not `NULL` itself) to make `oh4s_particle_buffer()` allocate the buffer for you and return the pointer to it through the argument.

3.8.5 `oh4s_per_grid_histogram()`

The function (subroutine) `oh4s_per_grid_histogram()` is similar to its level-4p counterpart `oh4p_per_grid_histogram()`, and thus is to let level-4s library functions know where the array of per-grid histogram is located in your simulator body, or to allocate the array for you. However, this function has an additional argument `pgindex` for per-grid index whose location is also given to the library or which is allocated by this function.

Fortran Interface

```
subroutine oh4s_per_grid_histogram(pghgram, pgindex)
  implicit none
  integer,intent(inout) :: pghgram
  integer,intent(inout) :: pgindex
end subroutine
```

C Interface

```
void oh4s_per_grid_histogram(int **pghgram, int **pgindex);
```

`pghgram` (for Fortran)
`**pghgram` (for C)
The argument `pghgram` for Fortran should be the origin of $(D+2)$-dimensional array for the per-grid histogram, say `h(0,0,0,1,1)` for the particles of the first species in the grid-voxel at $(0,0,0)$ of the primary subdomain. In C, it should be a double pointer to such an array element, say `&&h[0][0][0][0][0]`, or a pointer to `NULL` (not `NULL` itself) if you want the library to allocate the array and return the pointer to its origin element through the argument. Note that if you give the origin element to the function, the array must have the shape $\phi_x \times \phi_y \times \phi_z \times S \times 2$ where $\phi_d = \phi_d^u - \phi_d^l$ and $\phi_d^\beta =$ `fsizes`$(\beta, d, F{+}1)$ or $\phi_d^\beta =$ `fsizes`$[F][d][\beta]$ obtained through the `fsizes` argument of `oh4s_init()`.

`pgindex` (for Fortran)
`**pgindex` (for C)
The argument `pgindex` for Fortran should be the origin of $(D+2)$-dimensional array for the per-grid index, say `i(0,0,0,1,1)` for the particles of the first species in the grid-voxel at $(0,0,0)$ of the primary subdomain. In C, it should be a double pointer to such an array element, say `&&i[0][0][0][0][0]`, or a pointer to `NULL` (not `NULL` itself) if you want the library to allocate the array and return the pointer to its origin element through the argument. The shape of the array must be same as that specified for `pghgram` if the origin element is given to this function.

Note that `oh4s_transbound()` for Fortran will let each element of the per-grid index array `i`(x, y, z, s, c) have the *one-origin* index of the first primary $(c = 1)$ or secondary $(c = 2)$

particle of species s ($\in [1, S]$) in the grid-voxel at (x, y, z) in a half portion of the particle buffer, if grid-voxel has one or more particles, i.e. `h`$(x, y, z, s, c) > 0$. For C coded simulators, `oh4s_transbound()` will let `i`$[c][s][z][y][x]$ have the *zero-origin* index of the first primary ($c = 0$) or secondary ($c = 1$) particle of species s ($\in [0, S)$) in the grid-voxel at (x, y, z) in a half portion of the particle buffer, if `h`$[c][s][z][y][x] > 0$. On the other hand, if a grid-voxel has no particles, the corresponding element of the per-grid index array will have the index of the first particle in the *next* non-empty grid-voxel, or the index next to the last particle if there are no non-empty grid-voxels following the corresponging grid-voxel. Therefore, `i(-1,-1,-1,1,1)` $= 1$ for Fortran and `i[0][0][-1][-1][-1]` $= 0$ for C, always.

3.8.6 `oh4s_transbound()`

The function `oh4s_transbound()` at first performs operetaions for load balancing as same as that `oh1_transbound()` does; examination of the per-subdomain particle population histogram to check the balancing and (re)building of helpand-helper configuration if necessary. Then for each grid-voxel set sharing a z-coordinate value, it determines the node to accommodate particles in the set to assign primary/secondary subcuboids to each node. After that particles in the first/second half of the particle buffer, `pbuf` argument of `oh4s_particle_buffer()`, are transferred to satisfy load balancing, position-awareness and the accommodation of halo particles. Finally particles in each node are sorted according to the coordinates of grid-voxels in which they reside and the per-grid histogram and per-grid index in the node presented to `oh4s_per_grid_histogram()` are updated to show the number of particles in each grid-voxel and the `pbuf`'s index of the first particle in it. The sorted result is stored in the second/first half of `pbuf`.

Since the arguments of `oh4s_transbound()` and its return value are perfectly equivalent to those of `oh1_transbound()` (and its level-2/3/4p counterparts), see §3.4.4 for their definitions.

Fortran Interface

```
integer function oh4s_transbound(currmode, stats)
  implicit none
  integer,intent(in) :: currmode
  integer,intent(in) :: stats
end function
```

C Interface

```
int oh4s_transbound(const int currmode, const int stats);
```

3.8.7 `oh4s_exchange_border_data()`

The function (subroutine) `oh4s_exchange_border_data()` performs the inter-node communication for a particle-associated one-dimensional array so that its part corresponding to halo particles in each node has the value computed by other nodes responsible of the particles. In addition to the array `buf` of P_{lim} (or more) elements, the function needs to be given a send buffer `sbuf` and a receive buffer `rbuf` whose sizes are commonly P_{comm} (or more) reported through the argument `cbufsize` of `oh4s_init()`.

Fortran Interface

```
subroutine oh4s_exchange_border_data(buf, sbuf, rbuf, type)
  implicit none
  real*8,intent(inout) :: buf
  real*8,intent(out)   :: sbuf
  real*8,intent(out)   :: rbuf
  integer,intent(in)   :: type
end subroutine
```

C Interface

```
void oh4s_exchange_border_data(void *buf, void *sbuf, void *rbuf,
                               MPI_Datatype type);
```

`buf`[17] should be (the pointer to) the first element of the particle-associated array of P_{lim} (or more) elements whose halo part will have values computed by other nodes.

`sbuf` should be (the pointer to) the first element of an one-dimensional array of P_{comm} (or more) elements to be used as send buffer in the function.

`rbuf` should be (the pointer to) the first element of an one-dimensional array of P_{comm} (or more) elements to be used as receive buffer in the function.

`type` should have the MPI data-type of elements of the particle-associated array.

3.8.8 `oh4s_map_particle_to_neighbor()`

The function `oh4s_map_particle_to_neighbor()` is perfectly equivalent to `oh4p_map_particle_to_neighbor()` discussed in §3.7.7. It is also as same as the level-4p counterpart that you have to call `oh4s_map_particle_to_neighbor()` or `oh4s_map_particle_to_subdomain()` for *all* particles which the local node is responsible of, i.e., those residing in the primary/secondary subcuboids, for histogram maintenance by the library. This "all particles resoponsible of", however, does *not* means "all particles in the particle buffer" because the buffer has halo particles which other nodes are responsible of. In fact, the `nid` elements of halo particles are set to be negative by `oh4s_transbound()` when it received other nodes because they should be *eliminated* in the next call of `oh4s_transbound()`[18], and thus applying the mapping function on a halo particle should make erroneous duplication of it, i.e., one on the local node and the other on the node responsible of it.

Fortran Interface

```
integer function oh4s_map_particle_to_neighbor(part, ps, s)
  use oh_type
  implicit none
  type(oh_particle),intent(inout) :: part
  integer,intent(in)   :: ps
  integer,intent(in)   :: s
end function
```

[17] In the Fortran module file oh_mod4s.F90, the arguments `buf`, `sbuf` and `rbuf` of are declared as `real*8` type hoping it matches the type of the elements in your array. If this is incorrect, feel free to modify the declaration or to remove it, so that your compiler accept your calls of the library subroutines.

[18] Though a halo particle at a simulation step can be (or is likely) accommodated by the node as a halo or ordinary particle, it cannot stay in the node but has to travel from the node responsible of it.

C Interface

```
int  oh4s_map_particle_to_neighbor(struct S_particle *part, const int ps,
                                   const int s);
```

3.8.9 oh4s_map_particle_to_subdomain()

The function oh4s_map_particle_to_subdomain() is perfectly equivalent to oh4p_map_particle_to_subdomain() discussed in §3.7.8, and the caution about halo particles given in §3.8.8 is also applicable to this function.

Fortran Interface

```
integer function oh4s_map_particle_to_subdomain(part, ps, s)
  use oh_type
  implicit none
  type(oh_particle),intent(inout) :: part
  integer,intent(in)   :: ps
  integer,intent(in)   :: s
end function
```

C Interface

```
int  oh4s_map_particle_to_subdomain(struct S_particle *part, const int ps,
                                    const int s);
```

3.8.10 oh4s_inject_particle()

The function oh4s_inject_particle() is perfectly equivalent to oh4p_inject_particle() discussed in §3.7.9.

Fortran Interface

```
integer function oh4s_inject_particle(part, ps)
  use oh_type
  implicit none
  type(oh_particle),intent(inout) :: part
  integer,intent(in)   :: ps
end function
```

C Interface

```
int  oh4s_inject_particle(const struct S_particle *part, const int ps);
```

3.8.11 oh4s_remove_mapped_particle()

The function (subroutine) oh4s_remove_mapped_particle() is perfectly equivalent to oh4p_remove_mapped_particle() discussed in §3.7.10.

Fortran Interface

```
subroutine oh4s_remove_mapped_particle(part, ps, s)
  use oh_type
  implicit none
  type(oh_particle),intent(inout) :: part
  integer,intent(in)   :: ps
  integer,intent(in)   :: s
end subroutine
```

C Interface

```
void oh4s_remove_mapped_particle(struct S_particle *part, const int ps,
                                 const int s);
```

3.8.12 oh4s_remap_particle_to_neighbor()

The function oh4s_remap_particle_to_neighbor() is perfectly equivalent to oh4p_remap_particle_to_neighbor() discussed in §3.7.11.

Fortran Interface

```
integer function oh4s_remap_particle_to_neighbor(part, ps, s)
  use oh_type
  implicit none
  type(oh_particle),intent(inout) :: part
  integer,intent(in)   :: ps
  integer,intent(in)   :: s
end function
```

C Interface

```
int  oh4s_remap_particle_to_neighbor(struct S_particle *part, const int ps,
                                     const int s);
```

3.8.13 oh4s_remap_particle_to_subdomain()

The function oh4s_remap_particle_to_subdomain() is perfectly equivalent to oh4p_remap_particle_to_subdomain() discussed in §3.7.12.

Fortran Interface

```
integer function oh4s_remap_particle_to_subdomain(part, ps, s)
  use oh_type
  implicit none
  type(oh_particle),intent(inout) :: part
  integer,intent(in)   :: ps
  integer,intent(in)   :: s
end function
```

C Interface

```
int  oh4p_remap_particle_to_subdomain(struct S_particle *part, const int ps,
                                      const int s);
```

3.9 Particle Injection and Removal

As discussed in §3.5.5, level-2 library provides you with a function (subroutine) `oh2_inject_particle()` to inject a particle dynamically. The level-4p extended library also has its own version of the injection function `oh4p_inject_particle()` as shown in §3.7.9. This section revisits this issue and also discusses its counterpart, particle removal.

3.9.1 Level-1 Injection and Removal

If you use level-1 library only, what you need to do on injecting and/or removing particles is to maintain `nphgram` correctly as far as the library concerns. Since the function `oh1_transbound()` will not be surprised at a sudden apparition of a particle into any subdomain and any node, you may freely increase an element of `nphgram` to notify the library of the particle injection[19]. This unusual increase of `nphgram` elements, however, may cost if particles are injected into a node which is not responsible for the subdomain to which the particles have appeared or for that adjoining the subdomain. That is, `oh1_transbound()` needs some global communications to make the particle transfer schedule, which are unnecessary on usual boundary crossing transfers. On the other hand, decreasing elements of `nphgram` to remove particles[20] is no problem in terms of both logical correctness and performance of `oh1_transbound()`.

An important caution on the play with `nphgram` is that `oh1_transbound()` is only aware of the load balancing of particles whose populations in subdmains are reported in `nphgram`, of course. This means that if you have a *stock* of inactive particles in your particle buffer from which you pick particles to be injected and into which you fling removed particles, your buffer could overflow because `oh1_transbound()` does not know anything about the stock. Therefore, the stock should be sufficiently small, say up to some hundred thousands. Note that particle *recycling* without stock, i.e., injecting a particle only when another particle is removed by overwriting particle data, should cause no problem.

A way to avoid the overflow of the stock, especially when the stock is significantly large, is to include the number of particles in the stock into `nphgram` making them pretend to reside in a subdomain. This works well with respect to the balancing of required memory space but might cause severe imbalance of computation, because `oh1_transbound()` does not know that particles in the stock are *inactive*. Moreover, since `oh1_transbound()` may decide to throw particles in the stock away to other nodes, the node could find it has no particles to recycle in the stock on injection.

3.9.2 Level-2 (and 3) Injection and Removal

On the other hand, an injection by `oh2_inject_particle()` is not only as easy as just increasing `nphgram` but also consistent with other library functions especially with `oh2_transbound()` (and thus `oh3_transbound()` usually), which recognizes the particle, the subdomain into which it is injected, and the memory location at which it is stored. That is, `oh2_transbound()` automatically picks injected particles from the bottom of `pbuf` and places them into appropriate position in `pbuf` or transfers them to appropriate nodes which are responsible for the subdomains they reside. What you need to take care of is that you have to reserve some space (not a stock) in `pbuf` large enough to inject particles in a simulation time step. If the space is too large for a node due to a significantly large number of potential injections, you can limit the space to a reasonable size and let the node having

[19] Unless the total of `nphgram` reaches or exceeds $2^{31} - 1$.

[20] Or skipping the increment of `nphgram` element for the particle to be removed.

too many particles to be injected push overflown ones to other nodes. A simple solution to do it is to repeat `oh2_transbound()` and an all-reduce communication to confirm the completion of all particle injections, because it is assured that the space for injection is emptied each time `oh2_transbound()` is executed.

Particle removal can be implemented more easily with level-2 or level-3 library. What you need to do is to set `nid` element of the particle in problem to be -1, excluding it from counting particles for `nphgram`. Then `oh2_transbound()` will remove the particles reclaiming the space for them. However, if you want to remove an injected particle before the call of `oh2_transbound()` following the injection by your own special reason, you have to call `oh2_remove_injected_particle()` passing the particle in the reserved space into which the injected particle is stored by `oh2_inject_particle()`, *not* decrementing `nphgram` by yourself but delegating it to the function. This caution is based on that the library internally maintains information about injected particles so that `oh2_transbound()` properly handle them and thus you have to tell the library that the particle once injected is removed.

Similarly, if you want to *move* a particle after its injection and before the call of `oh2_transbound()`, you have to remove it by `oh2_remove_injected_particle()` and then call `oh2_remap_injected_particle()` after setting the structure elements of the particle especially `nid` and `spec`. If you are (almost) sure that injceted particles will move afterward, however, you can omit the call of `oh2_remove_injected_particle()` by giving the particle having negative `nid` when you call `oh2_inject_particle()`. Note that the maintenance of `nphgram` should be delegated to `oh2_remap_injected_particle()` as in the case of `oh2_inject_particle()` and `oh2_remove_injected_particle()`.

Another caution of the injection by `oh2_inject_particle()` and the removal by setting `nid` to -1 is that these operations are expected to be performed *after* the first call of `oh2_transbound()` or `oh3_transbound()` which takes care of initial particle distribution. Therefore, if by some reason your simulator code needs to inject/remove particles into/from initial setting of particles *before* the first call of `oh2_transbound()` or `oh3_transbound()`, you need to call `oh2_set_total_particles()` *before* the injection/removal setting `nphgram` correctly, as discussed in §3.5.8.

3.9.3 Level-4p and 4s Injection and Removal

The discussion above for level-2 (and level-3) injection/removal also holds for level-4p extension with its own injection function `oh4p_inject_particle()`, as far as you are fully aware of particle histograms maintained by this function and mapping functions `oh4p_map_particle_to_neighbor()` and `oh4p_map_particle_to_subdomain()`. That is, if the injected particle stays at the position, where you specified when you call `oh4p_inject_particle()`, until you call `oh4p_transbound()`, per-subdomain and per-grid histograms are properly passed to `oh4p_transbound()`. Similarly, if you set the `nid` element of a partilce to -1 without calling `oh4p_map_particle_to_neighbor()` nor `oh4p_map_particle_to_subdomain()`, the particle will be safely eliminated by `oh4p_transbound()`.

However, the injection/removal logic of your simulation code could violate the rule above. For example, you might wish to move a particle *after* injecting it and *before* the call of `oh4p_transbound()` to cause a trouble because `oh4p_transbound()` cannot recognize the motion. Calling a mapping function at moving cannot solve the problem because it simply causes double counting for its original and new positions. A simple solution is to call `oh4p_remove_mapped_particle()` to eliminate the particle in problem temporarily and then `oh4p_map_particle_to_neighbor()` or `oh4p_map_particle_to_subdomain()`, or the combined function `oh4p_remap_particle_to_neighbor()` or `oh4p_`

`remap_particle_to_subdomain()`. If it is troublesome due to, for example, the necessity of special care for injected particles in your particle pushing procedure, you may use `oh2_inject_particle()` instead of level-4p's `oh4p_inject_particle()` for the injection and then call `oh4p_map_particle_to_neighbor()` or `oh4p_map_particle_to_subdomain()`. One caution for this second solution is that you have to set `nid` element of the particle to -1 when you call `oh2_inject_particle()` so that the function excludes the particle from the per-subdomain histogram.

As for the removal, you have to call `oh4p_remove_mapped_particle()` if and only if the particle to be removed is mapped by a level-4p's mapping function or injected by `oh4p_inject_particle()` after the last call of `oh4p_transbound()`. For example, you might move a particle and then call a mapping function for it before you find the particle should be eliminated due to, e.g., ion-electron recombination. You can cope with this complication by calling `oh4p_remove_mapped_particle()` for the electron recombined with the ion. Note that if this recombination changes the species of the ion due to the discharge, you also have to call `oh4p_remove_mapped_particle()` for the ion and then inject it by `oh4p_inject_particle()` specifying the new species. Also note that the eliminating a particle which a mapping function detected going out-of-bounds returning -1 to the caller does not require to call `oh4p_remove_mapped_particle()`, though doing it is not harmful logically.

Finally, the discussion of level-4p injection and removal above perfectly holds for the level-4s extension and its functions `oh4s_inject_particle()`, `oh4s_map_particle_to_neighbor()`, `oh4s_map_particle_to_subdomain()`, `oh4s_remove_mapped_particle()`, `oh4s_remap_particle_to_neighbor()`, `oh4s_remap_particle_to_subdomain()` and `oh4s_transbound()`.

3.9.4 Identification of Injected Particles

The last issue on particle injection and removal is the identification of particles. In the default definition of the Fortran structured type `oh_particle` and C `struct` named `S_particle`, each particle has its identifier in `pid` element. Since this element is a 64-bit integer, the space for the identification number is large enough for local numbering without reclamation. For example, a node n may give a number $kN + n$ to the k-th particle created by the node. Since 2^{64} should be much larger than N, the identification space is hardly exhosted. For example, even if $N = 2^{20}$ and each node injects (and removes) 2^{20} particles in each simulation step in addition to its initial accommodation of 2^{30} particles, it will take about 16.8 million time steps or, even if your simulator has an excellent per-node performance of 10 million particles per second[21], 1.68 billion seconds or 53 *years*.

3.10 Statistics

Level-1 library provides you with the functions to collect, process and report two types of statistics data of timings and particle transfers. The timing statistics data is obtained by measuring the execution time of intervals in your program including the library functions. Since each interval is identified by a *key* being a non-negative unique integer, you have to define the set of keys for the intervals which you want to measure together with strings printed on the report, by modifying the C header file `oh_stats.h` as discussed in §3.10.1. Then, after calling `oh1_init()`, or one of its higher level counterparts `oh2_init()` and `oh3_init()`, giving it fundamental parameters for statistics as discussed in §3.10.2, you may call the following functions (subroutines) to collect, process and report statistics data as explained in §3.10.3, 3.10.4, 3.10.5 and 3.10.6.

[21] The per-node perormance of our simulator reported in [1] is 2.55 million particle per second.

`oh1_init_stats()` initializes internal data structures for statistics and starts the execution time measurement of the first interval.

`oh1_stats_time()` finishes the execution time measurement of the last interval, and starts that of the next interval.

`oh1_show_stats()` gathers timing and particle transfer statistics data measured in a simulation step and, if specified, reports a subtotal for recent steps.

`oh1_print_stats()` reports the grand total of statistics data.

3.10.1 Timing Statistics Keys and Header File oh_stats.h

You can measure the execution time of an interval in your program by calling `oh1_stats_time()` giving it a key to identify the interval. Since the key, a non-negative integer, should be unique to the interval and should be associated to a character string printed on the report together with the statistics of the measured timing, the library provides you with a C header file named oh_stats.h, which can be included from Fortran codes too, to assure the uniqueness and the association with the string easily.

The file consists of two parts and the default definition given by the first part is as follows.

```
#define STATS_TRANSBOUND  0
#define STATS_TRY_STABLE  (STATS_TRANSBOUND + 1)
#define STATS_REBALANCE   (STATS_TRY_STABLE + 1)
#define STATS_REB_COMM    (STATS_REBALANCE + 1)
#define STATS_TB_MOVE     (STATS_REB_COMM + 1)
#ifdef  OH_LIB_LEVEL_4PS
#define STATS_TB_SORT     (STATS_TB_MOVE + 1)
#define STATS_TB_COMM     (STATS_TB_SORT + 1)
#else
#define STATS_TB_COMM     (STATS_TB_MOVE + 1)
#endif
#define STATS_TIMINGS     (STATS_TB_COMM + 1)
```

The code above `#define`'s the following six (or seven if you activate level-4p/4s extension) keys to measure the execution times in `oh1_transbound()` and/or its higher level counterparts and a special key `STATS_TIMINGS` to have the number of keys.

`STATS_TRANSBOUND` is for the interval to examine if the execution mode in the next step is primary.

`STATS_TRY_STABLE` is for the interval to examine if the helpand-helper configuration can be kept in the next step.

`STATS_REBALANCE` is for the interval to (re)build a new helpand-helper configuration.

`STATS_REB_COMM` is for the interval to create family communicators for the newly built helpand-helper configuration.

`STATS_TB_MOVE` is for the interval in `oh2_transbound()` or `oh3_transbound()` to move particles in `pbuf`.

`STATS_TB_SORT` is for the interval in `oh4p_transbound()` for sorting particles by their position.

`STATS_TB_COMM` is for the interval in `oh2_transbound()` or `oh3_transbound()` to transfer particles among nodes.

On adding your own keys, it is recommended to follow the convention shown in the file. That is, defining a key by;

`#define` ⟨*new key*⟩ (⟨*last key*⟩`+1`)

will assure the uniqueness and continuity of keys. For example, to add three keys namely `STATS_PARTICLE_PUSHING`, `STATS_CURRENT_SCATTERING` and `STATS_FIELD_SOLVING` for the intervals of particle pushing, current scattering and field solving in your main loop, replacing the first line for `STATS_TRANSBOUND` with the followings is safe and correct.

```
#define STATS_PARTICLE_PUSHING   0
#define STATS_CURRENT_SCATTERING (STATS_PARTICLE_PUSHING + 1)
#define STATS_FIELD_SOLVING      (STATS_CURRENT_SCATTERING + 1)
#define STATS_TRANSBOUND         (STATS_FILED_SOLVING + 1)
```

Note that you must not remove any definitions given in the original oh_stats.h, or you cannot compile the library correctly.

The second part of the file defines the character strings for keys as follows.

```
#ifdef OH_DEFINE_STATS
static char *StatsTimeStrings[2*STATS_TIMINGS] = {
  "transbound",          "",
  "try_stable",          "",
  "rebalance",           "",
  "reb comm create",     "",
  "part move[pri]",      "part move[sec]",
#ifdef  OH_LIB_LEVEL_4PS
  "part sort[pri]",      "part sort[sec]",
#endif
  "part comm[pri]",      "part comm[sec]",
};
#endif
```

In the code above, `#ifdef`/`#endif` construct is to protect your code from erroneous compilation especially if your code is in Fortran. That is, `OH_DEFINE_STATS` is defined only in the library source files and thus the compiler for your own codes will skip the part which cannot be parsed as a Fortran code.

The important part in the code is the sequence of the string pairs, one pair for each line. The pairs correspond to keys in the same order and each pair gives a short explanation of the pair of intervals, one for primary particles/subdomains and the other for secondary ones, identified by the corresponding key. That is, if your interval is executed twice as a *primary execution* and a *secondary execution*, the first and second strings are used as the titles of two executions separately. Otherwise, or if you measure two executions as a whole, defining the first string and letting the second be empty string are necessary and sufficient.

For example, adding the following three lines just before the line having `"transbound"` is what you need to do for the three keys exemplified above, providing you want to measure primary and secondary executions of each interval separately.

```
"particle pushing[pri]",   "particle pushing[sec]",
"current scattering[pri]", "current scattering[sec]",
"field solving[pri]",      "field solving[sec]",
```

Remember that a title can be arbitrarily long but that of 30 characters or longer will cause an ungly line in the report.

3.10.2 Arguments of `oh1_init()` for Statistics

As shown in §3.4.1, the function (subroutine) `oh1_init()` and its higher level counterparts have the following two arguments to control statistics operations.

`stats` activates or inactivates statistics operations as follows.

- If `stats` = 0, statistics operations are inactivated and thus the functions discussed in the following sections do nothing.
- if `stats` = 1[22], statistics operations are activated but only the grand total is reported by `oh1_print_stats()`.
- if `stats` = 2, statistics operations are activated and `oh1_show_stats()` will report subtotal when it is given the simulation step count divisible by the argument `repiter` of `oh1_init()`.

Note that `oh1_transbound()` and its higher level counterparts also have an argument `stats` to control the statistics collection in the function temporarily overriding what `stats` of `oh1_init()` specifies. That is, statistics collection in `oh1_transbound()` is inactivated if its `stats` is 0 regardless `stats` of `oh1_init()`, while non-zero means that statistics collection follows what `stats` of `oh1_init()` specifies. This feature is useful to exclude statistics data in, for example, initialization process.

`repiter` defines the frequency of subtotal reporting by `oh1_show_stats()`. That is, if `stats` = 2, it defines the gap of periodical reporting by `oh1_show_stats()`.

3.10.3 `oh1_init_stats()`

The function (subroutine) `oh1_init_stats()` initializes internal data structures for statistics, and starts first interval of timing measurement, if `stats` of `oh1_init()` is not zero. The other statistics function must be called after `oh1_init_stats()` is called.

Fortran Interface

```
subroutine oh1_init_stats(key, ps)
  implicit none
  integer,intent(in) :: key
  integer,intent(in) :: ps
end subroutine
```

C Interface

```
void oh1_init_stats(int key, int ps);
```

`key` is the key of the first interval whose execution time is measured. If you do not want to include the first interval in the timing statistics, give this argment the special key `STATS_TIMINGS`.

`ps` indicates whether the first interval is for primary execution (0) or secondary execution(1).

[22] Or some other value excluding 0 and 2.

3.10.4 oh1_stats_time()

The function (subroutine) oh1_stats_time() finishes the last interval of timinig measurement and starts next one, if stats of oh1_init() is not zero.

Fortran Interface

```
subroutine oh1_stats_time(key, ps)
  implicit none
  integer,intent(in) :: key
  integer,intent(in) :: ps
end subroutine
```

C Interface

```
void oh1_stats_time(int key, int ps);
```

key is the key of the interval to start for execution time measurement. If you want only to finish the last interval, give this argment the special key STATS_TIMINGS.

ps indicates whether the next interval is for primary execution (0) or secondary execution (1).

3.10.5 oh1_show_stats()

The function (subroutine) oh1_show_stats() performs the following statistics operations if stats of oh1_init() non-zero.

- Finish the last interval of timing measurement.
- Gather statistics data measured since the last call of this function or the call of oh1_init_stats().
- Update grand total statistics and, if stats of oh1_init() is 2, subtoal statistics.
- Print subtotal statistics as oh1_print_stats() does, if stats of oh1_init() is 2 and step argument of this function is divisible by repiter of oh1_init().
- Start a new interval whose execution time is excluded from timing statistics.

It is expected to call this function every simulation step so that it collect statistics data for each step.

Fortran Interface

```
subroutine oh1_show_stats(step, currmode)
  implicit none
  integer,intent(in) :: step
  integer,intent(in) :: currmode
end subroutine
```

C Interface

```
void oh1_show_stats(int step, int currmode);
```

`step` is the simulation step count to control periodical statistics reporting. If `stats` of `oh1_init()` is 2 and `step` is divisible by `repiter` of `oh1_init()`, subtotal statistics is reported.

`currmode` indicates whether the current execution mode is primary (0) or secondary (1). This value should be corresponding to the return value of the last call of `oh1_transbound()` or its higher level counterparts.

3.10.6 `oh1_print_stats()`

The function (subroutine) `oh1_print_stats()` report the grand total (so far) of statistics through standard output in the following format. The first part of the report is for execution time of each interval as follows.

```
## Execution Times (sec)
  particle pushing[pri]        =    0.024 /    2.297 /    1.015 / 1824925.604
  particle pushing[sec]        =    0.077 /    2.440 /    1.564 / 2135827.627
  current scattering[pri]      =    0.011 /    1.223 /    0.422 /  736536.722
  current scattering[sec]      =    0.032 /    1.332 /    0.836 / 1296109.407
  field solving[pri]           =    0.003 /    0.089 /    0.011 /   27344.603
  field solving[sec]           =    0.003 /    0.053 /    0.012 /   19633.007
  transbound                   =    0.004 /    0.837 /    0.222 /  364201.720
                               = -------- / -------- / -------- / -----------
  try_stable                   =    0.001 /    0.025 /    0.002 /    2366.882
                               = -------- / -------- / -------- / -----------
  rebalance                    =    0.001 /    0.002 /    0.001 /      21.432
                               = -------- / -------- / -------- / -----------
  reb comm create              =    0.023 /    2.569 /    0.740 /    4358.333
                               = -------- / -------- / -------- / -----------
  part move[pri]               =    0.021 /    1.668 /    0.528 /  283491.149
  part move[sec]               =    0.014 /    1.772 /    0.541 /  606886.809
  part comm[pri]               =    0.001 /    1.077 /    0.025 /   16184.129
  part comm[sec]               =    0.002 /    1.677 /    0.023 /   21244.773
```

Each column of the table above shows the followings of each interval.

- Column-1: titie of the interval.
- Column-2: minimum execution time of the interval.
- Column-3: maximum execution time of the interval.
- Column-4: average execution time of the interval.
- Column-5: sum of execution times of the interval.

Note that the minimum, maximum, average and sum are over all occasions of each interval in all nodes and all simulation time steps.

Then the second part reports the statistics of particle transfer as follows.

```
## Particle Movements
  p2p transfer[pri,min]        =      235 /  4272367 /     7368 /    47153707
  p2p transfer[pri,max]        =     1891 /  8054375 /    14909 /    95416324
  p2p transfer[pri,ave]        =      441 /  6194818 /    12796 /    81894514
  get[pri,min]                 =        0 /      589 /        3 /       19210
  get[pri,max]                 =     6511 /  8054765 /    22971 /   147011962
  put[pri,min]                 =        0 /      984 /        5 /       29490
  put[pri,max]                 =     6209 /  8054375 /    16387 /   104877429
  put&get[pri,ave]             =       13 /    31464 /       90 /      574318
  p2p transfer[sec,min]        =        1 /      656 /        2 /       10488
  p2p transfer[sec,max]        =     2198 /  6034178 /    22907 /   146602986
  p2p transfer[sec,ave]        =       31 /  1393581 /     2748 /    17587875
  get[sec,min]                 =        0 /      289 /        2 /       10021
  get[sec,max]                 =     3577 /  8387296 /    51544 /   329883298
  put[sec,min]                 =        0 /     1476 /        4 /       24108
  put[sec,max]                 =     3809 /  9732486 /    47848 /   306224944
  put&get[sec,ave]             =      118 /  1812744 /     3473 /    22225683
  transition to pri            =     1594 /        0 /        1 /        1595
  transition to sec            =        1 /     4782 /       22 /        4805
```

The rows above except for the last two are for the following particle transfers which are scheduled in one execution of `oh1_transbound()` or are actually performed in one execution of `oh2_transbound()` or `oh3_transbound()`.

`p2p transfer[]` shows the number of transferred particles between a pair of nodes. The minimum, maximum and average are calculated over all pairs such that at least one particle is transferred between each node pair.

`get[]` shows the number of particles a node received. The minimum and maximum are calculated over all nodes including those received nothing.

`put[]` shows the number of particles a node sent. The minimum and maximum are calculated over all nodes including those sent nothing.

`put&get[]` shows the average number of particles a node received (or sent). The average are calculated over all nodes including those received nothing.

Note that the categorization of primary (`pri`) and secondary (`sec`) particles is based on the viewpoint of receivers. Also note that the columns from Column-2 to Column-5 of these rows are for the minimum, maximum, average and sum which are calculated over all simulation time steps.

On the other hand, the last two raws show number of transitions to primary and secondary modes. In these rows, Column-2 and Column-3 are for the number of transitions from primary and secondary modes respectively. Column-4 of the transition to primary is the number of primary to primary transition at which non-neighboring particle transfers are taken, while that of to secondary means the number of secondary to secondary with rebuilding of helpand-helper configuration. Finally Column-5 of both rows is the total number of transition to primary or secondary mode.

The function `oh1_show_stats()` also reports the statistics if `stats` and `repiter` of `oh1_init()` and `step` argument of the function satisfy the reporting condition, but the numbers shown in columns of the minimum and others are calculated over the recent `repiter` steps.

Fortran Interface

```
subroutine oh1_print_stats(nstep)
  implicit none
  integer,intent(in) :: nstep
end subroutine
```

C Interface

```
void oh1_print_stats(int nstep);
```

`nstep` is the total simulation step count to calculate the average numbers in Column-4.

3.11 Verbose Messaging

Although the application of the OhHelp library to your PIC simulator is fairly simple and straightforward, it should be hard to compose a bug-free program instantly. Therefore, you will want to investigate what is going on in your program including the functions in the library when you encounter a problem.

Verbose messaging provided by the library is a fundamental mean for the investigation. You can activate or inactivate the verbose messaging in library functions by giving one of the followings to the argument `verbose` of `oh1_init()` or its higher level counterparts.

- `verbose` = 0 inactivates verbose messaging and thus makes library functions execute silently.
- `verbose` = 1 activates verbose messaging to have fundamental reports from library functions.
- `verbose` = 2 activates more verbose messaging than the case of 1 to capture some details of the events happening in library functions.
- `verbose` = 3 is similar to 2 but you will have messages from all nodes with their identifier (MPI rank).

If activated, messages are printed to standard output with a common header "`*Starting`" optionally followed by a node identifier surrounded by brackets.

In addition, you may have your own verbose messaging to be controlled by `verbose` of `oh1_init()` by calling the following function `oh1_verbose()`.

Fortran Interface

```
subroutine oh1_verbose(message)
  implicit none
  character(*),intent(in) :: message
end subroutine
```

C Interface

```
void oh1_verbose(char *message);
```

`message` is a character string to be printed following the header. Since it should be null-ternminated, you have to remember that a Fortran string constant, say `'hello'` does not have the terminator and thus you have to explicitly give a null code by `'hello\0'`.

Note that your message is assumed fundamental and thus will be printed if `verbose` is 1 or larger. Also note that `oh1_verbose()` has `MPI_Barrier()` in it and thus it should be called from all nodes to avoid deadlock. For example;

```
if (sdid(2).ge.0) then
  oh1_verbose('secondary particle push\0')
  call particle_push(...)
end if
```

will cause deadlock because the root node of helpand-helper tree will not call `oh1_verbose()` while others do. Therefore, the code above should be modified as follows.

```
if (currmode.ne.0)
  oh1_verbatim('secondary particle push\0')
  if (sdid(2).ge.0) call particle_push(...)
end if
```

3.12 Aliases of Functions

As shown in previous sections, all library functions have one of prefixes '`oh1_`', '`oh2_`', '`oh3_`', '`oh4p`' or '`oh4s_`' to show the library layer they belong to. Although this naming makes it clear that in order to use a function, say `oh2_inject_particle()`, you have to incorporate level-2 or level-3 library, it will be tiresome to remember the layer number which a function belongs to especially when you use (almost) everything provided by the layer you chose and by lower ones.

Therefore, the library has special header files ohhelp_f.h for Fortran and ohhelp_c.h for C to give API function aliases which just have a common prefix '`oh_`'. To use these files, you have to `#define` a constant `OH_LIB_LEVEL` as the number of the layer you choose, i.e., 1, 2, 3 or 4 in your source files, or have to edit the lines defining that in oh_config.h. Then you have the aliases shown in Table 1 according to the layer number you chose.

Note that both header files `#include`'s the header file oh_config.h, and ohhelp_c.h does the followings in addition to aliasing.

- `#include` the standard MPI header file mpi.h.
- Declares prototypes of library functions in use according to the layer you chose.
- Define `struct` named `S_mycommc`.
- `#include` the header file oh_part.h to define `struct` named `S_particle` if you choose level-2 or higher.

Also note that the function `oh13_init()` does not have any aliases.

3.13 Sample Code

This section gives examples of application of the level-3 OhHelp library to tiny 3-dimensional PIC simulators coded in Fortran and C. The main loop of these codes consists of calls of the following subroutines/functions, besides libraray fucntions.

`particle_push()` does what its name implies. The acceleration vector of each particle is calculated by a subroutine/function named `lorentz()` whose code is outside the scope of this document.

Table 1: Aliases of Library Functions

layer	alias	autonym
any	oh_neighbors()	oh1_neighbors()
	oh_families()	oh1_families()
	oh_acc_mode()	oh1_acc_mode()
	oh_broadcast()	oh1_broadcast()
	oh_all_reduce()	oh1_all_reduce()
	oh_reduce()	oh1_reduce()
	oh_init_stats()	oh1_init_stats()
	oh_stats_time()	oh1_stats_time()
	oh_show_stats()	oh1_show_stats()
	oh_print_stats()	oh1_print_stats()
	oh_verbose()	oh1_verbose()
1	oh_init()	oh1_init()
	oh_transbound()	oh1_transbound()
2/3	oh_max_local_particles()	oh2_max_local_particles()
	oh_inject_particle()	oh2_inject_particle()
	oh_remap_injected_particle()	oh2_remap_injected_particle()
	oh_remove_injected_particle()	oh2_remove_injected_particle()
2/3/4p/4s	oh_set_total_particles()	oh2_set_total_particles()
2	oh_init()	oh2_init()
	oh_transbound()	oh2_transbound()
3/4p/4s	oh_grid_size()	oh3_grid_size()
	oh_bcast_field()	oh3_bcast_field()
	oh_reduce_field()	oh3_reduce_field()
	oh_allreduce_field()	oh3_allreduce_field()
	oh_exchange_borders()	oh3_exchange_borders()
3	oh_init()	oh3_init()
	oh_transbound()	oh3_transbound()
	oh_map_particle_to_neighbor()	oh3_map_particle_to_neighbor()
	oh_map_particle_to_subdomain()	oh3_map_particle_to_subdomain()
4p	oh_init()	oh4p_init()
	oh_max_local_particles()	oh4p_max_local_particles()
	oh_per_grid_histogram()	oh4p_per_grid_histogram()
	oh_transbound()	oh4p_transbound()
	oh_map_particle_to_neighbor()	oh4p_map_particle_to_neighbor()
	oh_map_particle_to_subdomain()	oh4p_map_particle_to_subdomain()
	oh_inject_particle()	oh4p_inject_particle()
	oh_remove_mapped_particle()	oh4p_remove_mapped_particle()
	oh_remap_particle_to_neighbor()	oh4p_remap_particle_to_neighbor()
	oh_remap_particle_to_subdomain()	oh4p_remap_particle_to_subdomain()
4s	oh_init()	oh4s_init()
	oh_particles_buffer()	oh4s_particles_buffer()
	oh_per_grid_histogram()	oh4s_per_grid_histogram()
	oh_transbound()	oh4s_transbound()
	oh_exchange_border_data()	oh4s_exchange_border_data()
	oh_map_particle_to_neighbor()	oh4s_map_particle_to_neighbor()
	oh_map_particle_to_subdomain()	oh4s_map_particle_to_subdomain()
	oh_inject_particle()	oh4s_inject_particle()
	oh_remove_mapped_particle()	oh4s_remove_mapped_particle()
	oh_remap_particle_to_neighbor()	oh4s_remap_particle_to_neighbor()
	oh_remap_particle_to_subdomain()	oh4s_remap_particle_to_subdomain()

`current_scatter()` also does what its name indicates. The contribution of each particle to the current densities at grid points surrouding it is calculated by an out-of-scope subroutine/function named `scatter()`.

`add_boundary_current()` culculates current density vectors of the grid points in boundary planes of a subdomain adding those obtained from neighboring subdomains to those calcluated by the family members of the local onde. This calls `add_boundary_curr()` for each boundary.

`field_solve_e()` is the first half of a leapfrog field solver to update electric field vectors. The rotation of magnetic field $\nabla \times \boldsymbol{B}$ for the electirc field vector of each grid point is calculated by an out-of-scope subroutine/fucntion named `rotate_b()`.

`field_solve_b()` is the second half of a leapfrog field solver to update magnetic field vectors. Similar to its electric counterpart, $\nabla \times \boldsymbol{E}$ is calculated by an out-of-scope subroutine/fucntion named `rotate_e()`.

In addition to them, it is assumed that we have two out-of-scope subroutines/functions for initialization, namely `initialize_eb()` for electromagnetic field and `initialize_particles()` for particles.

3.13.1 Fortran Sample Code

The Fortran sample code given in the file sample.F90 is composed in a Fortran module named `sample`. It starts with the following lines to `#include` the header file ohhelp_f.h for level-3 function aliasing and to `use` the Fortran module `ohhelp3` defined in oh_mod3.F90 for the `inteface`'s of level-3 and lower level libray functions.

```
#define OH_LIB_LEVEL 3
#include "ohhelp_f.h"
module sample
  use ohhelp3
```

Declaration

At first, we declare a few `parameter`'s, `MAXFRAC` = 20 for `maxfrac` argument of `oh3_init()`, field-array identifiers for electromagnetic field-array `eb(:,:,:,:,:)` (`FEB` = 1) and current density `cd(:,:,:,:,:)` (`FCD` = 2), and element numbers of these arrays, `EX`, `BX`, `JX` and so on.

```
  implicit none
  integer,parameter     :: MAXFRAC=20
  integer,parameter     :: FEB=1,FCD=2
  integer,parameter     :: EX=1,EY=2,EZ=3,BX=4,BY=5,BZ=6
  integer,parameter     :: JX=1,JY=2,JZ=3
```

Then the variables to pass `oh3_init()` are declared with the same names as defined in §3.6.1. We also declare two field-arrays, `eb(:,:,:,:,:)` for electromagnetic field and `cd(:,:,:,:,:)` for current density.

```
  integer               :: sdid(2)
  integer,allocatable   :: nphgram(:,:,:)
  integer,allocatable   :: totalp(:,:)
```

```
type(oh_particle),allocatable&
                      :: pbuf(:)
integer               :: pbase(3)
type(oh_mycomm)       :: mycomm
integer               :: nbor(3,3,3)
integer,allocatable   :: sdoms(:,:,:)
integer               :: bcond(2,OH_DIMENSION)
integer,allocatable   :: bounds(:,:,:)
integer               :: ftypes(7,3)
integer               :: cfields(3)
integer               :: ctypes(3,2,1,2)
integer               :: fsizes(2,OH_DIMENSION,2)
real*8,allocatable    :: eb(:,:,:,:,:)
real*8,allocatable    :: cd(:,:,:,:,:)
```

The last declarative work is to give the prototypes of out-of-scope subroutines.

```
interface
  subroutine initialize_eb(eb, sdom)
    implicit none
    real*8            :: eb(:,:,:,:)
    integer           :: sdom(:,:)
  end subroutine
  subroutine initialize_particles(pbuf, nspec, nphgram)
    use oh_type
    implicit none
    type(oh_particle) :: pbuf(:)
    integer           :: nspec
    integer           :: nphgram(:,:)
  end subroutine
  subroutine lorentz(eb, x, y, z, s, acc)
    implicit none
    real*8            :: eb(:,:,:,:)
    real*8            :: x, y, z
    integer           :: s
    real*8            :: acc(OH_DIMENSION)
  end subroutine
  subroutine scatter(p, s, c)
    use oh_type
    implicit none
    type(oh_particle) :: p
    integer           :: s
    real*8            :: c(3,2,2,2)
  end subroutine
  subroutine rotate_b(eb, x, y, z, rot)
    implicit none
    real*8            :: eb(:,:,:,:)
    integer           :: x, y, z
    real*8            :: rot(OH_DIMENSION)
  end subroutine
  subroutine rotate_e(eb, x, y, z, rot)
    implicit none
    real*8            :: eb(:,:,:,:)
    integer           :: x, y, z
```

```
    real*8            :: rot(OH_DIMENSION)
  end subroutine
end interface
```

Subroutine `pic()`

The first subroutine `pic()` is the core of the simulator and is called with a few simulation parameters to be given to the arguments of `oh3_init()`, which are `nspec`, `pcoord(3)` and `scoord(2,3)`. It also has arguments `npmax` for the absolute maximum number of the particle in the whloe simulation and `nstep` to determine the number of simulation steps.

```
contains
subroutine pic(nspec, pcoord, scoord, npmax, nstep)
  implicit none
  integer             :: nspec
  integer             :: pcoord(OH_DIMENSION)
  integer             :: scoord(2,OH_DIMENSION)
  integer*8           :: npmax
  integer             :: nstep

  integer             :: n, t, maxlocalp, currmode
```

The first job is to `allocate` the array `totalp(nspec,2)` and a few other arrays having N as the size of a dimension, i.e., `nphgram`, `sdoms` and `bounds`. The number of nodes $N = \Pi_x \times \Pi_y \times \Pi_z$ is calculated from `pcoord`. We also `allocate` the particle array `pbuf` whose size `maxlocalp` is determined by `oh2_max_local_particles()` from `npmax` and `MAXFRAC` without additional minimum margin.

```
  allocate(totalp(nspec,2))
  n = pcoord(1) * pcoord(2) * pcoord(3)
  allocate(nphgram(n, nspec, 2))
  allocate(sdoms(2, OH_DIMENSION, n))
  allocate(bounds(2, OH_DIMENSION, n))

  maxlocalp = oh_max_local_particles(npmax, MAXFRAC, 0)
  allocate(pbuf(maxlocalp))
```

We continue initial setting of variables for `oh3_init()`; `nbor` and `sdoms` have the special values to delegate their initializations to `oh3_init()`; `bcond` indicates fully periodic boundary conditions by having 1s in all of its elements; the first element of `ftypes` for `eb` shows that the range for its broadcast is from `eb(1,-1,-1,-1,:)` to `eb(6,`σ_x`,`σ_y`,`σ_z`,:)`, while the second element for `cd` gives that for the reduction being from `cd(1,-1,-1,-1,:)` to `cd(3,`σ_x`+1,`σ_y`+1,`σ_z`+1,:)`; `cfields` has just two elemnets for `eb` and `cd` and thus their communication type identifiers are same as their field identifiers; the first and sencond elements of `ctypes` for `eb` and `cd` are set as shown in Figure 13 and 14 respectively.

Now we can call `oh3_init()` and do it to have the sizes of field-arrays through `ftypes` by which we `allocate` the arrays `eb` and `cd`.

```
  nbor(1,1,1) = -1
  sdoms(1,1,1) = 0;  sdoms(2,1,1) = -1
  bcond(:,:) = reshape((/1,1, 1,1, 1,1/), (/2,OH_DIMENSION/))
```

```
ftypes(:,FEB) = (/6, 0,0, -1,1,  0,0/)                        ! for eb()
ftypes(:,FCD) = (/3, 0,0,  0,0, -1,2/)                        ! for cd()
ftypes(1,FCD+1) = -1                                          ! terminator
cfields(:) = (/FEB,FCD,0/)
ctypes(:,:,1,FEB) = reshape((/ 0,0,2,  -1,-1,1/), (/3,2/))  ! for eb()
ctypes(:,:,1,FCD) = reshape((/-1,2,3,  -1,-4,3/), (/3,2/))  ! for cd()

call oh_init(sdid(:), nspec, MAXFRAC, nphgram(:,:,:), totalp(:,:), &
             pbuf(:), pbase(:), maxlocalp, mycomm, nbor(:,:,:), &
             pcoord(:), sdoms(:,:,:), scoord(:,:), 1, bcond(:,:), &
             bounds(:,:,:), ftypes(:,:), cfields(:), ctypes(:,:,:,:), &
             fsizes(:,:,:), 0, 0, 0)

allocate(eb(6, fsizes(1,1,FEB):fsizes(2,1,FEB), &
               fsizes(1,2,FEB):fsizes(2,2,FEB), &
               fsizes(1,3,FEB):fsizes(2,3,FEB), 2))
allocate(cd(3, fsizes(1,1,FCD):fsizes(2,1,FCD), &
               fsizes(1,2,FCD):fsizes(2,2,FCD), &
               fsizes(1,3,FCD):fsizes(2,3,FCD), 2))
```

We still have a few initializations to have initial setting of `eb` for primary subdomain, whose size and location in the space domain is given in `sdoms(:,:,sdid(1))`, by the out-of-scope subroutine `initialize_eb()`, and that of primary particles in `pbuf` and the count for each of `nspec` species and each of subdomain in `nphgram(:,:,1)` by the out-of-scope subroutine `initialize_particles()`[23]. Then we call `oh3_transbound()` to examine whether the initial particle positioning is balanced and, if not, broadcast `eb` to the helpers of the local node by `oh3_bcast_field()`[24]. Finally, the boundary values of initial setting of `eb` are exchanged between adjacent nodes by `oh3_exchange_borders()`.

```
call initialize_eb(eb(:,:,:,:,1), sdoms(:,:,sdid(1)))
call initialize_particles(pbuf(:), nspec, nphgram(:,:,1))

currmode = oh_transbound(0, 0)
if (currmode.lt.0) then
  call oh_bcast_field(eb(1,0,0,0,1), eb(1,0,0,0,2), FEB)
  currmode = 1
end if
call oh_exchange_borders(eb(1,0,0,0,1), eb(1,0,0,0,2), FEB, currmode)
```

Now we start the main loop of simulation. First, we call `particle_push()` giving it primary particles and the electromagnetic field-array `eb` of primary subdomain. Then, if the local node has secondary particles and subdomain, i.e., `sdid(2)` for its secondary subdomain identifier is not negative, we call the subroutine again giving it secondary particles and the field-array of secondary subdomain. Then we call `oh3_transbound()` to transfer particles among nodes and, if it (re)built the helpand-helper configuration, `oh3_bcast_field()` to broadcast `eb` to helpers.

[23] It might need other parameters to initialize `pbuf`, e.g., the number of initial particles of each species as a whole, but such parameters are also *out-of-scope*.

[24] Broadcasting from the local subdomain coordinates $(-1,-1,-1)$ to $(\sigma_x,\sigma_y,\sigma_z)$ is a little bit larger than what we really need because `oh3_exchange_borders`() just follows, but it is safe and the additional communication cost is negligible.

```
do t=1, nstep
  call particle_push(pbuf(pbase(1):), nspec, totalp(:,1), &
                     eb(:,:,:,:,1), sdoms(:,:,sdid(1)), sdid(1), 0, &
                     nphgram(:,:,1))
  if (sdid(2).ge.0) &
    call particle_push(pbuf(pbase(2):), nspec, totalp(:,2), &
                       eb(:,:,:,:,2), sdoms(:,:,sdid(2)), sdid(2), 1, &
                       nphgram(:,:,2))
  currmode = oh_transbound(currmode, 0)
  if (currmode.lt.0) then
    call oh_bcast_field(eb(1,0,0,0,1), eb(1,0,0,0,2), FEB)
    currmode = 1
  end if
```

Next we call `current_scatter()` once or twice giving it primary and secondary particles and the field-array `cd` of subdomains, to have current density vectors in the primary subdomain, or a partial results of them in primary and secondary subdomains if we are in secondary mode. In the latter case, we call `oh3_allreduce_field()` to have almost complete sums of the vectors in both primary and secondary subdomains. Then, to obtain the contribution of the particles near by the subdomain boundaries and residing (or having resided) in adjacent subdomains, we call `oh3_exchange_borders()` to have the boundary values of `cd`, and `add_boundary_current()` to add them to those calculated by the local node. If the local node has the secondary subdomain, `add_boundary_current()` is called twice, one for the primary subdomain and the other for the secondary.

```
  call current_scatter(pbuf(pbase(1):), nspec, totalp(:,1), &
                       cd(:,:,:,:,1), sdoms(:,:,sdid(1)), &
                       ctypes(:,:,1,FCD))
  if (sdid(2).ge.0) &
    call current_scatter(pbuf(pbase(2):), nspec, totalp(:,2), &
                         cd(:,:,:,:,2), sdoms(:,:,sdid(2)), &
                         ctypes(:,:,1,FCD))
  if (currmode.ne.0) &
    call oh_allreduce_field(cd(1,0,0,0,1), cd(1,0,0,0,2), FCD)
  call oh_exchange_borders(cd(1,0,0,0,1), cd(1,0,0,0,2), FCD, currmode)
  call add_boundary_current(cd(:,:,:,:,1), sdoms(:,:,sdid(1)), &
                            ctypes(:,:,1,FCD))
  if (sdid(2).ge.0) &
    call add_boundary_current(cd(:,:,:,:,2), sdoms(:,:,sdid(2)), &
                              ctypes(:,:,1,FCD))
```

Next, we update field vectors $\boldsymbol{E}$ and $\boldsymbol{B}$ in the primary subdomain by calling `field_solve_e()` and `field_solve_b()` respectively, giving them the field-arrays of the primary subdomain. Then, if the local node has the secondary subdomain, we call these two subroutines again giving them field-arrays of the secondary subdomain. Finally, the boudary values of `eb` are exchanged between adjacent subdomains by `oh3_exchange_borders()` to have what we need in the next simulation step.

```
  call field_solve_e(eb(:,:,:,:,1), cd(:,:,:,:,1), sdoms(:,:,sdid(1)))
  call field_solve_b(eb(:,:,:,:,1), sdoms(:,:,sdid(1)))
  if (sdid(2).ge.0) then
    call field_solve_e(eb(:,:,:,:,2), cd(:,:,:,:,2), sdoms(:,:,sdid(2)))
```

```
      call field_solve_b(eb(:,:,:,:,2), sdoms(:,:,sdid(2)))
    end if
    call oh_exchange_borders(eb(1,0,0,0,1), eb(1,0,0,0,2), FEB, currmode)
  end do
end subroutine
```

Subroutine `particle_push()`

The second subroutine `particle_push()` is given eight arguments to specify primary or secondary particles, primary or secondary subdomain and its field-array; `pbuf` for particle buffer; `nspec` for the number of species; `totalp` for the number of particles in each species; `eb` for the electromagnetic field-array; `sdom` for the size and the location of the subdomain; `n` for the subdomain identifier; `ps` for primary or secondary mode; and `nphgram` for the particle population histogram.

```
subroutine particle_push(pbuf, nspec, totalp, eb, sdom, n, ps, nphgram)
  implicit none
  type(oh_particle) :: pbuf(:)
  integer           :: nspec
  integer           :: totalp(:)
  real*8            :: eb(:,:,:,:)
  integer           :: sdom(:,:)
  integer           :: n
  integer           :: ps
  integer           :: nphgram(:,:)

  integer           :: xl, yl, zl, xu, yu, zu
  integer           :: s, p, q, m
  real*8            :: acc(OH_DIMENSION)
```

Before we enter the double loop for species and particles in each of them, we get lower and upper subdomain boundaries from `sdom` to set them into `xl`, `xu` and so on, for the sake of conciseness (and efficiency if your compiler is not smart enough).

```
  xl=sdom(1,1);  yl=sdom(1,2);  zl=sdom(1,3)
  xu=sdom(2,1);  yu=sdom(2,2);  zu=sdom(2,3)
```

Now we start the double loop letting `nphgram(n+1,s)` have `totalp(s)` as its initial value at the beginning of the iteration for each species `s`, to mean that we will have `totalp(s)` particles in the subdomain `n` if all the particles of the species `s` stay in the subdomain. Then we call `lorentz()` to have the acceleration vector of each particle in the array `acc(3)`, whose elements are added to the velocity vector components of the particle. After this acceleration (or deceleration), the particle is moved by adding the velocity vector to the position vector.

```
  p = 0
  do s=1, nspec
    nphgram(n+1,s) = totalp(s)
    do q=1, totalp(s)
      p = p + 1
      call lorentz(eb, pbuf(p)%x-xl, pbuf(p)%y-yl, pbuf(p)%z-zl, s, acc)
      pbuf(p)%vx = pbuf(p)%vx + acc(1)
      pbuf(p)%vy = pbuf(p)%vy + acc(2)
```

```
      pbuf(p)%vz = pbuf(p)%vz + acc(3)
      pbuf(p)%x = pbuf(p)%x + pbuf(p)%vx
      pbuf(p)%y = pbuf(p)%y + pbuf(p)%vy
      pbuf(p)%z = pbuf(p)%z + pbuf(p)%vz
```

Now we finish the job for a particle if it is still staying in the subdomain. Otherwise, we call `oh3_map_particle_to_neighbor()` to obtain the identifier `m` of the subdomain in which the particle now resides. Then `nphgram(n+1,s)` is decreased by one to indicate that the particle has gone, while `nphgram(m+1,s)` is increased by one to represent its immigration. We also update `nid` element of the particle to show it now resides in the subdomain `m`.

```
      if (pbuf(p)%x.lt.xl .or. pbuf(p)%x.ge.xu .or. &
          pbuf(p)%y.lt.yl .or. pbuf(p)%y.ge.yu .or. &
          pbuf(p)%z.lt.zl .or. pbuf(p)%z.ge.zu) then
        m = oh_map_particle_to_neighbor(pbuf(p)%x, pbuf(p)%y, pbuf(p)%z, ps)
        nphgram(n+1,s) = nphgram(n+1,s) - 1
        nphgram(m+1,s) = nphgram(m+1,s) + 1
        pbuf(p)%nid = m
      end if
    end do
  end do
end subroutine
```

Subroutine `current_scatter()`

The third subroutine `current_scatter()` is given six arguments to specify primary or secondary particles, primary or secondary subdomain and its field-array; `pbuf` for particle buffer; `nspec` for the number of species; `totalp` for the number of particles in each species; `cd` for the field-array of current density vectors; `sdom` for the size and the location of the subdomain; and `ctype` to know the range in `cd` which the particles will contribute to.

```
subroutine current_scatter(pbuf, nspec, totalp, cd, sdom, ctype)
  implicit none
  type(oh_particle) :: pbuf(:)
  integer           :: nspec
  integer           :: totalp(:)
  real*8            :: cd(:,:,:,:)
  integer           :: sdom(:,:)
  integer           :: ctype(3,2)

  integer           :: xl, yl, zl, xu, yu, zu
  integer           :: s, p, q
  integer           :: i, j, k
  real*8            :: x, y, z
  real*8            :: c(3,2,2,2)
```

Before we enter the double loop for species and particles in each of them, we get lower subdomain boundaries from `sdom` to set them into `xl` and so on, and upper bondaries to set those in the local subdomain coordinates into `xu` and so on, for the sake of conciseness. Then we zero-clear `cd` including the boundary planes we will send to adjacent nodes referring to `ctype`.

```
  xl = sdom(1,1);  yl = sdom(1,2);  zl = sdom(1,3)
  xu = sdom(2,1)-xl;  yu = sdom(2,2)-yl;  zu = sdom(2,3)-zl
  do k=ctype(1,1), zu+ctype(1,2)+ctype(1,3)-1
    do j=ctype(1,1), yu+ctype(1,2)+ctype(1,3)-1
      do i=ctype(1,1), xu+ctype(1,2)+ctype(1,3)-1
        cd(JX, i, j, k) = 0.0d0
        cd(JY, i, j, k) = 0.0d0
        cd(JZ, i, j, k) = 0.0d0
  end do;  end do;  end do
```

Now we start the double loop. In each iteration for a particle, we call `scatter()` to have its contribution to the current density vectors of the grid points surrounding it in the array `c(3,2,2,2)`, whose elements are added to the corresponding elements of `cd`.

```
    p = 0
    do s=1, nspec
      do q=1, totalp(s)
        p = p + 1
        call scatter(pbuf(p), s, c)
        x = pbuf(p)%x - xl;  y = pbuf(p)%y - yl;  z = pbuf(p)%z - zl
        do k=0,1;  do j=0,1;  do i=0,1
          cd(JX, x+i, y+j, z+k) = cd(JX, x+i, y+j, z+k) + c(JX, i, j, k)
          cd(JY, x+i, y+j, z+k) = cd(JY, x+i, y+j, z+k) + c(JY, i, j, k)
          cd(JZ, x+i, y+j, z+k) = cd(JZ, x+i, y+j, z+k) + c(JZ, i, j, k)
        end do;  end do;  end do;
      end do
    end do
  end subroutine
```

Subroutine `add_boundary_current()`

The fourth subroutine `add_boundary_current()` is given three arguments to specify the primary or secondary subdomain and its field-array; `cd` for the field-array of current density vectors; `sdom` for the size and the location of the subdomain; and `ctype` to know the boundary planes in `cd`.

```
  subroutine add_boundary_current(cd, sdom, ctype)
    implicit none
    real*8            :: cd(:,:,:,:)
    integer           :: sdom(2,OH_DIMENSION)
    integer           :: ctype(3,2)

    integer           :: xu, yu, zu
    integer           :: sl, dl, nl, su, du, nu
```

First, we calculate the upper boundaries $\sigma_{x,y,z}$ of the subdomain in its local coordinates referring to `sdom` and set them into `xu` and so on. Then, to calculate the base (lowest corrdinate) of the boundary planes, $s^l_{x,y,z}$ and $s^u_{x,y,z}$ for the planes obtained from neighbors and $d^l_{x,y,z}$ and $d^u_{x,y,z}$ for those to add to, and the number of lower and upper boundary planes n_l and n_u, we refer to `ctype` elements to have the followings.

$$\begin{array}{rrr} s^l_{x,y,z} = \texttt{ctype(2,2)} & n_l = \texttt{ctype(3,2)} & d^l_{x,y,z} = s^l_{x,y,z} + n_l \\ s^u_{x,y,z} = \sigma_{x,y,z} + \texttt{ctype(2,1)} & n_u = \texttt{ctype(3,1)} & d^u_{x,y,z} = s^u_{x,y,z} - n_u \end{array}$$

That is, we suppose the planes to add to are at just *inside* of the planes obtained from neighbors.

```
xu = sdom(2,1) - sdom(1,1)
yu = sdom(2,2) - sdom(1,2)
zu = sdom(2,3) - sdom(1,3)
sl = ctype(2,2);  nl = ctype(3,2);  dl = sl + nl
su = ctype(2,1);  nu = ctype(3,1);  du = su - nu
```

Then we call `add_boundary_curr()` six times for lower and upper boundary planes perpendicular to z, y and x axes in this order to do the followings conceptually.

$$[s_x^l, s_x^u{+}n_u){\times}[s_y^l, s_y^u{+}n_u){\times}[d_z^l, d_z^l{+}n_l) \leftarrow$$
$$\quad [s_x^l, s_x^u{+}n_u){\times}[s_y^l, s_y^u{+}n_u){\times}[d_z^l, d_z^l{+}n_l) + [s_x^l, s_x^u{+}n_u){\times}[s_y^l, s_y^u{+}n_u){\times}[s_z^l, s_z^l{+}n_l)$$
$$[s_x^l, s_x^u{+}n_u){\times}[s_y^l, s_y^u{+}n_u){\times}[d_z^u, d_z^u{+}n_u) \leftarrow$$
$$\quad [s_x^l, s_x^u{+}n_u){\times}[s_y^l, s_y^u{+}n_u){\times}[d_z^u, d_z^u{+}n_u) + [s_x^l, s_x^u{+}n_u){\times}[s_y^l, s_y^u{+}n_u){\times}[s_z^u, s_z^u{+}n_u)$$
$$[s_x^l, s_x^u{+}n_u){\times}[d_y^l, d_y^l{+}n_l){\times}[d_z^l, d_z^u{+}n_u) \leftarrow$$
$$\quad [s_x^l, s_x^u{+}n_u){\times}[d_y^l, d_y^l{+}n_l){\times}[d_z^l, d_z^u{+}n_u) + [s_x^l, s_x^u{+}n_u){\times}[s_y^l, s_y^l{+}n_l){\times}[d_z^l, d_z^u{+}n_u)$$
$$[s_x^l, s_x^u{+}n_u){\times}[d_y^u, d_y^u{+}n_u){\times}[d_z^l, d_z^u{+}n_u) \leftarrow$$
$$\quad [s_x^l, s_x^u{+}n_u){\times}[d_y^u, d_y^u{+}n_u){\times}[d_z^l, d_z^u{+}n_u) + [s_x^l, s_x^u{+}n_u){\times}[s_y^u, s_y^u{+}n_u){\times}[d_z^l, d_z^u{+}n_u)$$
$$[d_x^l, d_x^l{+}n_l){\times}[d_y^l, d_y^u{+}n_u){\times}[d_z^l, d_z^u{+}n_u) \leftarrow$$
$$\quad [d_x^l, d_x^l{+}n_l){\times}[d_y^l, d_y^u{+}n_u){\times}[d_z^l, d_z^u{+}n_u) + [s_x^l, s_x^l{+}n_l){\times}[d_y^l, d_y^u{+}n_u){\times}[d_z^l, d_z^u{+}n_u)$$
$$[d_x^u, d_x^u{+}n_u){\times}[d_y^l, d_y^u{+}n_u){\times}[d_z^l, d_z^u{+}n_u) \leftarrow$$
$$\quad [d_x^u, d_x^u{+}n_u){\times}[d_y^l, d_y^u{+}n_u){\times}[d_z^l, d_z^u{+}n_u) + [s_x^u, s_x^l{+}n_u){\times}[d_y^l, d_y^u{+}n_u){\times}[d_z^l, d_z^u{+}n_u)$$

The operations above for a two-dimensional subdomain are illustarted in Figure 15.

```
    call add_boundary_curr(sl, sl, xu+(su+nu-sl), &
                           sl, sl, yu+(su+nu-sl), &
                           sl, dl, nl, cd)
    call add_boundary_curr(sl, sl, xu+(su+nu-sl), &
                           sl, sl, yu+(su+nu-sl), &
                           zu+su, zu+du, nu, cd)
    call add_boundary_curr(sl, sl, xu+(su+nu-sl), &
                           sl, dl, nl, &
                           dl, dl, zu+(du-dl), cd)
    call add_boundary_curr(sl, sl, xu+(su+nu-sl), &
                           yu+su, yu+du, nu, &
                           dl, dl, zu+(du-dl), cd)
    call add_boundary_curr(sl, dl, nl, &
                           dl, dl, yu+(du-dl), &
                           dl, dl, zu+(du-dl), cd)
    call add_boundary_curr(xu+su, xu+du, nu, &
                           dl, dl, yu+(du-dl), &
                           dl, dl, zu+(du-dl), cd)
end subroutine
```

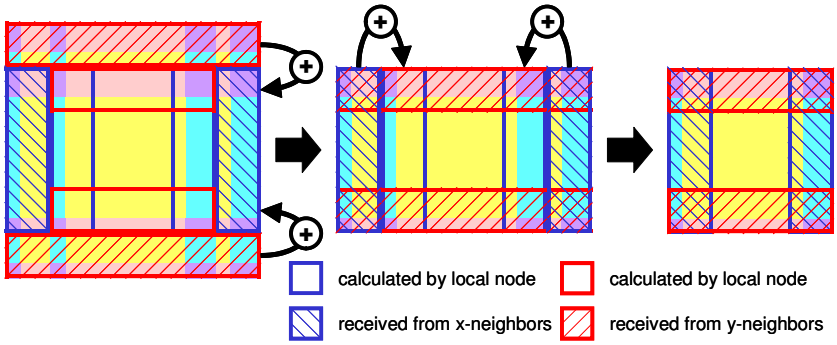


Figure 15: Adding boundary planes of current density vectors.

Subroutine `add_boundary_curr()`

The fifth subroutine `add_boundary_curr()` does the followings conceptually for each current density vector component in `cd` for the boundary plane addition in `add_boundary_current()`.

$$[\mathtt{xd},\mathtt{xd}+\mathtt{nx})\times[\mathtt{yd},\mathtt{yd}+\mathtt{ny})\times[\mathtt{zd},\mathtt{zd}+\mathtt{nz}) \leftarrow [\mathtt{xd},\mathtt{xd}+\mathtt{nx})\times[\mathtt{yd},\mathtt{yd}+\mathtt{ny})\times[\mathtt{zd},\mathtt{zd}+\mathtt{nz}) + [\mathtt{xs},\mathtt{xs}+\mathtt{nx})\times[\mathtt{ys},\mathtt{ys}+\mathtt{ny})\times[\mathtt{zs},\mathtt{zs}+\mathtt{nz})$$

```
subroutine add_boundary_curr(xs, xd, nx, ys, yd, ny, zs, zd, nz, cd)
  implicit none
  integer           :: xs, xd, nx, ys, yd, ny, zs, zd, nz
  integer           :: i, j, k
  real*8            :: cd(:,:,:,:)

  do k=0, nz-1;  do j=0, ny-1;  do i=0, nx-1
    cd(JX, xd+i, yd+j, zd+k) = &
         cd(JX, xd+i, yd+j, zd+k) + cd(JX, xs+i, ys+j, zs+k)
    cd(JY, xd+i, yd+j, zd+k) = &
         cd(JY, xd+i, yd+j, zd+k) + cd(JY, xs+i, ys+j, zs+k)
    cd(JZ, xd+i, yd+j, zd+k) = &
         cd(JZ, xd+i, yd+j, zd+k) + cd(JZ, xs+i, ys+j, zs+k)
  end do;  end do;  end do
end subroutine
```

Subroutine `field_solve_e()`

The sixth subroutine `field_solve_e()` is given three arguments to specify the primary or secondary subdomain and its field-arrays; `eb` for the electromagnetic field-array; `cd` for the field-array of current density vectors; and `sdom` for the size and the location of the subdomain.

```
subroutine field_solve_e(eb, cd, sdom)
  implicit none
  real*8            :: eb(:,:,:,:)
  real*8            :: cd(:,:,:,:)
  integer           :: sdom(2,OH_DIMENSION)
```

```
  integer           :: xu, yu, zu, x, y, z
  real*8            :: rot(OH_DIMENSION)
```

First, we calculate the upper boundaries $\sigma_{x,y,z}$ of the subdomain in its local coordinates referring to `sdom` and set them into `xu` and so on. Then, in the loop for $[0, \sigma_x] \times [0, \sigma_y] \times [0, \sigma_z]$, we update each electric field vector following the Maxwell's (or Ampèr's circuital) law using $\nabla \times \boldsymbol{B}$ calculated by the out-of-scope subroutine `rotate_b()` and set into `rot(3)`, and the current density vectors `cd`. Note that the constants `EPSILON` for ε_0 and `MU` for μ_0 are assumed to have been defined somewhere in the simulation code.

```
  xu = sdom(2,1) - sdom(1,1)
  yu = sdom(2,2) - sdom(1,2)
  zu = sdom(2,3) - sdom(1,3)
  do z=0, zu;  do y=0, yu;  do x=0, xu
    call rotate_b(eb(:,:,:,:), x, y, z, rot)
    eb(EX, x, y, z) = eb(EX, x, y, z) + &
                      (1/EPSILON)*((1/MU)*rot(1) + cd(JX, x, y, z))
    eb(EY, x, y, z) = eb(EY, x, y, z) + &
                      (1/EPSILON)*((1/MU)*rot(2) + cd(JY, x, y, z))
    eb(EZ, x, y, z) = eb(EZ, x, y, z) + &
                      (1/EPSILON)*((1/MU)*rot(3) + cd(JZ, x, y, z))
  end do;  end do;  end do
end subroutine
```

Subroutine `field_solve_b()`

The seventh and last subroutine `field_solve_b()` is given two arguments to specify the primary or secondary subdomain and its field-array; `eb` for the electromagnetic field-array; and `sdom` for the size and the location of the subdomain.

```
subroutine field_solve_b(eb, sdom)
  implicit none
  real*8            :: eb(:,:,:,:)
  integer           :: sdom(2,OH_DIMENSION)

  integer           :: xu, yu, zu, x, y, z
  real*8            :: rot(OH_DIMENSION)
```

First, we calculate the upper boundaries $\sigma_{x,y,z}$ of the subdomain in its local coordinates referring to `sdom` and set them into `xu` and so on. Then, in the loop for $[0, \sigma_x - 1] \times [0, \sigma_y - 1] \times [0, \sigma_z - 1]$, we update each magnetic field vector following the Maxwell's (or Faraday's induction) law using $\nabla \times \boldsymbol{E}$ calculated by the out-of-scope subroutine `rotate_e()` and set into `rot(3)`.

```
  xu = sdom(2,1) - sdom(1,1)
  yu = sdom(2,2) - sdom(1,2)
  zu = sdom(2,3) - sdom(1,3)
  do z=0, zu-1;  do y=0, yu-1;  do x=0, xu-1
    call rotate_e(eb(:,:,:,:), x, y, z, rot)
    eb(BX, x, y, z) = eb(BX, x, y, z) + rot(1)
    eb(BY, x, y, z) = eb(BY, x, y, z) + rot(2)
    eb(BZ, x, y, z) = eb(BZ, x, y, z) + rot(3)
```

```
    end do;  end do;  end do
  end subroutine
end module
```

3.13.2 C Sample Code

The C sample code is given in the file sample.c. It starts with the following lines to `#include` the header file ohhelp_c.h for level-3 function aliasing and prototypes of level-3 and lower level libray functions. It also `#include`'s the standard header file stdlib.h for `malloc()`.

```
#include <stdlib.h>
#define OH_LIB_LEVEL 3
#include "ohhelp_c.h"
```

Declaration

At first, we `#define` a few constants, `MAXFRAC` = 20 for `maxfrac` argument of `oh3_init()`, field-array identifiers for electromagnetic field-array `eb[]` (`FEB` = 0) and current density `cd[]` (`FCD` = 1).

```
#define MAXFRAC 20
#define FEB     0
#define FCD     1
```

Then the variables to pass `oh3_init()` are declared with the same names as defined in §3.6.1 and a part of them are initialized as follows; pointers `pbuf`, `nbor`, `sdoms` and `bounds` have `NULL` to make `oh3_init()` allocate them and initialize the last three in the default manner; `bcond` indicates fully periodic boundary conditions by having 0s in all of its elements; the first element of `ftypes` for `eb` shows that the range for its broadcast is from the local subdomain coordinates $(-1,-1,-1)$ to $(\sigma_x,\sigma_y,\sigma_z)$ while the second element for `cd` gives that for the reduction being from $(-1,-1,-1)$ to $(\sigma_x{+}1,\sigma_y{+}1,\sigma_z{+}1)$; `cfields` has just two elemnets for `eb` and `cd` and thus their communication type identifiers are same as their field identifiers; the first and sencond elements of `ctypes` for `eb` and `cd` are set as shown in Figure 13 and 14 respectively. We also declare two pointer arrays to field-arrays, `eb[2]` for electromagnetic field and `cd[2]` for current density, toghether with thier `struct` namely `ebfield` and `current`.

```
int sdid[2];
int **nphgram[2];
int *totalp[2];
struct S_particle *pbuf=NULL;
int pbase[3];
int *nbor=NULL;
int (*sdoms)[OH_DIMENSION][2]=NULL;
int bcond[OH_DIMENSION][2]={{0,0},{0,0},{0,0}}; /* fully periodic */
int *bounds=NULL;
int ftypes[3][7]={{6, 0,0, -1,1,  0,0},         /* for eb[] */
                  {3, 0,0,  0,0, -1,2},         /* for cd[] */
                  {-1,0,0,  0,0,  0,0},         /* terminator */
};
int cfields[3]={0,1,-1};                        /* for eb[] and cd[] */
```

```
int ctypes[2][1][2][3]={
  {{{ 0,0,2}, {-1,-1,1}}},                    /* for eb[] */
  {{{-1,2,3}, {-1,-4,3}}},                    /* for cd[] */
};
int fsizes[2][OH_DIMENSION][2];
struct ebfield {
  double ex, ey, ez, bx, by, bz;
} *eb[2];
struct current {
  double jx, jy, jz;
} *cd[2];
```

Another declarative work is to give the prototypes of functions defined in this source file and of out-of-scope ones.

```
/* prototypes of funcions defined in sample.c */
void  pic(int nspec, int pcoord[OH_DIMENSION], int scoord[OH_DIMENSION][2],
          long long int npmax, int nstep);
void  particle_push(struct S_particle *pbuf, int nspec, int *totalp,
                    struct ebfield *eb, int sdom[OH_DIMENSION][2],
                    int fsize[OH_DIMENSION][2], int n, int ps, int **nphgram);
void  current_scatter(struct S_particle *pbuf, int nspec, int *totalp,
                      struct current *cd, int sdom[OH_DIMENSION][2],
                      int ctype[2][3], int fsize[OH_DIMENSION][2]);
void  add_boundary_current(struct current *cd, int sdom[OH_DIMENSION][2],
                           int ctype[2][3], int fsize[OH_DIMENSION][2]);
void  add_boundary_curr(int xs, int xd, int nx, int ys, int yd, int ny,
                        int zs, int zd, int nz, struct current *cd,
                        int fsize[3][2]);
void  field_solve_e(struct ebfield *eb, struct current *cd,
                    int sdom[OH_DIMENSION][2], int fsizee[OH_DIMENSION][2],
                    int fsizec[OH_DIMENSION][2]);
void  field_solve_b(struct ebfield *eb, int sdom[OH_DIMENSION][2],
                    int fsize[OH_DIMENSION][2]);

/* prototypes of funcions not defined in sample.c */
void  initialize_eb(struct ebfield *eb, int sdom[OH_DIMENSION][2],
                    int fsize[OH_DIMENSION][2]);
void  initialize_particles(struct S_particle* pbuf, int nspec, int **nphgram);
void  lorentz(struct ebfield *eb, double x, double y, double z, int s,
              int fsize[OH_DIMENSION][2], double acc[OH_DIMENSION]);
void  scatter(struct S_particle p, int s, struct current c[2][2][2]);
void  rotate_b(struct ebfield *eb, double x, double y, double z,
               int fsize[OH_DIMENSION][2], double rot[OH_DIMENSION]);
void  rotate_e(struct ebfield *eb, double x, double y, double z,
               int fsize[OH_DIMENSION][2], double rot[OH_DIMENSION]);
```

The last declarative work is to define two functional macros `field_array_size(FS)` and `malloc_field_array(S,FS)`. The former is to calculate the number of elements in an array of conceptually three dimensional but one-dimensional in reality from `FS` being a subarray of `fsizes[][OH_DIMENSION][2]` reported from `oh3_init()`. The latter is to `malloc()` a field-array whose element is a `struct` named `S` and whose size is given by `FS` being a

subarray of `fsizes`. These macros are for a concise implementation of what we described in §3.6.1.

```
#define field_array_size(FS) \
  ((FS[0][1]-FS[0][0])*(FS[1][1]-FS[1][0])*(FS[2][1]-FS[2][0]))
#define malloc_field_array(S,FS) \
  ((struct S*)malloc(sizeof(struct S)*field_array_size(FS)*2)- \
   FS[0][0]+(FS[0][1]-FS[0][0])*(FS[1][0]+(FS[1][1]-FS[1][0])*FS[2][0]))
```

Function `pic()`

The first function `pic()` is the core of the simulator and is called with a few simulation parameters to be given to the arguments of `oh3_init()`, which are `nspec`, `pcoord[3]` and `scoord[3][2]`. The function also has arguments `npmax` for the absolute maximum number of the particle in the whloe simulation and `nstep` to determine the number of simulation steps.

```
void pic(int nspec, int pcoord[OH_DIMENSION], int scoord[OH_DIMENSION][2],
         long long int npmax, int nstep) {
  int n, i, j, t;
  int currmode;
```

The first job is the allocation of the *bodies* of `totalp` and `nphgram`, which we could depute `oh3_init()` to do but in this example we dare to do for the sake of clearity. The allocation for the former is fairly simple becase we just need an one-dimensional array of $S \times 2$ and make `totalp[0]` and `totalp[1]` point its element [0] and [S]. The allocatoin for the later is a little bit more complicated as exemplified in §3.2.4. Its size N for the number of nodes $N = \Pi_x \times \Pi_y \times \Pi_z$ is calculated from `pcoord`.

```
    totalp[0] = (int*)malloc(sizeof(int)*nspec*2);
    totalp[1] = totalp[0] + nspec;
    n = pcoord[0] * pcoord[1] * pcoord[2];
    nphgram[0] = (int**)malloc(sizeof(int*)*nspec*2);
    nphgram[1] = nphgram[0] + nspec;
    nphgram[0][0] = (int*)malloc(sizeof(int)*n*nspec*2);
    nphgram[1][0] = nphgram[0][0] + n*nspec;
    for (i=0; i<2; i++)  for (j=1; j<nspec; j++)
      nphgram[i][j] = nphgram[i][j-1] + n;
```

Now we can call `oh3_init()` and do it giving the size of `pbuf` calculated by `oh2_max_local_particles()` to its argument `maxlocalp`, and `NULL` to `mycomm` because it is unnecessary. Then, with the sizes of field-arrays given through `ftypes`, we allocate the arrays so that they are pointed by `eb` and `cd` using the macros `malloc_field_array()` and `field_array_size()`.

```
    oh_init((int**)(&sdid), nspec, MAXFRAC, nphgram[0], totalp, &pbuf,
            (int**)(&pbase), oh_max_local_particles(npmax, MAXFRAC, 0), NULL,
            &nbor, pcoord, (int**)(&sdoms), &scoord[0][0], 1, &bcond[0][0],
            &bounds, ftypes[0], cfields, ctypes[0][0][0], (int**)(&fsizes),
            0, 0, 0);

    eb[0] = malloc_field_array(ebfield, fsizes[FEB]);
```

```
eb[1] = eb[0] + field_array_size(fsizes[FEB]);
cd[0] = malloc_field_array(current, fsizes[FCD]);
cd[1] = cd[0] + field_array_size(fsizes[FCD]);
```

We still have a few initializations to have initial setting of `eb` for primary subdomain, whose size and location in the space domain is given in `sdoms[sdid[1]][][]`, by the out-of-scope function `initialize_eb()`, and that of primary particles in `pbuf` and the count for each of `nspec` species and each of subdomain in `nphgram[0][][]` by the out-of-scope function `initialize_particles()`[25]. Note that `initialize_eb()` is also given `fsizes[FEB][][]` as its argument to calculate one-dimensional indices of `eb`. Then we call `oh3_transbound()` to examine whether the initial particle positioning is balanced and, if not, broadcast `eb` to the helpers of the local node by `oh3_bcast_field()`[26]. Finally, the boundary values of initial setting of `eb` are exchanged between adjacent nodes by `oh3_exchange_borders()`.

```
initialize_eb(eb[0], sdoms[sdid[0]], fsizes[FEB]);
initialize_particles(pbuf, nspec, nphgram[0]);

currmode = oh_transbound(0, 0);
if (currmode<0) {
  oh_bcast_field(eb[0], eb[1], FEB);  currmode = 1;
}
oh_exchange_borders(eb[0], eb[1], FEB, currmode);
```

Now we start the main loop of simulation. First, we call `particle_push()` giving it primary particles and the electromagnetic field-array `eb` of primary subdomain. Then, if the local node has secondary particles and subdomain, i.e., `sdid[1]` for its secondary subdomain identifier is not negative, we call the function again giving it secondary particles and the field-array of secondary subdomain. Then we call `oh3_transbound()` to transfer particles among nodes and, if it (re)built the helpand-helper configuration, `oh3_bcast_field()` to broadcast `eb` to helpers.

```
for (t=0; t<nstep; t++) {
  particle_push(pbuf+pbase[0], nspec, totalp[0], eb[0], sdoms[sdid[0]],
                fsizes[FEB], sdid[0], 0, nphgram[0]);
  if (sdid[1]>=0)
    particle_push(pbuf+pbase[1], nspec, totalp[1], eb[1], sdoms[sdid[1]],
                  fsizes[FEB], sdid[1], 1, nphgram[1]);
  currmode = oh_transbound(0, 0);
  if (currmode<0) {
    oh_bcast_field(eb[0], eb[1], 0);  currmode = 1;
  }
```

Next we call `current_scatter()` once or twice giving it primary and secondary particles and the field-array `cd` of subdomains, to have current density vectors in the primary subdomain, or a partial results of them in primary and secondary subdomains if we are

[25] It might need other parameters to initialize `pbuf`, e.g., the number of initial particles of each species as a whole, but such parameters are also *out-of-scope*.

[26] Broadcasting from the local subdomain coordinates $(-1,-1,-1)$ to $(\sigma_x,\sigma_y,\sigma_z)$ is a little bit larger than what we really need because `oh3_exchange_borders`() just follows, but it is safe and the additional communication cost is negligible.

in secondary mode. In the latter case, we call `oh3_allreduce_field()` to have almost complete sums of the vectors in both primary and secondary subdomains. Then, to obtain the contribution of the particles near by the subdomain boundaries and residing (or having resided) in adjacent subdomains, we call `oh3_exchange_borders()` to have the boundary values of `cd`, and `add_boundary_current()` to add them to those calculated by the local node. If the local node has the secondary subdomain, `add_boundary_current()` is called twice, one for the primary subdomain and the other for the secondary.

```
    current_scatter(pbuf+pbase[0], nspec, totalp[0], cd[0], sdoms[sdid[0]],
                    ctypes[FCD][0], fsizes[FCD]);
    if (sdid[1]>=0)
      current_scatter(pbuf+pbase[1], nspec, totalp[1], cd[1], sdoms[sdid[1]],
                      ctypes[FCD][0], fsizes[FCD]);
    if (currmode)  oh_allreduce_field(cd[0], cd[1], FCD);
    oh_exchange_borders(cd[0], cd[1], FCD, currmode);
    add_boundary_current(cd[0], sdoms[sdid[0]], ctypes[FCD][0], fsizes[FCD]);
    if (sdid[1]>=0)
      add_boundary_current(cd[1], sdoms[sdid[1]], ctypes[FCD][0], fsizes[FCD]);
```

Next, we update field vectors $\boldsymbol{E}$ and $\boldsymbol{B}$ in the primary subdomain by calling `field_solve_e()` and `field_solve_b()` respectively, giving them the field-arrays of the primary subdomain. Then, if the local node has the secondary subdomain, we call these two functions again giving them field-arrays of the secondary subdomain. Finally, the boudary values of `eb` are exchanged between adjacent subdomains by `oh3_exchange_borders()` to have what we need in the next simulation step.

```
    field_solve_e(eb[0], cd[0], sdoms[sdid[0]], fsizes[FEB], fsizes[FCD]);
    field_solve_b(eb[0], sdoms[sdid[0]], fsizes[FEB]);
    if (sdid[1]>=0) {
      field_solve_e(eb[1], cd[1], sdoms[sdid[1]], fsizes[FEB], fsizes[FCD]);
      field_solve_b(eb[1], sdoms[sdid[1]], fsizes[FEB]);
    }
    oh_exchange_borders(eb[0], eb[1], FEB, currmode);
  }
}
```

Function `particle_push()`

The second function `particle_push()` is given nine arguments to specify primary or secondary particles, primary or secondary subdomain and its field-array; `pbuf` for particle buffer; `nspec` for the number of species; `totalp` for the number of particles in each species; `eb` for the electromagnetic field-array; `sdom` for the size and the location of the subdomain; `fsize` for the size of `eb`; `n` for the subdomain identifier; `ps` for primary or secondary mode; and `nphgram` for the particle population histogram.

Then, in the local variable declaration, we get lower and upper subdomain boundaries from `sdom` to set them into `xl`, `xu` and so on, for the sake of conciseness (and efficiency if your compiler is not smart enough).

```
void particle_push(struct S_particle *pbuf, int nspec, int *totalp,
                   struct ebfield *eb, int sdom[OH_DIMENSION][2],
                   int fsize[OH_DIMENSION][2], int n, int ps, int **nphgram) {
  int xl=sdom[0][0], yl=sdom[1][0], zl=sdom[2][0];
  int xu=sdom[0][1], yu=sdom[1][1], zu=sdom[2][1];
```

```
    int s, p, q, m;
    double acc[OH_DIMENSION];
```

Now we start the double loop for species and particles in each of them. We let `nphgram[s][n]` have `totalp[s]` as its initial value at the beginning of the iteration for each species `s`, to mean that we will have `totalp[s]` particles in the subdomain `n` if all the particles of the species `s` stay in the subdomain. Then we call `lorentz()` to have the acceleration vector of each particle in the array `acc[3]`, whose elements are added to the velocity vector components of the particle. After this acceleration (or deceleration), the particle is moved by adding the velocity vector to the position vector.

```
    for (s=0,p=0; s<nspec; s++) {
      nphgram[s][n] = totalp[s];
      for (q=0; q<totalp[s]; p++,q++) {
        lorentz(eb, pbuf[p].x-xl, pbuf[p].y-yl, pbuf[p].z-zl, s, fsize, acc);
        pbuf[p].vx += acc[0];
        pbuf[p].vy += acc[1];
        pbuf[p].vx += acc[2];
        pbuf[p].x += pbuf[p].vx;
        pbuf[p].y += pbuf[p].vy;
        pbuf[p].x += pbuf[p].vz;
```

Now we finish the job for a particle if it is still staying in the subdomain. Otherwise, we call `oh3_map_particle_to_neighbor()` to obtain the identifier `m` of the subdomain in which the particle now resides. Then `nphgram[s][n]` is decreased by one to indicate that the particle has gone, while `nphgram[s][m]` is increased by one to represent its immigration. We also update `nid` element of the particle to show it now resides in the subdomain `m`.

```
        if (pbuf[p].x<xl || pbuf[p].x>=xu ||
            pbuf[p].y<yl || pbuf[p].y>=yu ||
            pbuf[p].z<zl || pbuf[p].z>=zu) {
          m = oh_map_particle_to_neighbor(&pbuf[p].x, &pbuf[p].y, &pbuf[p].z,
                                          ps);
          nphgram[s][n]--;  nphgram[s][m]++;
          pbuf[p].nid = m;
        }
      }
    }
  }
```

Function `current_scatter()`

The third function `current_scatter()` is given seven arguments to specify primary or secondary particles, primary or secondary subdomain and its field-array; `pbuf` for particle buffer; `nspec` for the number of species; `totalp` for the number of particles in each species; `cd` for the field-array of current density vectors; `sdom` for the size and the location of the subdomain; `ctype` to know the range in `cd` which the particles will contribute to; and `fsize` for the size of `cd`.

Then, in the local variable declaration, we get lower subdomain boundaries from `sdom` to set them into `xl` and so on, and upper bondaries to set those in the local subdomain coordinates into `xu` and so on, for the sake of conciseness. We also have local variables `w` for *width* of the field array `cd` and `wd` for width times *depth* of it to calculate the index of `cd` correspoinding to the local subdomain coordinates (x, y, z) by $x + \mathtt{w} \cdot y + \mathtt{wd} \cdot z$.

```
void current_scatter(struct S_particle *pbuf, int nspec, int *totalp,
                     struct current *cd, int sdom[OH_DIMENSION][2],
                     int ctype[2][3], int fsize[OH_DIMENSION][2]) {
  int xl=sdom[0][0], yl=sdom[1][0], zl=sdom[2][0];
  int xu=sdom[0][1]-xl, yu=sdom[1][1]-yl, zu=sdom[2][1]-zl;
  int w=fsize[0][1]-fsize[0][0], wd=w*(fsize[1][1]-fsize[1][0]);
  int s, p, q;
  int i, j, k;
  struct current c[2][2][2];
```

First we zero-clear `cd` including the boundary planes we will send to adjacent nodes referring to `ctype`.

```
  for (k=ctype[0][0]; k<zu+ctype[1][0]+ctype[1][2]; k++)
    for (j=ctype[0][0]; j<yu+ctype[1][0]+ctype[1][2]; j++)
      for (i=ctype[0][0]; i<xu+ctype[1][0]+ctype[1][2]; i++)
        cd[i+w*j+wd*k].jx = cd[i+w*j+wd*k].jy = cd[i+w*j+wd*k].jz = 0.0;
```

Now we start the double loop. In each iteration for a particle, we call `scatter()` to have its contribution to the current density vectors of the grid points surrounding it in the array `c[2][2][2]`, whose elements are added to the corresponding elements of `cd`.

```
  for (s=0,p=0; s<nspec; s++) {
    for (q=0; q<totalp[s]; p++,q++) {
      int x=pbuf[p].x-xl, y=pbuf[p].y-yl, z=pbuf[p].z-zl;
      scatter(pbuf[p], s, c);
      for (k=0; k<2; k++)  for (j=0; j<2; j++)  for (i=0; i<2; i++) {
          cd[(x+i)+w*(y+j)+wd*(z+k)].jx += c[k][j][i].jx;
          cd[(x+i)+w*(y+j)+wd*(z+k)].jy += c[k][j][i].jy;
          cd[(x+i)+w*(y+j)+wd*(z+k)].jz += c[k][j][i].jz;
      }
    }
  }
}
```

Function `add_boundary_current()`

The fourth function `add_boundary_current()` is given four arguments to specify the primary or secondary subdomain and its field-array; `cd` for the field-array of current density vectors; `sdom` for the size and the location of the subdomain; `ctype` to know the boundary planes in `cd`; and `fsize` for the size of `cd`.

In the local variable declaration, we calculate the upper boundaries $\sigma_{x,y,z}$ of the subdomain in its local coordinates referring to `sdom` and set them into `xu` and so on. Then, to calculate the base (lowest corrdinate) of the boundary planes, $s^l_{x,y,z}$ and $s^u_{x,y,z}$ for the planes obtained from neighbors and $d^l_{x,y,z}$ and $d^u_{x,y,z}$ for those to add to, and the number of lower and upper boundary planes n_l and n_u, we refer to `ctype` elements to have the followings.

$$\begin{array}{lll} s^l_{x,y,z} = \texttt{ctype[1][1]} & n_l = \texttt{ctype[1][2]} & d^l_{x,y,z} = s^l_{x,y,z} + n_l \\ s^u_{x,y,z} = \sigma_{x,y,z} + \texttt{ctype[0][1]} & n_u = \texttt{ctype[0][2]} & d^u_{x,y,z} = s^u_{x,y,z} - n_u \end{array}$$

That is, we suppose the planes to add to are at just *inside* of the planes obtained from neighbors.

```
void add_boundary_current(struct current *cd, int sdom[OH_DIMENSION][2],
                          int ctype[2][3], int fsize[OH_DIMENSION][2]) {
  int xu=sdom[0][1]-sdom[0][0], yu=sdom[1][1]-sdom[1][0],
      zu=sdom[2][1]-sdom[2][0];
  int sl=ctype[1][1], nl=ctype[1][2], dl=sl+nl;
  int su=ctype[0][1], nu=ctype[0][2], du=su-nu;
```

Then we call `add_boundary_curr()` six times for lower and upper boundary planes perpendicular to z, y and x axes in this order to do the followings conceptually.

$$[s_x^l, s_x^u{+}n_u){\times}[s_y^l, s_y^u{+}n_u){\times}[d_z^l, d_z^l{+}n_l) \leftarrow$$
$$\quad [s_x^l, s_x^u{+}n_u){\times}[s_y^l, s_y^u{+}n_u){\times}[d_z^l, d_z^l{+}n_l) + [s_x^l, s_x^u{+}n_u){\times}[s_y^l, s_y^u{+}n_u){\times}[s_z^l, s_z^l{+}n_l)$$
$$[s_x^l, s_x^u{+}n_u){\times}[s_y^l, s_y^u{+}n_u){\times}[d_z^u, d_z^u{+}n_u) \leftarrow$$
$$\quad [s_x^l, s_x^u{+}n_u){\times}[s_y^l, s_y^u{+}n_u){\times}[d_z^u, d_z^u{+}n_u) + [s_x^l, s_x^u{+}n_u){\times}[s_y^l, s_y^u{+}n_u){\times}[s_z^u, s_z^u{+}n_u)$$
$$[s_x^l, s_x^u{+}n_u){\times}[d_y^l, d_y^l{+}n_l){\times}[d_z^l, d_z^u{+}n_u) \leftarrow$$
$$\quad [s_x^l, s_x^u{+}n_u){\times}[d_y^l, d_y^l{+}n_l){\times}[d_z^l, d_z^u{+}n_u) + [s_x^l, s_x^u{+}n_u){\times}[s_y^l, s_y^l{+}n_l){\times}[d_z^l, d_z^u{+}n_u)$$
$$[s_x^l, s_x^u{+}n_u){\times}[d_y^u, d_y^u{+}n_u){\times}[d_z^l, d_z^u{+}n_u) \leftarrow$$
$$\quad [s_x^l, s_x^u{+}n_u){\times}[d_y^u, d_y^u{+}n_u){\times}[d_z^l, d_z^u{+}n_u) + [s_x^l, s_x^u{+}n_u){\times}[s_y^u, s_y^u{+}n_u){\times}[d_z^l, d_z^u{+}n_u)$$
$$[d_x^l, d_x^l{+}n_l){\times}[d_y^l, d_y^u{+}n_u){\times}[d_z^l, d_z^u{+}n_u) \leftarrow$$
$$\quad [d_x^l, d_x^l{+}n_l){\times}[d_y^l, d_y^u{+}n_u){\times}[d_z^l, d_z^u{+}n_u) + [s_x^l, s_x^l{+}n_l){\times}[d_y^l, d_y^u{+}n_u){\times}[d_z^l, d_z^u{+}n_u)$$
$$[d_x^u, d_x^u{+}n_u){\times}[d_y^l, d_y^u{+}n_u){\times}[d_z^l, d_z^u{+}n_u) \leftarrow$$
$$\quad [d_x^u, d_x^u{+}n_u){\times}[d_y^l, d_y^u{+}n_u){\times}[d_z^l, d_z^u{+}n_u) + [s_x^u, s_x^l{+}n_u){\times}[d_y^l, d_y^u{+}n_u){\times}[d_z^l, d_z^u{+}n_u)$$

The operations above for a two-dimensional subdomain are illustarted in Figure 15.

```
  add_boundary_curr(sl, sl, xu+(su+nu-sl),
                    sl, sl, yu+(su+nu-sl),
                    sl, dl, nl, cd, fsize);
  add_boundary_curr(sl, sl, xu+(su+nu-sl),
                    sl, sl, yu+(su+nu-sl),
                    zu+su, zu+du, nu, cd, fsize);
  add_boundary_curr(sl, sl, xu+(su+nu-sl),
                    sl, dl, nl,
                    dl, dl, zu+(du-dl), cd, fsize);
  add_boundary_curr(sl, sl, xu+(su+nu-sl),
                    yu+su, yu+du, nu,
                    dl, dl, zu+(du-dl), cd, fsize);
  add_boundary_curr(sl, dl, nl,
                    dl, dl, yu+(du-dl),
                    dl, dl, zu+(du-dl), cd, fsize);
  add_boundary_curr(xu+su, xu+du, nu,
                    dl, dl, yu+(du-dl),
                    dl, dl, zu+(du-dl), cd, fsize);
}
```

Function `add_boundary_curr()`

The fifth function `add_boundary_curr()` does the followings conceptually for each current density vector component in `cd` for the boundary plane addition in `add_boundary_current()`.

$$[\mathtt{xd},\mathtt{xd}{+}\mathtt{nx})\times[\mathtt{yd},\mathtt{yd}{+}\mathtt{ny})\times[\mathtt{zd},\mathtt{zd}{+}\mathtt{nz}) \leftarrow [\mathtt{xd},\mathtt{xd}{+}\mathtt{nx})\times[\mathtt{yd},\mathtt{yd}{+}\mathtt{ny})\times[\mathtt{zd},\mathtt{zd}{+}\mathtt{nz}) + [\mathtt{xs},\mathtt{xs}{+}\mathtt{nx})\times[\mathtt{ys},\mathtt{ys}{+}\mathtt{ny})\times[\mathtt{zs},\mathtt{zs}{+}\mathtt{nz})$$

```
void  add_boundary_curr(int xs, int xd, int nx, int ys, int yd, int ny,
                        int zs, int zd, int nz, struct current *cd,
                        int fsize[3][2]) {
  int w=fsize[0][1]-fsize[0][0], wd=w*(fsize[1][1]-fsize[1][0]);
  int i, j, k;

  for (k=0; k<nz; k++)  for (j=0; j<ny; j++)  for (i=0; i<nz; i++) {
    cd[(xd+i)+w*(yd+j)+wd*(zd+k)].jx += cd[(xs+i)+w*(ys+j)+wd*(zs+k)].jx;
    cd[(xd+i)+w*(yd+j)+wd*(zd+k)].jy += cd[(xs+i)+w*(ys+j)+wd*(zs+k)].jy;
    cd[(xd+i)+w*(yd+j)+wd*(zd+k)].jz += cd[(xs+i)+w*(ys+j)+wd*(zs+k)].jz;
  }
}
```

Function `field_solve_e()`

The sixth function `field_solve_e()` is given five arguments to specify the primary or secondary subdomain and its field-arrays; `eb` for the electromagnetic field-array; `cd` for the field-array of current density vectors; `sdom` for the size and the location of the subdomain; and `fsizee` and `fsizec` for the sizes of `eb` and `cd`.

In the local variable declaration, we calculate the upper boundaries $\sigma_{x,y,z}$ of the subdomain in its local coordinates referring to `sdom` and set them into `xu` and so on. We also calculate the width and width times depth of `eb` and `cd` to set them into `we`, `wde`, `wc` and `wdc`.

```
void field_solve_e(struct ebfield *eb, struct current *cd,
                   int sdom[OH_DIMENSION][2],
                   int fsizee[OH_DIMENSION][2], int fsizec[OH_DIMENSION][2]) {
  int xu=sdom[0][1]-sdom[0][0], yu=sdom[1][1]-sdom[1][0],
      zu=sdom[2][1]-sdom[2][0];
  int we=fsizee[0][1]-fsizee[0][0], wde=we*(fsizee[1][1]-fsizee[1][0]);
  int wc=fsizec[0][1]-fsizec[0][0], wdc=wc*(fsizec[1][1]-fsizec[1][0]);
  int x, y, z;
  double rot[OH_DIMENSION];
```

Then, in the loop for $[0,\sigma_x]\times[0,\sigma_y]\times[0,\sigma_z]$, we update each electric field vector following the Maxwell's (or Ampèr's circuital) law using $\nabla\times\boldsymbol{B}$ calculated by the out-of-scope function `rotate_b()` and set into `rot[3]`, and the current density vectors `cd`. Note that the constants `EPSILON` for ε_0 and `MU` for μ_0 are assumed to have been defined somewhere in the simulation code.

```
  for (z=0; z<=zu; z++)  for (y=0; y<=yu; y++)  for (x=0; x<=xu; x++) {
    rotate_b(eb, x, y, z, fsizee, rot);
    eb[x+y*we+z*wde].ex += (1/EPSILON)*((1/MU)*rot[0] + cd[x+y*wc+z*wdc].jx);
```

```
      eb[x+y*we+z*wde].ey += (1/EPSILON)*((1/MU)*rot[1] + cd[x+y*wc+z*wdc].jy);
      eb[x+y*we+z*wde].ez += (1/EPSILON)*((1/MU)*rot[2] + cd[x+y*wc+z*wdc].jz);
    }
  }
```

Function `field_solve_b()`

The seventh and last function `field_solve_b()` is given three arguments to specify the primary or secondary subdomain and its field-array; `eb` for the electromagnetic field-array; `sdom` for the size and the location of the subdomain; and `fsize` for the size of `eb`.

In the local variable declaration, we calculate the upper boundaries $\sigma_{x,y,z}$ of the subdomain in its local coordinates referring to `sdom` and set them into `xu` and so on. We also calculate the width and width times depth of `eb` to set them into `w` and `wd`.

```
void field_solve_b(struct ebfield *eb, int sdom[OH_DIMENSION][2],
                   int fsize[OH_DIMENSION][2]) {
  int xu=sdom[0][1]-sdom[0][0], yu=sdom[1][1]-sdom[1][0],
      zu=sdom[2][1]-sdom[2][0];
  int w=fsize[0][1]-fsize[0][0], wd=w*(fsize[1][1]-fsize[1][0]);
  int x, y, z;
  double rot[OH_DIMENSION];
```

Then, in the loop for $[0, \sigma_x - 1] \times [0, \sigma_y - 1] \times [0, \sigma_z - 1]$, we update each magnetic field vector following the Maxwell's (or Faraday's induction) law using $\nabla \times \boldsymbol{E}$ calculated by the out-of-scope function `rotate_e()` and set into `rot[3]`.

```
  for (z=0; z<xu; z++)  for (y=0; y<xu; y++)  for (x=0; x<xu; x++) {
    rotate_e(eb, x, y, z, fsize, rot);
    eb[x+y*w+z*wd].bx += rot[0];
    eb[x+y*w+z*wd].by += rot[1];
    eb[x+y*w+z*wd].bz += rot[2];
  }
}
```

3.14 How to `make`

Since the OhHelp library includes header files which may be (or is expected to be) customized to your own simulator, it should be confusing if we provide a `Makefile` to build a library archive which could be mistakingly assumed independent of your customization. Therefore, the distribution of OhHelp merely has *samples* of `Makefile` namely samplef.mk and samplec.mk to `make` your simulator in Fortran and C together with the librarary coded in C.

The sample `Makefile` for Fortran samplef.mk represents the dependency shown in Table 2, while its C counterpart samplec.mk corresponds to that shown in Table 3, providing that you choose level-L library[27]. In the sample files, it is assumed that your simulator has just two sources, sample.F90 and simulator.F90 or sample.c and simulator.c, and simulator.{F90,c} provides `main` routines and out-of-scope subroutines/functions used in sample.{F90,c}. It is also assumed your source files need neither of your own header files nor module files to be `#include`'d or `use`'d, although usually you should have some of them.

[27] The tables show dependencies accurately and strictly, but sample `Makefile`'s have redundant (but safe) dependencies such as that ohhelp1.c depends on ohhelp3.h.

Table 2: File Dependency of Fortran Codes.

file	depends on
simulator	simulator.o sample.o oh_mod$l$.o ohhelp$l$.o ($l \in [1, L]$)
simulator.o	simulator.F90 sample.o[*1] oh_mod$L$.o[*1] ohhelp_f.h[*2] oh_config.h[*3] oh_stats.h[*4]
sample.o	sample.F90 oh_mod$L$.o[*1] ohhelp_f.h[*2] oh_conifg.h[*3] oh_stats.h[*4]
oh_mod4p.o	oh_mod4p.F90 oh_mod3.o[*1] oh_config.h
oh_mod4s.o	oh_mod4s.F90 oh_mod3.o[*1] oh_config.h
oh_mod3.o	oh_mod3.F90 oh_mod2.o[*1] oh_config.h
oh_mod2.o	oh_mod2.F90 oh_mod1.o[*1] oh_config.h
oh_mod1.o	oh_mod1.F90 oh_type.o[*1] oh_config.h
oh_type.o	oh_type.F90
ohhelp4p.o	ohhelp4p.c ohhelp4p.h ohhelp3.h ohhelp2.h ohhelp1.h oh_config.h oh_stats.h oh_part.h
ohhelp4s.o	ohhelp4s.c ohhelp4s.h ohhelp3.h ohhelp2.h ohhelp1.h oh_config.h oh_stats.h oh_part.h
ohhelp3.o	ohhelp3.c ohhelp3.h ohhelp2.h ohhelp1.h oh_config.h oh_stats.h oh_part.h
ohhelp2.o	ohhelp2.c ohhelp2.h ohhelp1.h oh_config.h oh_stats.h oh_part.h
ohhelp1.o	ohhelp1.c ohhelp1.h oh_config.h oh_stats.h

[*1] Dependence to `*.o` files represents that a file providing a `module` must be compiled prior to files which `use` it if it is modified.

[*2] If you use function aliasing.

[*3] If you refer to `OH_DIMENSION`.

[*4] If you use statistics functions.

Table 3: File Dependency of C Codes.

file	depends on
simulator	simulator.o sample.o ohhelp$l$.o ($l \in [1, L]$)
simulator.o	simulator.c ohhelp_c.h[*1] oh_part.h[*2] oh_config.h[*3] oh_stats.h[*4]
sample.o	sample.c ohhelp_c.h[*1] oh_part.h[*2] oh_config.h[*3] oh_stats.h[*4]
ohhelp4p.o	ohhelp4p.c ohhelp4p.h ohhelp3.h ohhelp2.h ohhelp1.h oh_config.h oh_stats.h oh_part.h
ohhelp4s.o	ohhelp4s.c ohhelp4s.h ohhelp3.h ohhelp2.h ohhelp1.h oh_config.h oh_stats.h oh_part.h
ohhelp3.o	ohhelp3.c ohhelp3.h ohhelp2.h ohhelp1.h oh_config.h oh_stats.h oh_part.h
ohhelp2.o	ohhelp2.c ohhelp2.h ohhelp1.h oh_config.h oh_stats.h oh_part.h
ohhelp1.o	ohhelp1.c ohhelp1.h oh_config.h oh_stats.h

[*1] If you use function aliasing.

[*2] If $L \geq 2$.

[*3] If you refer to `OH_DIMENSION`.

[*4] If you use statistics functions.

4 Implementation

This section describes the implementation details of the OhHelp library showing every line of almost all source files, which are extracted from this section to make the source files perfectly correspond to the explanation given in this section.

The library package consists of the following files.

Common headers oh_config.h, oh_part.h and oh_stats.h for the library and simulator body programs, which were discussed and shown in §3.3, §3.5.1 and §3.10.1.

Level-1 library sources ohhelp1.h and ohhelp1.c, whose source lines are explained and shown in §4.2 and §4.3, to implement the fundamental part of OhHelp algorithm, basic collective communications among family members, statistics collection and reporting, and verbose messaging.

Level-2 library sources ohhelp2.h and ohhelp2.c, whose source lines are explained and shown in §4.4 and §4.5, to implement particle transfer and injection.

Level-3 library sources ohhelp3.h and ohhelp3.c, whose source lines are explained and shown in §4.6 and §4.7, to implement particle-to-subdomain mapping and communications of field-arrays.

Level-4p library sources ohhelp4p.h and ohhelp4p.c, whose source lines are explained and shown in §4.9 and §4.10, to implement position-aware particle management.

Level-4s library sources ohhelp4s.h and ohhelp4s.c, whose source lines are explained and shown in §4.12 and §4.13, to implement yet another position-aware particle management for, e.g., SPH method.

Fortran module sources oh_type.F90 shown in §3.4.1 and §3.5.1, oh_mod1.F90 shown §3.4, oh_mod2.F90 shown in §3.5, oh_mod3.F90 shown in §3.6, oh_mod4p.F90 shown in §3.7, and oh_mod4s.F90 shown in §3.8, to provide Fortran structured data type `oh_mycomm` and `oh_particle` and the prototypes of Fortran API functions.

Headers for function aliasing ohhelp_f.h for Fortran and ohhelp_c.h for C, which are discussed and shown in §4.2.11, §4.4.5, §4.6.6, §4.9.7 and §4.12.7.

Sample simulator sources sample.F90 and sample.c which were shown in §3.13.1 and §3.13.2.

Sample `make` files samplef.mk for Fortran and samplec.mk for C, which are given in §4.14.1 and §4.14.2.

4.1 Naming Convention

To name indentifiers, i.e., variables, functions and so on, we use the following conventions.

Macro Constants are named only with uppercase letters and underscores as usual. For example, `OH_DIMENSION` is a macro constant.

Global Variables are named with a combination of uppercase and lowercase letters. The first letter of an atomic varialbe is lowercase, while an array or a structured variable is capitalized. For example, `nOfNodes` is an integer global variable, while `NOfPrimaries` is a global and (concepturally) three-dimensional array. Their names usually do not

have underscores but there are two exceptions. One is for MPI data-types whose names are prefixed by `T_` like `T_Histogram`. The other is for MPI operators whose names start with `Op_` like `Op_StatsTime`.

Local Variables are named only with lowercase letters without underscores, such as `i` and `nn`.

Structures have a prefix `S_` followed by lowercase letters. An element of a structure is named only with lowercase letters without underscores. For example, `S_heap` is a `struct` having three elements `n`, `node` and `index`.

Functions are named with lowercase letters and usually with underscores. API functions is prefixed by `oh`*l*`_` where *l* is the library level identifier (i.e., 1, 2, 3, 4p or 4s), and also postfixed by an underscore for those called from Fortran. For example, `oh1_transbound()` is an API functions for C codes while its Fortran counterpart is named as `oh1_transbound_()`.

Functional Macros have capitalized name like `Vprint()`. If a name has underscores, the letters following them are also capitalized like `Stats_Reduce_Part_Min()`.

4.2 Header File ohhelp1.h

The header file of level-1 library, ohhelp1.h, `#define`'s a few basic constants and shorthands, defines `struct`ured data types, declares global variables used in level-1 and higher level C codes, and gives prototypes of API functions and those called by higher level codes.

4.2.1 Header File Inclusion

The first part of ohhelp1.h has a few lines to include the following standard headers.

- stdio.h for printing debug and statistics messages.
- stdlib.h for `malloc()` for allocate data allocations and `qsort()` for balanced particle distribution in a family.
- string.h for `strcat()` to create debug messages.
- limits.h to refer to `INT_MAX` for statistics calculation.
- float.h to refer to `DBL_MAX` for statistics calculation.
- stdarg.h for `vprintf()` and other variable-number-argument stuff for verbose and debug messaging.
- mpi.h for MPI functions and constants.

In addtion to them, we also include our own header files, oh_config.h to define the number of dimensions $D =$ `OH_DIMENSION` of simulated space and the library level as discussed in §3.3, and oh_stats.h to define keys and identification strings for timing measurement intervals as discussed in §3.10.1.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <float.h>
#include <stdarg.h>
#include <mpi.h>

#include "oh_config.h"
#include "oh_stats.h"
```

4.2.2 Constants and Shorthands

Next stuff is a sequence of a few constant and shorthand definitions. We define the following constants.

`TRUE` `FALSE`
- If `TRUE` and `FALSE` have not been defined in any standard header files we have included, they are simply defined as integer constants 1 and 0.

`OH_POS_AWARE`
- The switch `OH_POS_AWARE` is defined if `OH_LIB_LEVEL_4PS` is also defined to mean that level-4p/4s extension and thus position-aware particle management are in effect.

OH_DIM_X
OH_DIM_Y
OH_DIM_Z

- The constants `OH_DIM_X` = 0, `OH_DIM_Y` = 1 and `OH_DIM_Z` = 2 are used as indices of arrays having dimension-dependent values such as that keeping the grid size of the simulated space domain.

OH_NEIGHBORS

- The constant `OH_NEIGHBORS` is defined as 3^D using `#if/#elif/#else/#endif` construct examining `OH_DIMENSION`. This number is one more than the number of subdomain cuboids contacted with a subdomain, in order to include the subdomain itself into its *neighbors*.

Then we define the following shorthands.

MCW

- The constant `MCW` is the shorthand of the MPI constant `MPI_COMM_WORLD`. Since almost all MPI function requires communicator arguments and they are almost always the lengthy `MPI_COMM_WORLD`, we introduced this shorthand to save typing. Also it might be useful if your simulator has some MPI processes not correspoing to decomposed subdomains, because you may replace `MPI_COMM_WORLD` with a global variable having the communicator for prosseses working with subdomains. That is, for example, you may modify the initializer function `oh1_init[_]()` (§4.3.3) to give it an additional argument having your own communicator and to assign it to the global variable equivalent to `MCW`. If you do so and you create the communicator in Fortran code, remember that you have to transform it into C's counterpart by `MPI_Comm_f2c()` in the modified `oh1_init[_]()`.

dint

- The data-type `dint` is the short hand of 64-bit integer `long␣long␣int`. Note that `long␣int` can be used with 64-bit compliant C compilers but `long␣long␣int` is safe for both of 32-bit and 64-bit compilers (so far).

```
#ifndef TRUE
#define TRUE 1
#endif
#ifndef FALSE
#define FALSE 0
#endif

#ifdef  OH_LIB_LEVEL_4PS
#define OH_POS_AWARE
#endif

/* constants for D-dimensional simulation */
#define OH_DIM_X        0
#define OH_DIM_Y        1
#define OH_DIM_Z        2
#if OH_DIMENSION==1
#define OH_NEIGHBORS    3
#elif OH_DIMENSION==2
#define OH_NEIGHBORS    (3*3)
#else
#define OH_NEIGHBORS    (3*3*3)
#endif

#define MCW MPI_COMM_WORLD      /* shorthand of MPI_COMM_WORLD */

typedef long long int dint;     /* shorthand of 64-bit integer */
```

4.2.3 Basic Process Configuration Variables

Here we start the declarations of global variables and structured data-types. Before listing the first variable group for basic configuration, we note a well-known trick to avoid duplicated declarations.

EXTERN

Each global variable declaration has a prefix `EXTERN` which will be defined as an empty string or C keyword `extern`. That is, among C source files in which we include `ohhelp1.h`, only one file, namely `ohhelp1.c`, defines `EXTERN` as empty to give the variables their *home*, while other files, namely `ohhelp2.c` and higher level library sources, let `EXTERN` be `#undef`'ined to make `ohhelp1.h` `#define` it as `extern` and to refer to the variables defined in the other C file.

Now here is the list of global variables to represent the basic process configuration of simulation.

nOfNodes

- The integer variable `nOfNodes` has the number of computation nodes (MPI processes) involved in the parallel simulation. That is, `nOfNodes` is the `size` of `MPI_COMM_WORLD` given by `MPI_Comm_size()` and is set by `init1()`. This variable is referred to in many functions. Hereafter the value of `nOfNodes` is denoted by N.

myRank

- The integer variable `myRank` has the rank of the local node (process). That is, `myRank` is the `rank` of `MPI_COMM_WORLD` given by `MPI_Comm_rank()` and is set by `init1()`. This variable is referred to in many functions.

RegionId
SubdomainId

- The integer array `RegionId`[2] has identifiers of primary (0) and secondary (1) subdomains. Since a subdomain identifier is that of the node which is responsible for the subdomain as its primary one, `RegionId`[0] is always equivalent to `myRank` and thus is set by `init1()`. On the other hand, `RegionId`[1] has the rank of the local node n's parent being $parent(n)$ and thus it may be -1 if n is the root of the helpand-helper tree or we are in primary mode. Therefore, it is set to -1 by `init1()` and `try_primary1()` to indicate that the local node does not have the secondary subdomain as any other nodes, while `rebalance1()` sets it to $parent(n)$ when it (re)builds the helpand-helper relationship. The array is referred to by `count_stay()`, `transbound3()`, `oh3_map_particle_to_neighbor()`, `oh3_exchange_borders()` and `set_border_exchange()`, while the simulator body does so through the *shadow* of the array pointed by `SubdomainId` to protect `RegionId` from accidental modifications. That is, the body of `SubdomainId` is allocated by the simulator body which gives (double) pointer to it `oh1_init()` through the argument `sdid`, or by `init1()` if `sdid` points `NULL`, and `init1()`, `try_primary1()` and `rebalance1()` update the body when they update `RegionId[]`.

currMode

- The integer variable `currMode` has one of the following values, which are usually returned from `oh1_transbound()`, or of its level-2/3 counterparts `oh2_transbound()` or `oh3_transbound()`, but can be modified by other functions to force anywhere accommodation indicated by bit-1.

MODE_NORM_PRI

0: (`MODE_NORM_PRI`) The next simulation step is executed in primary mode.

MODE_NORM_SEC

1: (`MODE_NORM_SEC`) The next simulation step is executed in secondary mode keeping the helper-tree unchanged from the last step.

MODE_REB_SEC

-1: (`MODE_REB_SEC`) The next simulation step is executed in secondary mode with the reconfiguration of the helper-tree.

MODE_ANY_PRI
: 2: (MODE_ANY_PRI) The current simulation step is executed in primary mode with anywhere accommodation regardless of the real accommodation status.

MODE_ANY_SEC
: 3: (MODE_ANY_SEC) The current simulation step is executed in secondary mode with anywhere accommodation regardless of the real accommodation status.

After initialized to MODE_NORM_PRI by init1(), currMode is set to MODE_NORM_PRI, MODE_NORM_SEC, or MODE_REB_SEC by transbound1() or transbound2() and possibly modified by functions such as those for position-aware particle management. Then the variable is referred to by three functions; transbound1() to check the simulator body and the library agree the execution mode; set_total_particles() to know if NOfPLocal[1][][] are valid; and the level-3 API oh3_exchange_borders() to decide whether it broadcasts exchanged boundary values of a field-array to helpers.

The functions above and others called from them with the argument currmode as the local version of currMode use the following macros to examine and/or modify the bit-0 of primary/secondary mode indicator and bit-1 of normal/anywhere accommodation indicator.

Mode_PS()
: – Mode_PS(M) examines primary/secondary mode indicator of M.

Mode_Acc()
: – Mode_Acc(M) examines normal/anywhere accommodation indicator of M.

Mode_Set_Pri()
: – Mode_Set_Pri(M) is to set mode indicator of M to primary.

Mode_Set_Sec()
: – Mode_Set_Sec(M) is to set mode indicator of M to secondary.

Mode_Set_Norm()
: – Mode_Set_Norm(M) is to set accommodation indicator of M to normal.

Mode_Set_Any()
: – Mode_Set_Any(M) is to set accommodation indicator of M to anywhere.

Mode_Is_Norm()
: – Mode_Is_Norm(M) is true iff M indicates normal accommodation including rebalancing.

Mode_Is_Any()
: – Mode_Is_Any(M) is true iff M indicates anywhere accommodation.

accMode
: • The integer variable accMode has 0 for normal accommodation or 1 for anywhere one given by oh1_transbound(). It is also initialized by init1() to be 0. The variable is referred to by oh1_accom_mode() to give its value to the caller of the function as the return value.

```
#ifndef EXTERN
#define EXTERN extern
#endif

/* Basic process configuration variables */
EXTERN int nOfNodes;
EXTERN int myRank;
EXTERN int RegionId[2], *SubdomainId;
#define MODE_NORM_PRI (0)
#define MODE_NORM_SEC (1)
#define MODE_REB_SEC  (-1)
#define MODE_ANY_PRI  (2)
#define MODE_ANY_SEC  (3)
#define Mode_PS(M)       (M&1)
#define Mode_Acc(M)      (M&2)
#define Mode_Set_Pri(M)  (M&2)
#define Mode_Set_Sec(M)  (M|1)
```

```
#define Mode_Set_Norm(M) (M&1)
#define Mode_Set_Any(M)  (M|2)
#define Mode_Is_Norm(M)  (M<2)
#define Mode_Is_Any(M)   (M>=2)
EXTERN int currMode, accMode;
```

4.2.4 Particle Histograms

The next variable group is for particle histograms. We have the followings to count the number of particles.

`nOfSpecies`

- The integer variable `nOfSpecies` has the number of *species* of particles. This number is not necessary to mean the *real* number of species, e.g., the number of variations of particle mass and charge. Instead, this variable must have the number of memory regions each of which accommodates particles of a species, as discussed in §3.2.1. This variable should be given by the simulator body through the argument `nspec` of `oh1_init()`, and is referred to in many functions. Hereafter the value of `nOfSpecies` is denoted by S.

`maxFraction`

- The integer variable `maxFraction` has the tolerance factor percentage. This variable should be given by the simulator body through the argument `maxfrac` of `oh1_init()`, and is referred to in `transbound1()` to calculate `nOfLocalPMax`. Hereafter the value of `maxFraction` is denoted by α.

`NOfPLocal`

- The element $[p][s][m]$ of the integer array `NOfPLocal`$[2][S][N]$ has the number of primary ($p = 0$) or secondary ($p = 1$) particles of species s residing in the subdomain m and accommodated by the local node. The simulator body should give the (double) pointer to the array through the argument `nphgram` of `oh1_init()`, or a pointer to `NULL` to allocate its body by `init1()`, and should set each element by counting particles before calling `oh1_transbound()` which then clears all elements in the array to 0 upon its return. This array is also referred to in many other functions. Hereafter `NOfPLocal` of the node n is denoted by $q(n)$.

`NOfPrimaries`

- The element $[p][s][m]$ of the integer array `NOfPrimaries`$[2][S][N]$ has the number of particles of species s in the local node's primary subdomain and accommodated by the node m as its primary ($p = 0$) or secondary ($p = 1$) particles. Since `NOfPrimaries`$[p][s][m]$ of the local node n is equal to $q(m)[p][s][n]$, `NOfPrimaries` is built by collecting $q(m)[*][*][n]$ using `MPI_Alltoall()` in `transbound1()`. This array is allocated by `init1()` and is referred to in `try_primary1()`, `schedule_particle_exchange()`, `sched_comm()`, `stats_primary_comm()`, `try_primary2()` and `move_to_sendbuf_primary()`.

`TotalPGlobal`

- The element $[m]$ of the 64-bit integer array `TotalPGlobal`$[N+1]$ has the system-wide total number of particles residing in the subdomain m, namely P_m. Since `TotalPGlobal`$[m]$ for $m < N$ is defined as

$$P_m = \texttt{TotalPGlobal}[m] = \sum_{k=0}^{N-1} \sum_{p\in\{0,1\}} \sum_{s=0}^{S-1} q(k)[p][s][m]$$

  the values of this array are calculated by `MPI_Allreduce()` in `transbound1()`. On the other hand, the element $[N]$ is used to detect non-neighboring particle transfer and is

non-zero if so, in `transbound1()`. This array is allocated by `init1()` and is referred to in `try_primary1()`, `try_stable1()`, `rebalance1()`, `push_heap()`, `remove_heap()`, `try_primary2()` and `move_to_sendbuf_primary()`.

`nOfParticles`

- The 64-bit integer variable `nOfParticles` has the total number of particles residing in the simulated space domain. Thus its value, denoted by P hereafter, is calculated by `transbound1()` such that $P = \sum_{m=0}^{N-1} P_m$. The variable is referred to in `rebalance1()`.

`nOfLocalPMax`

- The integer variable `nOfLocalPMax` have the maximum number of particles which a local node can accommodate. Its value $P_{\max}$ is calculated by `transbound1()` by;

$$P_{\max} = \lfloor P(100 + \alpha)/(100N) \rfloor$$

This variable is referred to in `try_primary1()`, `try_stable1()`, `assign_particles()` and `try_primary2()`.

`NOfPToStay`

- The element $[m]$ of the 64-bit integer array `NOfPToStay`$[N]$ has the number of particles residing in the subdomain m and accommodated by nodes responsible for m as their primary or secondary subdomains, excluding those injected in the subdomain by the nodes themselves. That is, `NOfPToStay`$[m]$ is defined as

$$\begin{aligned} q'(m)[0][s][m] &= q(m)[0][s][m] - q^{\mathrm{inj}}(m)[0][s] \\ q'(c)[0][s][\mathit{parent}(c)] &= q(c)[1][s][\mathit{parent}(c)] - q^{\mathrm{inj}}(c)[1][s] \\ \texttt{NOfPToStay}[m] &= \sum_{s=0}^{S-1} \left(q'(m)[0][s][m] + \sum_{c \in H(m)} q'(c)[1][s][m] \right) \end{aligned}$$

where $q^{\mathrm{inj}}(m)[p][s]$ is `InjectedParticles`$[p][s]$ of m, and $H(m)$ is the set of helpers of m, or $H(m) = \{c \,|\, \mathit{parent}(c) = m\}$ in other word. Therefore the values of this array are calculated by each local node and then gathered by `MPI_Allgather()` in `count_stay()`. Then its caller `try_stable1()` refers to it possibly decrementing the element $[\mathit{parent}(m)]$ if the node m has to throw a part of its secondary particles away. The array is allocated by `init1()`.

`TotalP`

- The element $[p][s]$ of the integer array `TotalP`$[2][S]$ has the number of primary ($p = 0$) or secondary ($p = 1$) particles of species s accommodated by the local node n. Setting its elements is done in `transbound1()` by copying corresponding elements from `TotalPNext[][]`. The function also allocates the body of the array on its first call and initializes its primary elements $[0][s]$ by $\sum_{m=0}^{N-1} q(n)[0][s][m]$ and clears secondary elements $[1][s]$ with 0 by `set_total_particles()`. This implies that `TotalP`$[p][s]$ does not the sum of $q(n)[p][s][m]$ for all m in the second and successive calls, because $q(n) =$ `NOfPLocal` may reflect particle injections and/or removals and thus may not represent the layout in particle buffer. The copying from `TotalPNext` is also done by the level-2 counterpart `transbound2()`, and the array is referred to in `move_to_sendbuf_primary()` and `move_to_sendbuf_secondary()`.

`TotalPNext`

- The element $[p][s]$ of the integer array `TotalPNext`$[2][S]$ has the value to be set to `TotalP`$[p][s]$ at the end of `transbound1()` or `transbound2()`. The values of this array are calculated by `try_primary1()` or `move_to_sendbuf_primary()` if the next mode is primary, or by `make_comm_count()`, `make_recv_count()`, `count_next_particles()`, and/or `move_to_sendbuf_secondary()` otherwise. This array is shown

to the simulator body, which gives the (double) pointer to it through `totalp` argument of `oh1_init()`, or the pointer to `NULL` to alllocate it by `init1()`, and thus works as the shadow of `TotalP`.

`primaryParts`

- The integer variable `primaryParts` is calculated by `set_total_particles()` on the first call of `oh1_transbound()` or an explicit call of `oh2_set_total_particles()` to have $Q_n^n = \sum_{s=0}^{S-1} \texttt{TotalP}[0][s]$, i.e., the number of primary particles which the local node n initially accommodates. After that, the variable is calculated by `try_primary2()` or `move_to_sendbuf_secondary()` to show the size/base of priamry/secondary particle buffer in the next step, so that it can be referred to by `move_to_sendbuf_secondary()` itself and `move_to_sendbuf_primary()`.

`totalParts`

- The integer variable `totalParts` is made equal to `primaryParts` by `set_total_particles()` in the first call of `transbound1()` or an explicit call of `oh2_set_total_particles()` to show the number of (primary) particles which the local node initially accommodates. After that, it is calculated at the end of `transbound2()` to show $Q_n = \sum_{p\in 0,1} \sum_{s=0}^{S-1} \texttt{TotalP}[p][s]$, i.e., the total number of particles which the local node currently accommodates, to the functions `move_to_sendbuf_secondary()`, `move_injected_to_sendbuf()` and `oh2_inject_particle()` which need to know the bottom of the particle buffer in the next simulation step.

`NOfRecv` `RecvCounts`

- The element $[p][s][m]$ of the integer array `NOfRecv`$[2][S][N]$ is calculated by `try_primary1()` if we will be in primary mode in the next step, or `make_comm_count()` and `make_recv_count()` otherwise, to have the number of primary ($p = 0$) or secondary ($p = 1$) particles of species s which the local node should received from the node m. The shadow of this array `RecvCounts` replicated by `transbound1()` is an API for the simulator body to notify it how particles are received, and thus the (double) pointer to the array, or a pointer to `NULL` to allocate its body by `init1()` as well as `NOfRecv`, is given through the argument `rcounts` of `oh1_init()`. We need the shadow because `NOfRecv` is referred to by `stats_secondary_comm()`.

`NOfSend` `SendCounts`

- The element $[p][s][m]$ of the integer array `NOfSend`$[2][S][N]$ is calculated by `try_primary1()` if we will be in primary mode in the next step, or `make_comm_count()`, `make_recv_count()` and `make_send_count()` otherwise, to have the number of particles of species s which the local node should send to the node m as its primary ($p = 0$) or secondary ($p = 1$) particles. The shadow of this array `SendCounts` replicated by `transbound1()` is an API for the simulator body to notify it how particles are sent, and thus the (double) pointer to the array, or a pointer to `NULL` to allocate its body by `init1()` as well as `NOfSend`, is given through the argument `scounts` of `oh1_init()`. We need the shadow because `NOfSend` is referred to by `stats_secondary_comm()`.

`InjectedParticles`

- The element of $[0][p][s]$ of the integer array `InjectedParticles`$[2][2][S]$ has $q^{\text{inj}}(n)[p][s]$ being the number of particles of species s injected by the local node n into its primary ($p = 0$) or secondary ($p = 1$) subdomain using `oh2_inject_particle()` or its higher level counterparts. That is, after allocated by `init1()` and cleared by it and `transbound2()` at its end, each element is incremented by `oh2_inject_particle()` if the injected particle is in the primary/secondary subdomain to have the number of them at the call of `transbound2()`. Then the elements in $[0][p][s]$ are at first referred by `count_stay()` so that injected particles are excluded from the staying primary/secondary particle count `NOfPToStay[]` so that they are considered to be floating. After that, the elements in $[0][p][s]$ are referred to by `set_sendbuf_disps()`

to keep the space for injected particles in SendBuf[], and, if we will be in primary mode in the next step, by move_injected_from_sendbuf() to move back injected primary particles into Particles[] from SendBuf[] into which move_injected_to_sendbuf() moved them. On the other hand if the next mode is secondary, the second half elements $[1][p][s]$ are set to the number of primary/secondary particles which are injected and stay in the local node by move_to_sendbuf_secondary(), and are referred to by move_injected_from_sendbuf() for moving back.

TempArray

- The integer array TempArray[N] is used for a temporay store in the functions count_stay(), assign_particles(), schedule_particle_exchange(), sched_comm(), rebalance1(), try_primary2() and exchange_particles(). The array is allocated by init1() which also uses it, or by its level-4p counterpart due to the necessity of a larger store.

T_Histogram

- The MPI_Datatype variable T_Histogram has the MPI data-type for a slice $[*][*][m]$ in a array of $[2][S][N]$ for MPI_Alltoall() communications to exchange histograms of particle amounts. The value of this variable is created by MPI_Type_vector() and MPI_Type_struct() called in init1(), and is used for MPI_Alltoall() in transbound1() and make_comm_count().

```
/* Number of particles and related variables */
EXTERN int  nOfSpecies;
EXTERN int  maxFraction;
EXTERN int  *NOfPLocal;                 /* [2][nOfSpecies][nOfNodes] */
EXTERN int  *NOfPrimaries;              /* [2][nOfSpecies][nOfNodes] */
EXTERN dint *TotalPGlobal;              /* [nOfNodes+1] */
EXTERN dint nOfParticles;
EXTERN int  nOfLocalPMax;
EXTERN dint *NOfPToStay;                /* [nOfNodes] */
EXTERN int  *TotalP;                    /* [2][nOfSpecies] */
EXTERN int  *TotalPNext;                /* [2][nOfSpecies] */
EXTERN int  primaryParts, totalParts;
EXTERN int  *NOfRecv, *RecvCounts;      /* [2][nOfSpecies][nOfNodes] */
EXTERN int  *NOfSend, *SendCounts;      /* [2][nOfSpecies][nOfNodes] */
EXTERN int  *InjectedParticles;         /* [2][2][nOfSpecies] */
EXTERN int  *TempArray;                 /* [nOfNodes] */
EXTERN MPI_Datatype T_Histogram;
```

4.2.5 Node Descriptors

S_node

The next variable group is for S_node type data structures to keep various information of each node (MPI process) for load balancing. The S_node structure has the following elements.

- stay.prime is set to the number of primary particles accommodated by the node, Q_n^n for the node n, by count_stay().
- stay.sec is set to the number of secondary particles accommodated by the node, $Q_n^{parent(n)}$ for the node n, by count_stay().

- `get.prime` has the maximum number of primary particles that the helpers of the node can accommodate further, or 0 if the node must get all the primary particles other than those can stay in the helpers, before the node is visited by the bottom-up traversal of the family tree in `try_stable1()`. That is, it has the following P_n^{put} for a node n.

$$P_m^{\min} = \begin{cases} P_m & H(m) = \emptyset \\ \max(0,\ P_m - \sum_{c \in H(m)} (P_{\max} - P_c^{\min})) & H(m) \neq \emptyset \end{cases}$$
$$Q_m^{\mathrm{get}} = P_{\max} - (P_m^{\min} + Q_m^{parent(m)})$$
$$P_n^{\mathrm{put}} = \sum_{m \in H(n)} \max(0, Q_m^{\mathrm{get}})$$

  Then, when the node is visited, it is set to the minimum number of primary particles the node must get from other nodes if positive, or the reversed maximum number of the node can put out to its helpers if negative. That is, it is set to the folllowing P_n^{get} for a node n.

$$Q_n^{\mathrm{stay}} = Q_n^n + \sum_{m \in H(n)} \min(Q_m^n,\ P_{\max} - P_m^{\min})$$
$$P_n^{\mathrm{get}} = \max(P_n - Q_n^{\mathrm{stay}} - P_n^{\mathrm{put}},\ -Q_n^n)$$

  Finally, when the node is visited again by the top-down traversal in `try_stable1()`, it is set to the exact number of the primary particles the node gets from others if positive, or the reversed one the node puts to its helpers otherwise.

- `get.sec` is set to the reversed number of secondary particles the node must put out in order to accommodate its primaries, or 0 otherwise, by the bottom-up traversal of the family tree in `try_stable1()`. That is, it is set to $\min(0, Q_n^{\mathrm{get}})$ for a node n. Then, when the helpand of the node is visited in the top-down traversal in `try_stable1()`, it is set to the exact number to get from other nodes if it was 0 meaning the node can accommodate some secondary particles further.

- `comm.prime` is set to the index of the MPI communicator array `Comms.body`$[N]$ for the family rooted by the node if it has helpers, or -1 otherwise, by `rebalance1()`.

- `comm.sec` is set to the index of the MPI communicator array `Comms.body`$[N]$ for the family rooted by the node if it has the helpand, or -1 otherwise, by `rebalance1()`.

- `comm.black` is set to 0 if the family rooted by the node is in *red* group, or 1 otherwise (i.e., *black* group), by `rebalance1()`.

- `comm.rank` is set to the MPI rank of the node in the communicator for the family rooted by the node by `rebalance1()`.

- `parent` is set to the pointer to the `S_node` structure for the helpand of the node, or `NULL` if the node is the root of the family-tree, by `rebalance1()`.

- `sibling` is set to the pointer to the `S_node` structure for the sibling in the family to which the node belongs as a helper, or `NULL` if the node is the last helper or is the root of the family tree, by `rebalance1()`.

- `child` is set to the pointer to the `S_node` structure for the first helper of the family rooted by the node, or `NULL` if the node is a leaf of the family tree, by `rebalance1()`.
- `id` is set to the MPI rank of the node by `init1()`.
- `parentid` is set to the MPI rank of the helpand of the node, or -1 if the node is the root of the family tree, by `rebalance1()`.

Then we have three array variables of `S_node` and its pointer types.

`Nodes`
- The element $[n]$ of the array `Nodes`$[N]$ has the `S_node` structure whose MPI rank is n. That is, `init1()` makes `Nodes`$[n]$`.id` $= n$ for all n when it allocates the array. This array is referred to by many functions.

`NodesNext`
- The array `NodesNext`$[N]$ temporaily has `Nodes[]` for the next simulation step when rebalancing is performed. This array, allocated by `init1()` and copied from `Nodes[]` by `rebalance1()`, is necessary because `schedule_particle_exchange()`, `sched_comm()` and `rebalance2()` access both family tree for the current (before rebalancing) and next (after rebalancing) steps. That is, the former is accessed to find neighboring and own family members who may accommodate particles residing the subdomain in question, while the latter is accessed to find new family menbers to whom those particles are distributed.

`NodeQueue`
- The array `NodeQueue`$[N]$ has the pointers to all `Nodes[]` elements in the order of a bottom-up traversal of the family tree. It is assured that for $0 \le \forall i < \forall j < N$, the node pointed by the element $[i]$ is not an ancestor of the node pointed by $[j]$. That is, the helpand of the node pointed by $[i]$ is pointed from some $[j]$ such that $i < j$. The array is allocated by `init1()`, while its elements are set by `rebalance1()` and referred to by `try_stable1()`.

```
/* Computation node descriptors */
struct S_node {
  struct {int prime, sec;} stay;
  struct {dint prime, sec;} get;
  struct {int prime, sec, black, rank;} comm;
  struct S_node *parent, *sibling, *child;
  int id, parentid;
};
EXTERN struct S_node *Nodes, *NodesNext, **NodeQueue;
```

4.2.6 Heap Structures for Rebalancing

`S_heap` The next variable group is for `S_heap` type data structures to keep subdomain ID's in ascending and descending order of particle populations in subdomains. The `S_heap` structure has the following elements.

- `n` has the number of elements registered in a heap.
- `node`$[N+1]$ has subdomain ID's. The element [1] is the root and has the ID of subdomain whose number of particles is minimum (maximum). For other element $[i]$ $(i > 1)$, it is assured that the number of particles in the corresponding subdomain is not less than (not greater than) that in the subdomain whose ID is registered in the

parent element $[\lfloor i/2 \rfloor]$. Note that the element [0] is never referred to and thus this element is not allocated.

- `index`$[N]$ has indices of `node[]`. The element $[n]$ has the index of `node[]` at which the subdomain ID n is registered, or 0 if the subdomain in not registered in the heap. That is, `node`[`index`$[n]] = n$ if `index`$[n] \neq 0$.

Then we have the following two `S_heap` variables, which are allocated by `init1()` and manipulated by `rebalance1()` directly or through `push_heap()`, `remove_heap()` and `pop_heap()`.

`LessHeap`

- The variable `LessHeap` is the `S_heap` structure to keep ID's of subdomains whose particle populations are less than average in ascending (minimum first) order.

`GreaterHeap`

- The variable `GreaterHeap` is the `S_heap` structure to keep ID's of subdomains whose particle populations are not less than average in descending (maximum first) order.

```
/* Heap structure for load rebalancing */
struct S_heap {
  int n, *node, *index;
};
EXTERN struct S_heap LessHeap, GreaterHeap;
```

4.2.7 Variables for Particle Transfer Scheduling

`S_commlist` The next variable group is for an array of `S_commlist` type structure named `CommList` and variables related to the array. Each elelemnt of the array represents a secondary-mode particle transfer which the local node or its family members have to perform. An array element of `S_commlist` type has the following integer elements whose values are set by `sched_comm()`.

- `sid` is the ID (MPI rank) of the node from which particles are transferred.
- `rid` is the ID (MPI rank) of the node to which particles are transferred.
- `region` is the ID of the subdomain in which the transferred particles reside.
- `count` is the number of particles to be transferred.
- `tag` has the number $pS + s$ to indicate the species s of the transferred particles and whether they are primary ($p = 0$) or secondary ($p = 1$) ones for the receiver. By using this element for the tag of MPI point-to-point communication, the receiver can recognize where the received particles should be placed in its particle store. Moreover, the tag can be used for the one-dimensional index of a (conceptually) two dimensional array of $[2][S]$.

Then we have the following variables, `CommList` array, arrays having indices of `CommList`, a pointer and size variable for a subarray of `CommList`, and an MPI data-type to transfer `S_commlist` data.

`CommList`

- The array `CommList`$[2 \cdot 3^D(NS + 1) + N(S + 3)]$ of `S_commlist` type is conceptually divided into following five blocks.

primary receiving block is built by each node for particles in its primary subdomain to be received by the node itself or its helpers. Its size is at most $N+NS$ because the block corresponds to a shortest path in a conceptual two-dimensional array of $|F(n)| \times \#s(n) \cdot S$, where $F(n) = H(n) \cup \{n\}$ is the set of family members for the subdomain n and $\#s(n)$ is the number of sender nodes which has particles in n, from its south-west corner to north-east corner and $|F(n)|$ and $\#s(n)$ are at most (but may be) N.

primary sending block is exchanged by neighboring node (subdomain) pairs. A node receives a part of the primary receiving block from each neighbor for particles sent from the family members rooted by the node to the family members rooted by the neighbor. The size of the block $B(n)$ for a node n is given by;

$$B(n) \leq \sum_{m \in nbor(n)} (|F(m)| + |F(n)|S) \leq \sum_{m \in nbor(n)} |F(m)| + (3^D - 1)|F(n)|S$$

where $nbor(n)$ is the set of nodes neighboring to n and thus $|nbor(n)| \leq 3^D - 1$. Since we have;

$$\sum_{m \in nbor(n)} |F(m)| + |F(n)| \leq 3^D + N \qquad |F(n)| \leq N$$

because the sets of helpers of n and its neighbors are exclusive each other, we can bound $B(n)$ as follows.

$$\begin{aligned} B(n) &\leq \sum_{m \in nbor(n)} |F(m)| + (3^D - 1)|F(n)|S \\ &\leq 3^D + N - |F(n)| + (3^D - 1)|F(n)|S \\ &\leq 3^D + (3^D - 1)NS \end{aligned}$$

secondary receiving block for a node is the copy of primary receiving block of its helpand which broadcasts the block to its helpers to show them particle receptions for its primary subdomain and thus helper's secondary subdmain. Therefore, the size of this block is at most $N + NS$.

secondary sending block for a node is the copy of primary sending block of its helpand which broadcasts the block to its helpers to show them particle transmissions for its primary subdomain and thus helper's secondary subdomain. Therefore, the size of this block is at most $3^D + (3^D - 1)NS$.

alternative secondary receiving block for a node is the copy of primary receiving block of its helpand which broadcasts the block to its helpers which become its family members by rebalancing. A node must refer to both of secondary receiving blocks gotten from its old and new helpand because the former may have particle transmissions for its old secondary subdomain. The size of this block is at most $N + NS$.

The array is allocated by `init1()`, while its elements are set by `sched_comm()`. The whole or a part of the array or its elements are referred to by `schedule_particle_exchange()`, `make_comm_count()`, `make_recv_count()`, `make_send_count()`, `count_next_particles()`, `exchange_particles()`, `receive_particles()` and `send_particles()`.

SecRList
SecRLSize

- The S_commlist type pointer SecRList points the head of the block of CommList for the secondary particle transfers, and integer variable SecRLSize has its size. The block is either of the secondary receiving block if we continue secondary mode without rebalancing, or the alternative secondary receiving block if rebalanced. The variables are set by make_comm_count() and referred to by rebalance2().

RLIndex

- The element k of the integer array RLIndex[3^D+1] has the index of CommList from which particle receptions from k-th neighbor are recorded. Therefore, we send records from CommList[RLIndex[k]] to CommList[RLIndex[$k+1$]-1] to k-th neighbor as a part of its primary sending block. This array is manipulated by schedule_particle_exchange() and sched_comm().

SLHeadTail

- The first element ([0]) of the integer array SLHeadTail[2] has the head index of the primary sending block of CommList, while its second element ([1]) has the tail index plus one, or the head of the secondary receiving block. The elements of this array are set by schedule_particle_exchange() and are referred to by make_comm_count() and try_stable2().

SecSLHeadTail

- The first element ([0]) of the integer array SecSLHeadTail[2] has the head index of the secondary sending block of CommList, while its second element ([1]) has the tail index plus one, or the head of the alternative secondary receiving block. Both indices are displacements from the head of secondary receiving block. The elements of this array are set by schedule_particle_exchange() and are referred to by make_comm_count() and try_stable2().

T_Commlist

- The MPI_Datatype type variable T_Commlist has the MPI data-type for a S_commlist type record, a contiguous data-type whose size is sizeof(struct S_commlist) in bytes . The value of this variable is created by MPI_Type_contiguous() called in init1(), and used for MPI communications in schedule_particle_exchange() and make_comm_count().

S_commsched_context

In addition, we have another struct for particle transfer named S_commsched_context to keep the execution context of the function sched_comm() with the following elements, which are intialized by the caller schedule_particle_exchange() and are updated and referred to by sched_comm().

- neighbor is the neighboring index of the node which sched_comm() is visiting as the root of a sender family.
- sender is the ID of the node which sched_comm() is examining its particles to be sent to the local node's family.
- spec is the particle species which sched_comm() is examining for sending particles from sender to the local node's family.
- dones is the number of particles of the species spec, which sched_comm() has already processed for sending from sender to the local node's family.
- donen is the number of particles which sched_comm() has already processed for sending from sender belonging to the local node's family.

```
/* Structured variables for particle transfer */
struct S_commlist {
```

```
  int sid, rid, region, count, tag;     /* tag = spec + nOfSpecies*sec */
};
EXTERN struct S_commlist *CommList, *SecRList;
EXTERN int RLIndex[OH_NEIGHBORS+1];
EXTERN int SLHeadTail[2], SecSLHeadTail[2], SecRLSize;
EXTERN MPI_Datatype T_Commlist;
struct S_commsched_context {
  int neighbor, sender, spec, comidx, dones, donen;
};
```

4.2.8 Variables for Family Communicators

The next variable group is for the MPI communicators of helpand-helper families.

GroupWorld

- The `MPI_Group` type variable `GroupWorld` has the group structure of MPI-processes belonging to `MPI_COMM_WORLD` (or whatever `MCW` refers to). It is initialized by `init1()` and is referred to by `rebalance1()` to extract processes to build communicators of families.

Comms

- The `struct` variable `Comms` has the following elements to store family communicators.
  - `n` has the number of family communicators, i.e., the number of non-leaf nodes in the family tree.
  - `body`$[N]$ has family communicators. More specifically its element $[i]$ has the communicator of the family rooted by the node in `NodeQueue`$[N-i-1]$.

  The elements above are initialized by `init1()`, and are updated and referred to by `rebalance1()`.

MyComm
MyCommC
S_mycommc

- The variable `MyComm` is the pointer to a `struct` named `S_mycommc` to have information of the family communicators for the local node with the following elements.
  - `prime` has the communicator of the family which the local node belongs to as the helpand, or `MPI_COMM_NULL` if it is a leaf.
  - `sec` has the communicator of the family which the local node belongs to as a helper, or `MPI_COMM_NULL` if it is the root.
  - `rank` is the MPI rank of the local node in the communicator `prime`, or -1 if it is a leaf.
  - `root` is the MPI rank of the local node's helpand in the communicator `sec`, or -1 if the local node is the root.
  - `black` indicates whether the communicator `prime` is in *red* group (0) or *black* group (1). The red (black) color is given to families rooted by nodes which belong to black (red) families as helpers. The coloring is necessary to perform collective communications in families without serialization. That is, we perform collective communications in black families at first in parallel, and then do in red families also in parellel.

  The structure is allocated by `init1()` and its elements are set by `rebalance1()`. The elements are referred to by `oh1_broadcast()`, `oh1_all_reduce()` and `oh1_reduce()`.

A C-coded simulator body may allocate a `S_mycommc` structure and give the pointer to the structure through the argument `mycomm` of `oh1_init()` which set it into `MyCommC`, or simply passes `NULL` through `mycomm` to show the unawareness of the structure. In the former case, `rebalance1()` copies the updated values of `MyComm` into `MyCommC` to make it the shadow. To make it possible for the simulator body to access the `S_mycommc` structure, the C header file ohhelp_c.h has the declaration of `S_mycommc` same as that in ohhelp1.h at its very beginning but just following;

```
#include <mpi.h>
```

to obtain the type declaration of `MPI_Comm`.

`MyCommF` `S_mycommf`

- The variable `MyCommF` is the pointer to a `struct` named `S_mycommf` being the Fortran counterpart of `MyComm`. The `S_mycommf` structure has elements same as `S_mycommc` but its `prime` and `sec` are not `MPI_Comm` type but integers. A Fortran-coded simulator body should allocate a `S_mycommf` structure, or more accurately, `oh_mycomm` type defined in oh_type.F90 and shown in §3.4.1, and give the pointer to it through the argument `mycomm` of `oh1_init_()`. The values of the structure are copied from `MyComm` with C-to-Fortran translation by `rebalance1()`.

```
/* Structured variables for MPI communicator */
EXTERN MPI_Group GroupWorld;
EXTERN struct {
  int n;
  MPI_Comm *body;       /* [nOfNodes] */
} Comms;
struct S_mycommc {
  MPI_Comm prime, sec;
  int rank, root, black;
};
EXTERN struct S_mycommc *MyComm, *MyCommC;
EXTERN struct S_mycommf {
  int prime, sec;
  int rank, root, black;
} *MyCommF;
```

In addition to the global variables shown above, the C source file ohhelp1.c has two global but private arrays of integers `FamIndex`$[N+1]$ and `FamMembers`$[2N-1]$ being the arguments of `oh1_families()` to report the configurations of all families to the simulator body, as discussed in §4.3.8.

4.2.9 Variables for Neighboring Information

`DstNeighbors` `SrcNeighbors` `Neighbors`

Next, we declare the integer arrays `DstNeighbors`$[3^D]$ and `SrcNeighbors`$[3^D]$ whose element $[k]$ has the k-th neighbor of the local node. More specifically, let k be as follows,

$$k = \sum_{d=0}^{D-1} \nu_d 3^d \quad (\nu_d \in \{0,1,2\})$$

and let $(\pi_0, \ldots, \pi_{D-1})$ be the coordinate of the local node in the conceptual D-dimensional coordinate system in which MPI processes (or equivalently their primary subdomains) are

laid out. The element `DstNeighbors`[k] basically has the MPI rank r of the process at $(\pi_0+\nu_0-1, \ldots, \pi_{D-1}+\nu_{D-1}-1)$, but must have $-(r+1)$ if there is $k' < k$ such that `DstNeighbors`[k'] $= r$. Note that the local node itself is in the element $k = 1+3+\cdots+3^{D-1}$ because $\nu_d = 1$ for all $d \in [0, D-1]$. Similarly, `SrcNeighbors`[3^D-1-k] has r or $-(r+1)$ for k defined above with $k' > k$. For both arrays, a special identifier $-(N+1)$ means that no MPI process is at the corresponding neighboring location. Therefore, when we perform a neighboring communication along the direction defined by k without multiple sending/receiving to/from an existing neighboring process, we send a data to `DstNeighbors`[k] if non-negative and receive a data from `SrcNeighbors`[k] if non-negative.

The contents of the arrays are initialized by `init1()` based on the neighboring information given by the simulator body through the argument `nbor`[3^D] of `oh1_init()`, or on the process grid size given by the argument `pcoord`[D]. In the latter case, `init1()` initialzes the array `nbor` simply with $(\pi_0+\nu_0-1, \ldots, \pi_{D-1}+\nu_{D-1}-1)$ assuming that $\pi_d \in [0, \Pi_d-1]$ where $\Pi_d =$ `pcoord[d]` and the rank r of the process at $(\pi_0, \ldots, \pi_{D-1})$ is defined as follows.

$$r_{D-1} = \pi_{D-1} \qquad r_d = r_{d+1}\Pi_d + \pi_d \qquad r = r_0$$

The elements of arrays are referred to by `schedule_particle_exchange()`, `sched_comm()`, `try_primary2()` and `init3()`.

The array `DstNeighbors`[3^D] is the element [0] of the array `Neighbors`[3][3^D] in reality. Its elements [1] and [2] are `DstNeighbors` for the helpand of the local node which broadcasts one of them in `build_new_comm()`. More specifically, the element [2] is used transitionally when we need both sets of neighbors of the helpands before ([1]) and after ([2]) rebalancing for position-aware particle management etc., while [1] is for non-transional use in `transbound1()` and `oh3_map_particle_to_neighbor()`, which also refer to [0].

```
/* Neighboring information */
EXTERN int Neighbors[3][OH_NEIGHBORS], SrcNeighbors[OH_NEIGHBORS];
  /* <BSW,BS,BSE,BW,B,BE,BNW,BN,BNE,    : 00..04..08
       SW, S, SE, W,O  E, NW, N, NE,    : 09..13..17
      TSW,TS,TSE,TW,T,TE,TNW,TN,TNE>    : 18..22..26 */
EXTERN int *DstNeighbors;
```

In addition to the global variables shown above, the C source file ohhelp1.c has two global but private arrays of integers `NeighborsShadow`[3][3^D] and `NeighborsTemp`[3^D] being the argument `nbor` of `oh1_neighbors()` and `init1()` respectively to report the neighbors of the local node's primary/secondary subdomains to the simulator body, as discussed in §4.3.3.

4.2.10 Variables for Statistics and Verbose Messaging

The next variable group is for statistics reporting and verbose messaging. Before explaning the group, we revisit the definition in the header file oh_stats.h explained in §3.10.1. The file `#define`'s integer keys to identify execution intervals whose execution times are measured. We name each key as `STATS_`*key* where *key* is a sequence of uppercase letters and underscores unique to the key but should not start with `PART_`. Each key must be `#define`'d as a unique integer in the range $[0, K_t-1]$ where K_t should be the definition of the special key `STATS_TIMINGS`.

The prototype of oh_stats.h defines the following keys.

`STATS_TRANSBOUND` for the interval from the beginning of `transbound1()`.

STATS_TRY_STABLE for the interval from the beginning of `try_stable1()`.

STATS_REBALANCE for the interval from the beginning of `rebalance1()`.

STATS_REB_COMM for the interval from the beginning of the family communicator creation in `rebalance1()`.

STATS_TB_MOVE for the particle packing in the particle store and move those to be sent to the send buffer in `move_to_sendbuf_primary()` or `move_to_sendbuf_secondary()`.

STATS_TB_COMM for particle transfer in `try_primary2()` or `exchange_particles()`.

The header file oh_stats.h also has the declaration and initialization of the array of character strings `StatsTimeStrings`$[2K_t]$ whose elements $[2k]$ and $[2k+1]$ are strings to be printed with the timing statistics of the execution intervals for the simulation of primary and secondary particles and/or subdomains identified by the key numbered k. The declaration is surrounded by C's macro construct `#ifdef␣OH_DEFINE_STATS` and `#endif` so that only ohhelp1.c includes this declaration. The array is referred to by `print_stats()`.

Now we come back to ohhelp1.h and `#define` following keys for the statistics of particle transfers.

STATS_PART_MOVE_PRI_MIN
STATS_PART_MOVE_PRI_MAX
STATS_PART_MOVE_PRI_AVE
STATS_PART_MOVE_SEC_MIN
STATS_PART_MOVE_SEC_MAX
STATS_PART_MOVE_SEC_AVE

- The keys `STATS_PART_MOVE_`x`_`y where $x \in \{$`PRI`, `SEC`$\}$ and $y \in \{$`MIN`, `MAX`, `AVE`$\}$ are for `MIN`nimum, `MAX`imum and `AVE`rage numbers of `PRI`mary and `SEC`ondary particles between a sender/receiver pair.

STATS_PART_GET_PRI_MIN
STATS_PART_GET_PRI_MAX
STATS_PART_GET_SEC_MIN
STATS_PART_GET_SEC_MAX

- The keys `STATS_PART_GET_`x`_`y where $x \in \{$`PRI`, `SEC`$\}$ and $y \in \{$`MIN`, `MAX`$\}$ are for `MIN`imum and `MAX`imum numbers of `PRI`mary and `SEC`ondary particles received by a node.

STATS_PART_PUT_PRI_MIN
STATS_PART_PUT_PRI_MAX
STATS_PART_PUT_SEC_MIN
STATS_PART_PUT_SEC_MAX

- The keys `STATS_PART_PUT_`x`_`y where $x \in \{$`PRI`, `SEC`$\}$ and $y \in \{$`MIN`, `MAX`$\}$ are for `MIN`imum and `MAX`imum numbers of `PRI`mary and `SEC`ondary particles sent by a node.

STATS_PART_PG_PRI_AVE
STATS_PART_PG_SEC_AVE

- The keys `STAS_PART_PG_`x`_AVE` where $x \in \{$`PRI`, `SEC`$\}$ are for average numbers of `PRI`mary and `SEC`ondary particles received (or sent equivalently) by a node.

STATS_PART_PRIMARY

- The key `STATS_PART_PRIMARY` is for the number of transitions to (or staying at) primary mode.

STATS_PART_SECONDARY

- The key `STATS_PART_SECONDARY` is for the number of transitions to (or staying at) secondary mode.

STATS_PARTS

- The macro constant `STATS_PARTS` is `#define`'d to be the number of keys of particle transfer statistics, K_p.

StatsPartStrings

Then we declare and initialize the array of character strings `StatsPartStrings`$[K_p]$ whose elements $[k]$ is the string to be printed with the particle transfer statistics identified by the key numbered k. The declaration is surrounded by C's macro construct `#ifdef␣OH_DEFINE_STATS` and `#endif` so that only ohhelp1.c includes this declaration. The array is referred to by `print_stats()`.

```
/* Structures and variables for statistics and verbose messaging */
#define STATS_PART_MOVE_PRI_MIN 0
#define STATS_PART_MOVE_PRI_MAX 1
#define STATS_PART_MOVE_PRI_AVE 2
#define STATS_PART_GET_PRI_MIN  3
#define STATS_PART_GET_PRI_MAX  4
#define STATS_PART_PUT_PRI_MIN  5
#define STATS_PART_PUT_PRI_MAX  6
#define STATS_PART_PG_PRI_AVE   7
#define STATS_PART_MOVE_SEC_MIN 8
#define STATS_PART_MOVE_SEC_MAX 9
#define STATS_PART_MOVE_SEC_AVE 10
#define STATS_PART_GET_SEC_MIN  11
#define STATS_PART_GET_SEC_MAX  12
#define STATS_PART_PUT_SEC_MIN  13
#define STATS_PART_PUT_SEC_MAX  14
#define STATS_PART_PG_SEC_AVE   15
#define STATS_PART_PRIMARY      16
#define STATS_PART_SECONDARY    17
#define STATS_PARTS             (STATS_PART_SECONDARY+1)

#ifdef OH_DEFINE_STATS
static char *StatsPartStrings[STATS_PARTS] = {
  "p2p transfer[pri,min]",
  "p2p transfer[pri,max]",
  "p2p transfer[pri,ave]",
  "get[pri,min]",
  "get[pri,max]",
  "put[pri,min]",
  "put[pri,max]",
  "put&get[pri,ave]",
  "p2p transfer[sec,min]",
  "p2p transfer[sec,max]",
  "p2p transfer[sec,ave]",
  "get[sec,min]",
  "get[sec,max]",
  "put[sec,min]",
  "put[sec,max]",
  "put&get[sec,ave]",
  "transition to pri",
  "transition to sec",
};
#endif
```

The next part is the sequence of the following `struct` declarations.

`S_statscurr`

- The structure `S_statscurr` is for the statistics of the currently executed simulation step and has the following elements.
  - `time.value` has the double-float wall-clock time at the call of `oh1_stats_time()` which starts an interval to be measured.
  - `time.val`$[2K_t+2]$ is a double-float array whose element $[2k+p]$ has the time spent in the interval identified by k for the simulation of primary ($p = 0$) or

secondary ($p = 1$) particles and/or subdomains. The elements $[2K_t]$ and $[2K_t+1]$ are special entries to eliminate the time spent from the statistics processing.

- `time.key` has the identifier of the interval currently measured.
- `time.ev`$[2K_t+2]$ is an integer array whose element $[2k+p]$ is 1 if and only if the interval identified by k for the simulation of primary ($p = 0$) or secondary ($p = 1$) particles and/or subdomains is executed.
- `part`$[K_p]$ is a 64-bit integer array whose element $[k]$ has the statistics count of the particle transfer identified by k.

The elements belonging to `time` are cleared by `oh1_init_stats()` and updated by `oh1_stats_time()` while the elements of `part` are set and modified by `stats_primary_comm()`, `stats_secondary_comm()` and `stats_comm()`. Then they are referred to and partly reinitialized by `update_stats()`.

`S_statstime`

- The structure `S_statstime` is for the timing statistics for each key and has following elements.
  - `min` has the double-float hitherto-minimum measured time.
  - `max` has the double-float hitherto-maximum measured time.
  - `total` has the double-float total of measured times.
  - `ev` has the number of executions of the measured intervals.

`S_statspart`

- The structure `S_statspart` is for the particle transfer statistics for each key and has following elements.
  - `min` has the 64-bit integer hitherto-minimum measured count.
  - `max` has the 64-bit integer hitherto-maximum measured count.
  - `total` has the 64-bit integer total of measured counts.

`S_statstotal`

- The structure `S_statstotal` has the following arrays of structures `S_statstime` and `S_statspart` structures for statistics keys.
  - `time`$[2K_t]$ is the array of `S_statstime` structures to keep timing statistics of the interval identified by k for the simulation of primary ($p = 0$) or secondary ($p = 1$) particles and/or subdomains in its element $[2k+p]$.
  - `part`$[K_p]$ is the array of `S_statspart` structures to keep particle transfer statistics identified by k in its element $[k]$.

The elements are initialized by `clear_stats()` and updated by `update_stats()`, while they are referred to by `print_stats()`.

`Stats` `S_stats`

- The variable `Stats` of `S_stats` structure has the following elements to keep measured statistics.
  - `curr` is a `S_statscurr` structure to keep the statistics measured in the most recent simulation time-step.
  - `subtotal` is a `S_statstotal` structure to keep the hitherto total statistics in the current time-steps for subtotal measurement.
  - `total` is a `S_statstotal` structure to keep the hitherto total statistics from the beginning of the simulation.

```
struct S_statscurr {
  struct {
    double value, val[2*STATS_TIMINGS+2];
    int key, ev[2*STATS_TIMINGS+2];
  } time;
  dint part[STATS_PARTS];
};
struct S_statstime {
  double min, max, total;
  int ev;
};
struct S_statspart {
  dint min, max, total;
};
struct S_statstotal {
  struct S_statstime time[2*STATS_TIMINGS];
  struct S_statspart part[STATS_PARTS];
};
EXTERN struct S_stats {
  struct S_statscurr curr;
  struct S_statstotal subtotal, total;
} Stats;
```

Finally, we declare a few variables related to staticstics and verbose messaging as follows.

`T_StatsTime`

- The `MPI_Datatype` variable `T_StatsTime` is for the MPI communication to reduce `S_statstime` statistics data. The value of this variable is created by `oh1_init_stats()` and used for `MPI_Reduce()` called in `print_stats()`.

`Op_StatsTime` `Op_StatsPart`

- The `MPI_Op` variables `Op_StatsTime` and `Op_StatsPart` are for the MPI communications to reduce `S_statstime` and `S_statspart` statistics data by the functions `stats_reduce_time()` and `stats_reduce_part()` respectively. The variables are initialized by `oh1_init_stats()` and used for `MPI_Reduce()` called in `update_stats()` and `print_stats()`.

`statsMode`

- If and only if the variable `statsMode` has a non-zero value (1 or 2, in a usual sense) statistics data are measured and reported. If it is 2, the reporting is repeated every r simulation steps where $r =$ `reportIteration`, while the report is made only at the end of simulation otherwise. The simulator body must gives its value through the argument `stats` of `oh1_init()` so that `init1()` copies it into `statsMode`. The variable is referred to in `transbound1()`, `oh1_init_stats[_]()`, `oh1_stats_time[_]()`, `oh1_show_stats[_]()`, `update_stats()`, `oh1_print_stats[_]()` and `transbound2()`.

`reportIteration`

- The variable `reportIteration` specifies the number of simulation steps at the every end of which the statistics are reported if `statsMode` = 2. The simulator body must gives its value through the argument `repiter` of `oh1_init()` so that `init1()` copies it into `reportIteration`. The variable is referred to in `oh1_show_stats()`.

`verboseMode`

- The variable `verboseMode` specifies the level of verbose execution as follows.
  - 0 means to execute silently.
  - 1 means to execute reasonably verbosely.

– 2 means to execute very verbosely.

– 3 (or larger) means to execute with very verbose messages from all processes.

The simulator body must gives the value of `verboseMode` through the argument `verbose` of `oh1_init()` so that `init1()` copies it into `verboseMode`. The variable is referred to in `transbound1()`, `try_primary1()`, `try_stable1()`, `rebalance1()`, `init1()` and `oh1_verbose[_]()` through the functional macro `Verbose()`, and in `vprint()` directly.

```
EXTERN MPI_Datatype T_StatsTime;
EXTERN MPI_Op Op_StatsTime, Op_StatsPart;
EXTERN int statsMode, reportIteration, verboseMode;
```

4.2.11 Function Prototypes

The next block is to declare function prototypes. First we declare the prototypes of the API function *pairs* each of which consists of API for Fortran and C. An API for Fortran has name ending with an underscore while its C counterpart is named by removing the tail underscore. The API functions are listed below.

- The function `oh1_init[_]()` initializes data strucutures of the level-1 library.
- The function `oh1_neighbors[_]()` is to specify the array into which the neighborhood information of the local node is given by the library.
- The function `oh1_families[_]()` is to specify the array pair into which the configuration of all family is given by the library.
- The function `oh1_transbound[_]()` examines whether particles distribution among nodes are balanced well and, if imbalanced, reconfigures helpand-helper relationships. Then it notifies the simulator body how the local node should receive and send particles through `RecvCounts` and `SendCounts`.
- The function `oh1_accom_mode[_]()` is to show its caller whether the particle accommodation mode is normal or anywhere by its return value.
- The functions `oh1_broadcast[_]()`, `oh1_all_reduce[_]()` and `oh1_reduce[_]()` performs collective communications in the families which the local node belongs to as the helpand and a helper.
- The function `oh1_init_stats[_]()` starts statistics measurement.
- The function `oh1_stats_time[_]()` declares the beginning of the interval whose execution time is measured.
- The function `oh1_show_stats[_]()` notifies the library of the end of a simulation step so as to let it update statistics with those measured in the step. It also reports the statistics if the step is at the end of iterations defined by `reportIteration` and `statsMode` = 2.
- The function `oh1_print_stats[_]()` notifies the library of the end of the simulation so as to let it report the statistics.

- The function `oh1_verbose[_]()` prints given message if `verboseMode` is not zero.

Before showing the source code for the prototypes, we show the first part of the header files ohhelp_c.h for C-coded simulators and ohhelp_f.h for Fortran-coded ones. At first these files `#include`'s oh_config.h to `#define` D = `OH_DIMENSION` and the constant `OH_LIB_LEVEL` for the default library level.

```
#include "oh_config.h"
```

Then they `#define` the aliases of level-1 API functions which do not have higher level counterparts.

```
#define oh_neighbors(A1) \
        oh1_neighbors(A1)
#define oh_families(A1,A2) \
        oh1_families(A1,A2)
#define oh_accom_mode() \
        oh1_accom_mode()
#define oh_broadcast(A1,A2,A3,A4,A5,A6) \
        oh1_broadcast(A1,A2,A3,A4,A5,A6)
#define oh_all_reduce(A1,A2,A3,A4,A5,A6,A7,A8) \
        oh1_all_reduce(A1,A2,A3,A4,A5,A6,A7,A8)
#define oh_reduce(A1,A2,A3,A4,A5,A6,A7,A8) \
        oh1_reduce(A1,A2,A3,A4,A5,A6,A7,A8)
#define oh_init_stats(A1,A2)    oh1_init_stats(A1,A2)
#define oh_stats_time(A1,A2)    oh1_stats_time(A1,A2)
#define oh_show_stats(A1,A2)    oh1_show_stats(A1,A2)
#define oh_print_stats(A1)      oh1_print_stats(A1)
#define oh_verbose(A1)          oh1_verbose(A1)
```

Then ohhelp_c.h gives the prototypes of the functions above, which are also given in ohhelp1.h.

```
void oh1_neighbors(int **nbor);
void oh1_families(int **famindex, int **members);
int  oh1_accom_mode();
void oh1_broadcast(void *pbuf, void *sbuf, int pcount, int scount,
                   MPI_Datatype ptype, MPI_Datatype stype);
void oh1_all_reduce(void *pbuf, void *sbuf, int pcount, int scount,
                    MPI_Datatype ptype, MPI_Datatype stype,
                    MPI_Op pop, MPI_Op sop);
void oh1_reduce(void *pbuf, void *sbuf, int pcount, int scount,
                MPI_Datatype ptype, MPI_Datatype stype,
                MPI_Op pop, MPI_Op sop);
void oh1_init_stats(int key, int ps);
void oh1_stats_time(int key, int ps);
void oh1_show_stats(int step, int currmode);
void oh1_print_stats(int nstep);
void oh1_verbose(char *message);
```

Then both headers `#define` the aliases level-1 specific API functions if `OH_LIB_LEVEL` is 1.

```
#if OH_LIB_LEVEL==1
```

```
#define oh_init(A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13) \
        oh1_init(A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13)
#define oh_transbound(A1,A2) oh1_transbound(A1,A2)
```

Finally, the prototypes of these functions are given in ohhelp_c.h and ohhelp1.h.

```
void oh1_init(int **sdid, int nspec, int maxfrac, int **nphgram,
              int **totalp, int **rcounts, int **scounts, void *mycomm,
              int **nbor, int *pcoord, int stats, int repiter, int verbose);
int  oh1_transbound(int currmode, int stats);
```

Oh the other hand, the prototypes of Fortran API functions are solely given in ohhelp1.h, while their Fortran versions are given in oh_mod1.F90 as shown in §3.4.

```
void oh1_neighbors_(int *nbor);
void oh1_families_(int *famindex, int *members);
int  oh1_accom_mode_();
void oh1_broadcast_(void *pbuf, void *sbuf, int *pcount, int *scount,
                    int *ptype, int *stype);
void oh1_all_reduce_(void *pbuf, void *sbuf, int *pcount, int *scount,
                     int *ptype, int *stype, int *pop, int *sop);
void oh1_reduce_(void *pbuf, void *sbuf, int *pcount, int *scount,
                 int *ptype, int *stype, int *pop, int *sop);
void oh1_init_stats_(int *key, int *ps);
void oh1_stats_time_(int *key, int *ps);
void oh1_show_stats_(int *step, int *currmode);
void oh1_print_stats_(int *nstep);
void oh1_verbose_(char *message);
void oh1_init_(int *sdid, int *nspec, int *maxfrac, int *nphgram,
               int *totalp, int *rcounts, int *scounts,
               struct S_mycommf *mycomm, int *nbor, int *pcoord, int *stats,
               int *repiter, int *verbose);
int  oh1_transbound_(int *currmode, int *stats);
```

Next we declare the prototypes of the following functions used in level-2 and level-3 libraries.

- The function `init1()` is the body of `oh1_init()`.
- The function `mem_alloc()` allocates a memory space by `malloc()`.
- The function `mem_alloc_error()` aborts the simulation due to the memory shortage reporting its cause.
- The function `errstop()` aborts the simulation due to an error detected by all processes reporting given error message.
- The function `local_errstop()` aborts the simulation due to an error detected by the local process reporting given error message.
- The function `set_total_particles()` is to initialize `TotalP`, `primaryParts` and `totalParts` with `NOfPLocal` upon the first call of `oh1_transbound()` or an explict call of `oh2_set_total_particles()`.
- The function `transbound1()` is the body of `oh1_transbound()`.

- The function `try_primary1()` is to examine whether particle distribution among subdomains is balanced well and thus we can perform the simulation in primary mode.
- The function `try_stable1()` is to examine whether particle distribution among nodes is balanced well and thus we can keep the current helpand-helper configuration.
- The function `rebalance1()` is to (re)build the helpand-helper configuration to cope with an unacceptable load imbalance.
- The function `build_new_comm()` is to build communicators for the helpand-helper families built by `rebalance1()`.
- The function `vprint()` prints a verbose message specified by its variable number of arguments.
- The function `dprint()` prints a message for debugging. This function is not used in the production version of the library.

```
/* Prototypes for the functions called from higher-level library code */
void  init1(int **sdid, int nspec, int maxfrac, int **nphgram,
            int **totalp, int **rcounts, int **scounts,
            struct S_mycommc *mycommc, struct S_mycommf *mycommf, int **nbor,
            int *pcoord, int stats, int repiter, int verbose);
void* mem_alloc(int esize, int count, char* varname);
void  mem_alloc_error(char* varname, size_t size);
void  errstop(char* format, ...);
void  local_errstop(char* format, ...);
void  set_total_particles();
int   transbound1(int currmode, int stats, int level);
int   try_primary1(int currmode, int level, int stats);
int   try_stable1(int currmode, int level, int stats);
void  rebalance1(int currmode, int level, int stats);
void  build_new_comm(int currmode, int level, int nbridx, int stats);
void  vprint(char* format, ...);
void  dprint(char* format, ...);
```

4.2.12 Macro `Verbose()`

`Verbose()` The last block `#define`'s a macro named `Verbose(L,VP)` for verbose messaging. This macro examines whether the given verbose level `L` conforms the level defined by `verboseMode`. That is, if `L` $\geq$ `verboseMode`, it prints a message using the expression given by `VP` which should be a call of `vprint()`, after performing global barrier synchronization by `MPI_Barrier()`. The printing is always done if the rank of the local node is 0, or `verboseMode` is 3 (or larger).

This macro is used in `init1()`, `transbound1()`, `try_primary1()`, `try_stable1()`, `rebalance1()` and `oh1_verbose[_]()`.

```
/* Macro for verbose messaging. */
#define Verbose(L,VP) {\
  if (verboseMode>=L) {\
```

```
    MPI_Barrier(MCW);\
    if (myRank==0 || verboseMode>=3) VP;\
  }\
}
```

4.3 C Source File ohhelp1.c

4.3.1 Header File Inclusion

The first job done in ohhelp1.c is the inclusion of the header file ohhelp1.h. Before the inclusion, we `#define` the macro `EXTERN` as empty so as to provide variables declared in ohhelp1.h with their *homes*, as discussed in §4.2.3.

`OH_DEFINE_STATS` We also `#define` the macro `OH_DEFINE_STATS` to have the private variable declarations of `StatsTimeStrings` and `StatsPartStrings` as discussed in §4.2.10.

```
#define EXTERN
#define OH_DEFINE_STATS
#include "ohhelp1.h"
```

4.3.2 Function Prototypes

The next and last job to do prior to function definitions is to declare the prototypes of the following functions private for the level-1 library.

- The function `count_stay()` counts the number of primary and secondary particles in each node.
- The function `assign_particles()` determines the number and destination of floating particles for a family.
- The function `compare_int()` compares two integers given by `qsort()`.
- The function `schedule_particle_exchange()` determines the particle transfer schedule for the family rooted by the local node.
- The function `count_real_stay()` counts the number of primary/secondary particles accommodated in the local node or its helper.
- The function `sched_comm()` determines the transfer schedule of the particle residing in and visiting to the primary subdomain of the local node.
- The function `make_comm_count()` gives particle counts to `TotalPNext`, `NOfRecv` and `NOfSend`.
- The function `make_recv_count()` counts particles for `NOfRecv`.
- The function `make_send_count()` counts particles for `NOfSend`.
- The function `count_next_particles()` counts particles for `TotalPNext`.
- The function `push_heap()` pushes an element to a heap structure.
- The function `pop_heap()` pops the top element from a heap structure.
- The function `remove_heap()` removes an element from a heap structure.
- The function `clear_stats()` clears statistics data in a `S_statstotal` structure.

- The function `stats_primary_comm()` calculates statistics data of the particle transfers in primary mode.
- The function `stats_secondary_comm()` calculates statistics data of the particle transfers in secondary mode.
- The function `stats_comm()` performs the particle transfer statistics calculation for `stats_primary_comm()` and `stats_secondary_comm()`
- The function `update_stats()` update statistics data in a `S_statstotal` structure.
- The function `stats_reduce_part()` performs pairwise reduction for the particle transfer statistics.
- The function `print_stats()` prints statitstics.
- The function `stats_reduce_time()` performs pairwise reduction for the timing statistics.

```
/* Prototypes for private functions. */
static void  count_stay();
static dint  assign_particles(dint npr, dint npt, struct S_node *ch, int incgp,
                              int *nget);
static int   compare_int(const void* x, const void* y);
static void  schedule_particle_exchange(int reb);
static int   count_real_stay(int *np);
static void  sched_comm(int toget, int rid, int tag, int reb,
                        struct S_commsched_context *context);
static void  make_comm_count(int currmode, int level, int reb, int oldparent,
                             int stats);
static void  make_recv_count(struct S_commlist* rlist, int rlsize);
static void  make_send_count(struct S_commlist* slist, int slsize);
static void  count_next_particles(struct S_commlist* rlist, int rlsize);
static void  push_heap(int id, struct S_heap* heap, int greater);
static int   pop_heap(struct S_heap* heap, int greater);
static void  remove_heap(struct S_heap* heap, int greater, int rem);
static void  clear_stats(struct S_statstotal *stotal);
static void  stats_primary_comm(int currmode);
static void  stats_secondary_comm(int currmode, int reb);
static void  stats_comm(int* nrecv, int* nsend, dint* scp, int ns);
static void  update_stats(struct S_statstotal *stotal, int step, int currmode);
static void  stats_reduce_part(void* inarg, void* ioarg, int* len,
                               MPI_Datatype* type);
static void  print_stats(struct S_statstotal *stotal, int cstep, int n);
static void  stats_reduce_time(void* inarg, void* ioarg, int* len,
                               MPI_Datatype* type);
```

4.3.3 `oh1_init()` and `init1()`

`oh1_init_()` `oh1_init()`

The API functions `oh1_init_()` for Fortran and `oh1_init()` for C receive a set of array/structure variables through which level-1 library functions communicate with the simulator body, and a few integer parameters to specify the behavior of the library. The functions have the following arguments.

- The argument `sdid` is the (double) pointer to a two-element integer array, which is referred to as `SubdomainId`[2] in the library funcions as the shadow of `RegionId`[2]. The array has the subdomain identifier of the local node's primary subdomain in [0], while the element [1] has that of the secondary subdomain. Since the subdomain identifier is equivalent to the MPI rank of the node responsible for the subdomain as its primary one, [0] is always the rank of the local node and [1] is that of its helpand unless the local node is the family tree root. The array is allocated by `init1()` if `sdid` points `NULL`, and then initialized by `init1()` while its element [1] is updataed by `rebalance1()`.

- The integer input argument `nspec` should have the number of *species* of particles, i.e. S. This number is not necessary to mean the *real* number of species, e.g., the number of variations of particle mass and charge. Instead, this variable must have the number of memory regions each of which accommodates particles of a species, as discussed in §3.2.1. The value of the argument is set into `nOfSpecies` by `init1()`.

- The integer input argument `maxfrac` should have the tolerance factor percentage. The value of the argument is set into $\alpha =$ `maxFraction` by `init1()`.

- The argument `nphgram` should be the (double) pointer to an integer array of [2][S][N] (or of [$2 \times S \times N$]) which is referred to as `NOfPLocal`[][][] in the library functions. Each time the simulator body calls `oh1_transbound()`, it should set the element [p][s][m] to the number of primary ($p = 0$) or secondary ($p = 1$) particles of species s residing in the subdomain m and accommodated by the local node. Then `oh1_transbound()` clears all elements to zero upon its return to the caller. The array itself is allocated by `init1()` if `nphgram` points `NULL`.

- The argument `totalp` should be the (double) pointer to an integer array of [2][S] (or of [$2 \times S$]) which is referred to as `TotalPNext`[][] in the library functions as the shadow of `TotalP`[][]. Each array element [p][s] is updated each time `oh1_transbound()` is called to notify the simulator body of the number of primary ($p = 0$) or secondary ($p = 1$) particles of species s which the local node will have to accommodate in the next simulation step. The array itself is allocated by `init1()` if `totalp` points `NULL`.

- The argument `rcounts` should be the (double) pointer to an integer array of [2][S][N] (or of [$2 \times S \times N$]) which is referred to as `RecvCounts`[][][] in the library functions as the shadow of `NOfRecv`[][]. Each array element [p][s][m] is updated each time `oh1_transbound()` is called to notify the simulator body of the number of primary ($p = 0$) or secondary ($p = 1$) particles of species s which the local node will have to receive from the node m. The array itself is allocated by `init1()` if `rcounts` points `NULL`.

- The argument `scounts` should be the (double) pointer to an integer array of [2][S][N] (or of [$2 \times S \times N$]) which is referred to as `SendCounts`[][][] in the library functions as the shadow of `NOfSend`[][]. Each array element [p][s][m] is updated each time `oh1_transbound()` is called to notify the simulator body of the number of particles of species s which the local node will have to send to the node m as m's primary ($p = 0$) or secondary ($p = 1$) ones. The array itself is allocated by `init1()` if `scounts` points `NULL`.

- The argument `mycomm` should be the pointer to a `S_mycommf` or `S_mycommc` structure, or `NULL` for C-coded simulators. The simulator body can be unaware of the contents of the structure if it only uses API funtions for collective communications in each

family such as `oh1_broadcast()`. If so, it is solely required to allocate the structure body, or C-coded body may be free from even the allocation by giving `NULL` through this argument. The Fortran API argument is referred to as `MyCommF` in the library functions, while C's counterpart is `MyCommC` which acts as the shadow of `MyComm`.

- The argument `nbor` should be the (double) pointer to an integer array of $[3^D]$, which the simulator body can fully specify to make element `nbor`$[k]$ have the MPI rank of a neighbor of the local node conceptually at $(\pi_0, \ldots, \pi_{D-1})$ in a D-dimensional integer coordinate system each grid point $(\pi'_d, \ldots, \pi'_{D-1})$ of which has a MPI process whose rank is $rank(\pi'_d, \ldots, \pi'_{D-1})$.

$$k = \sum_{d=0}^{D-1} \nu_d 3^d \quad (\nu_d \in \{0, 1, 2\})$$
$$\texttt{nbor}[k] = rank(\pi_0 + \nu_0 - 1,\ \ldots,\ \pi_{D-1} + \nu_{D-1} - 1)$$

  Note that $rank(\pi'_0, \ldots, \pi'_{D-1})$ can be -2 or less to indicated that the grid point $(\pi'_0, \ldots, \pi'_{D-1})$ has no processes.

  On the other hand, the simulator body may entrust the setup of the array elements to `init1()` either by giving the pointer to `NULL` through `nbor` or the pointer to an array of $[3^D]$ whose first element is -1. In this case, `init1()` consults the array of $[D]$ given through `pcoord` assuming that $r = rank(\pi_0, \ldots, \pi_{D-1})$ is given as follows where Π_d is the element $[d]$ of the array.

$$r_{D-1} = \pi_{D-1} \qquad r_d = r_{d+1}\Pi_d + \pi_d \qquad r = r_0$$

- The argument `pcoord` should be the pointer to an integer array of $[D]$ to describe the process coordinate space $\Pi_0 \times \cdots \times \Pi_{D-1}$ where Π_d is its element $[d]$, if the simulator body entrusts the initialization of the array specified by `nbor`.

- The integer input argument `stats` should have 0, 1 or 2 to specify the mode of statistics collection and reporting, and is set into `statsMode` by `init1()`. If and only if it is 1 or 2, statistics data are collected and measured. If it is 2, the reporting is repeated every r simulation steps where $r =$ `repiter`, while the report is made only at the end of simulation otherwise.

- The integer input argument `repiter` should have a non-negative number to specify the number of simulation steps at the every end of which statistics are reported if `stats` $= 2$. The value is set into `reportIteration` by `init1()`.

- The integer input argument `verbose` should have a number in $[0, 3]$ to specify the level of verbose execution as follows.

  - 0 means to execute silently.
  - 1 means to execute reasonably verbosely.
  - 2 means to execute very verbosely.
  - 3 (or larger) means to execute with very verbose messages from all processes.

  The value is set into `verboseMode` by `init1()`.

The API functions almost simply call `init1()` passing all given arguments to it except for the followings.

- `oh1_init_()` passes the pointers to `sdid`, `nphgram`, `totalp`, `rcounts`, `scounts` and `nbor` rather than themselves.
- `oh1_init_()` passes `mycomm` to `mycommf` of `init1()` while `NULL` is passed through `mycommc` of `init1()` to keep it from allocation of `MyCommC`.
- `oh1_init()` passes `mycomm` to `mycommc` of `init1()` while `NULL` is passed through `mycommf` of `init1()` telling it that the body of `MyCommF` is not required. It also *casts* the argument as `S_mycommc` pointer type, because `mycomm` is declared as a `void` pointer to allow the simulator body to be completely unaware of the structure.

```
void
oh1_init_(int *sdid, int *nspec, int *maxfrac, int *nphgram,
          int *totalp, int *rcounts, int *scounts, struct S_mycommf *mycomm,
          int *nbor, int *pcoord, int *stats, int *repiter, int *verbose) {
  init1(&sdid, *nspec, *maxfrac, &nphgram, &totalp, &rcounts, &scounts,
        NULL, mycomm, &nbor, pcoord, *stats, *repiter, *verbose);
}
void
oh1_init(int **sdid, int nspec, int maxfrac, int **nphgram,
         int **totalp, int **rcounts, int **scounts, void *mycomm,
         int **nbor, int *pcoord, int stats, int repiter, int verbose) {
  init1(sdid, nspec, maxfrac, nphgram, totalp, rcounts, scounts,
        (struct S_mycommc*)mycomm, NULL, nbor, pcoord, stats, repiter,
        verbose);
}
```

`NeighborsShadow` `NeighborsTemp`

Prior to give the definition of `init1()`, we have to declare two pointer variables `NeighborsShadow`[3][3^D] and `NeighborsTemp`[3^D] being global but private to ohhelp1.c for the communiations among `init1()`, `oh1_neighbors()` and `build_new_comm()`.

The former keeps what the argument `nbor` of `oh1_neighbors()` points, i.e, `*nbor`, so that `build_new_comm()` lets its elements [1][] have what the helpand of the local node have in [0][] after rebalancing, and [2][] have what the local node itself had in [1][] before rebalancing. As for the elements [0][], they should be consistent with `*nbor`[] of `init1()`. Therefore, `init1()`'s `*nbor` is kept in `NeighborsTemp` so that `oh1_neighbors()` makes `NeighborsShadow`[0][] consistent with `NeighborsTemp`[] by copying the elements unless the two pointers are equivalent, if `oh1_neighbors()` is called after `init1()` is called as recommended. Note that if `*nbor` was `NULL` upon the call of `init1()` requiring to allocate the neighborhood array, we allocate an array of [3][3^D] for `*nbor` instead of [3^D] so that `*nbor` can be passed to `oh1_neighbors()` without allocating two arrays.

If `oh1_neighbors()` is called before `init1()` is called, on the other hand, `init1()` notices this order reversal by `NeighborsShadow` $\neq$ `NULL` and initializes `NeighborsShadow`[0][] to make them consistent with its `*nbor`[] = `NeighborsTemp`[]. Note that if `*nbor` = `NULL` in this case, `init1()` allocates an array of [3^D] instead of [3][3^D] because `*nbor` given to the function is definitely different from that given to `oh1_neighbors()`. Also note that `oh1_neighbors()` notices the reversal by `NeighborsTemp` = `NULL` to delegate the initialization to `init1()`.

```
static int (*NeighborsShadow)[OH_NEIGHBORS] = NULL;
static int *NeighborsTemp = NULL;
```

`init1()` The function `init1()` implements the initialization for its caller API functions, `oh1_init_()` and `oh1_init()`, or part of that for its higher level counterparts `init2()` and `init3()`. The arguments of this function are almost same as those of `oh1_init()` but its `mycomm` is split into two arguments `mycommc` and `mycommf`, which are `NULL` if called from `oh1_init_()` or `oh1_init()` respectively.

```
void
init1(int **sdid, int nspec, int maxfrac, int **nphgram,
      int **totalp, int **rcounts, int **scounts, struct S_mycommc *mycommc,
      struct S_mycommf *mycommf, int **nbor, int *pcoord,
      int stats, int repiter, int verbose) {

  int nn, ns, me, i, s, clsize;
  int *nb = *nbor;
  int bl[2]={1,1};
  MPI_Datatype tmptype[2]={MPI_DATATYPE_NULL, MPI_UB};
  MPI_Aint disp[2]={0, sizeof(int)};
```

First, we obtain the size of `MPI_COMM_WORLD` and the local node's rank in it by `MPI_Comm_size()` and `MPI_Comm_rank()` to set them into `nOfNodes` $= N$ and `myRank`. Then, we initialize `currMode` to `MODE_NORM_PRI` and `accMode` to 0 because we are in primary mode and normal accommodataion at initial, and set the argument `verbose` into `verboseMode` and messaging verbosely. After that we set arguments into corresponding global variables as follows; `*sdid` into `SubdomainId` allocating its body if necessary; `nspec` into `nOfSpecies` $= S$; `maxfrac` into `maxFraction` $= \alpha$; `*nphgram` into `NOfPLocal` allocating its body if necessary; `*totalp` into `TotalPNext` allocating its body if necessary. The allocation of the argument array bodies are done by `mem_alloc()`. As for `SubdomainId`, its first element is set to `myRank` while the second element is initialized to -1 to indicate the local node has no helpand, together with its *substance* `RegionId`. On the other hand, `TotalP`, the substance of `TotalPNext`, is initialized to `NULL` to indicate it should be allocated and initialized by `transbound1()` on its first call. Finally, `NOfPLocal` is zero-cleared for the first particle counting.

```
  MPI_Comm_size(MCW, &nn);  nOfNodes = nn;
  MPI_Comm_rank(MCW, &me);  myRank = me;
  currMode = MODE_NORM_PRI;  accMode = 0;

  verboseMode = verbose;
  Verbose(1, vprint("oh_init"));

  if (!*sdid)  *sdid = (int*)mem_alloc(sizeof(int), 2, "SubdomainID");
  SubdomainId = *sdid;
  (*sdid)[0] = RegionId[0] = me;  (*sdid)[1] = RegionId[1] = -1;
  ns = nOfSpecies = nspec;
  maxFraction = maxfrac;

  if (!*nphgram)
```

```
    *nphgram = (int*)mem_alloc(sizeof(int), 2*ns*nn, "NOfPLocal");
  NOfPLocal = *nphgram;
  if (!*totalp)  *totalp = (int*)mem_alloc(sizeof(int), 2*ns, "TotalP");
  TotalPNext = *totalp;
  TotalP = NULL;
  for(i=0; i<2*ns*nn; i++)  NOfPLocal[i] = 0;
```

Next, we allocate `RecvCounts`$[2][S][N]$ and `SendCounts`$[2][S][N]$ by `mem_alloc()`, unless their corresponding arguments `rcounts` and `scounts` are `NULL` meaning that `init1()` is called from `init2()`, and if they point `NULL`. On the other hand, if `rcounts` and `scounts` point non-`NULL` pointers, the pointers are set into `RecvCounts` and `SendCounts`. In any cases, their substance `NOfRecv`$[2][S][N]$ and `NOfSend`$[2][S][N]$ are allocated.

```
  if (rcounts) {
    if (!*rcounts)
      *rcounts = (int*)mem_alloc(sizeof(int), 2*ns*nn, "RecvCounts");
    RecvCounts = *rcounts;
  }
  if (scounts) {
    if (!*scounts)
      *scounts = (int*)mem_alloc(sizeof(int), 2*ns*nn, "SendCounts");
    SendCounts = *scounts;
  }
  NOfRecv = (int*)mem_alloc(sizeof(int), 2*ns*nn, "NOfRecv");
  NOfSend = (int*)mem_alloc(sizeof(int), 2*ns*nn, "NOfSend");
```

Next we allocate the body of the following global variables for particle population locally used in the library, by `mem_alloc()`; `NOfPrimaries`$[2][S][N]$, `TotalPGlobal`$[N+1]$, `NOfPToStay`$[N]$, and `InjectedParticles`$[2][2][N]$. We also allocate `TempArray`$[N]$ unless `OH_POS_AWARE` is defined to mean it is allocated by level-4p initializer `init4p()` with larger amount.

We also define the MPI data-type for the communication of particle histograms, a slice $[2][S][1]$ of integer arrays of $[2][S][N]$ as follows. Since a slice is a strided vector having $2S$ elements separated by $N-1$ array elements, the basic type for the slice is constructed by `MPI_Type_vector()`. However, the slices should be arrayed contiguously for, e.g., `MPI_Alltoall()`, so that a slice $[*][*][n]$ is followed by $[*][*][n+1]$. Therefore, we need to use `MPI_Type_struct()` to create a two-element structure with the strided vector and `MPI_UB` to make the resulting type `T_Histogram` have the extent of `sizeof(int)`. Finally, we commit the use of the type by `MPI_Type_commit()`.

```
  NOfPrimaries = (int*) mem_alloc(sizeof(int),  2*ns*nn, "NOfPrimaries");
  TotalPGlobal = (dint*)mem_alloc(sizeof(dint), nn+1, "TotalPGlobal");
  NOfPToStay   = (dint*)mem_alloc(sizeof(dint), nn, "NOfPToStay");
  InjectedParticles = (int*)mem_alloc(sizeof(int), 4*ns, "InjectedParticles");
  for (s=0; s<ns*2; s++)  InjectedParticles[s] = 0;
#ifndef OH_POS_AWARE
  TempArray    = (int*) mem_alloc(sizeof(int),  nn, "TempArray");
#endif

  MPI_Type_vector(2*ns, 1, nn, MPI_INT, tmptype);
  MPI_Type_struct(2, bl, disp, tmptype, &T_Histogram);
  MPI_Type_commit(&T_Histogram);
```

Next, we allocate `Nodes`[N], `NodesNext`[N] and `NodeQueue`[N] by `mem_alloc()`. For each of `Nodes`[n], we give the constant n to its `id` element.

```
Nodes = (struct S_node*)mem_alloc(sizeof(struct S_node), nn, "Nodes");
NodesNext = (struct S_node*)mem_alloc(sizeof(struct S_node), nn,
                                      "NodesNext");
for (i=0; i<nn; i++) Nodes[i].id = i;
NodeQueue =(struct S_node**)mem_alloc(sizeof(struct S_node*), nn,
                                      "NodeQueue");
```

The next allocation with `mem_alloc()` is done for `LessHeap` and `GreaterHeap`. Although their `node` elements are indexed in the range of $[0, N]$, we only refer to the elements in $[1, N]$. Therefore, we allocate a memory space for $2N$ integers for each heap structure, N for `node` and the other N for `index`, and make the pointer `node` point its non-exsistent element [0] at one-element behind the allocated space.

```
LessHeap.node     = (int*)mem_alloc(sizeof(int), nn*2, "LessHeap") - 1;
LessHeap.index    = LessHeap.node + nn + 1;
GreaterHeap.node  = (int*)mem_alloc(sizeof(int), nn*2, "GreaterHeap") - 1;
GreaterHeap.index = GreaterHeap.node + nn + 1;
```

Next we allocate `CommList`[] which could have $2 \cdot 3^D(NS+1) + N(S+3)$ elemets as discussed in §4.2.7. The required size is, however, can be larger than it with position-aware particle management, that could need $(14+4S)N$ elements, when $D = 1$ and $S < 4$. Therfore we allocate `CommList`[] using the larger one by `mem_alloc()`. We also define the MPI data-type for its element, which is simply a `MPI_BYTE` sequence of `sizeof(struct␣S_commlist)`, by `MPI_Type_contiguous()`[28].

```
clsize = 2*OH_NEIGHBORS*(nn*ns+1)+nn*(ns+3);
if (clsize<(14+4*ns)*nn)  clsize = (14+4*ns)*nn;
CommList = (struct S_commlist*)mem_alloc(sizeof(struct S_commlist), clsize,
                                         "CommList");
MPI_Type_contiguous(sizeof(struct S_commlist), MPI_BYTE, &T_Commlist);
MPI_Type_commit(&T_Commlist);
```

Next we allocate `Comms.body`[N] and initialize `Comms.n` to be 0 because it has no communicators in it. Then, after obtaining the group corresponding to `MPI_COMM_WORLD` to be set into `GroupWorld` by `MPI_Comm_group()`, we initialize `MyComm` after allocating it by `mem_alloc()`. We initialize its `prime` and `sec` elements to be `MPI_COMM_NULL` and `rank`, `root` and `black` elements to be 0, so that even an accidental invocation of `oh1_broadcast()` or other collective communication functions before the first call of `oh1_transbound()` results in just no-operation rather than an error. Then if `mycommc` is not `NULL`, we copy `MyComm` into its body after setting `MyCommC` to be the pointer. The initialization of `MyCommF` = `mycommf` is similarly done but `MPI_COMM_NULL` for `prime` and `sec` elements is translated into its Fortran form by `MPI_Comm_c2f()`.

```
Comms.body = (MPI_Comm*)mem_alloc(sizeof(MPI_Comm), nn, "Comms");
Comms.n = 0;
```

[28] Because we ignore endian problem which could arise if an OhHelp'ed simulator were executed on a heterogeneous parallel system.

```
MPI_Comm_group(MCW, &GroupWorld);
MyComm = (struct S_mycommc*)mem_alloc(sizeof(struct S_mycommc), 1, "MyComm");
MyComm->prime = MyComm->sec  = MPI_COMM_NULL;
MyComm->rank  = MyComm->root = MyComm->black = 0;
if ((MyCommC=mycommc))  *mycommc = *MyComm;
if ((MyCommF=mycommf)) {
  MyCommF->prime = MyCommF->sec = MPI_Comm_c2f(MPI_COMM_NULL);
  MyCommF->rank = MyCommF->root = MyCommF->black = 0;
}
```

Next, if the argument `nbor` points `NULL`, we allocate an array of $[3^D]$ or $[3][3^D]$ by `mem_alloc()` and returns the pointer to it through `nbor`, according to `NeighborsShadow` $\neq$ `NULL` or not, i.e., `oh1_neighbors()` has been called beforehand or not. Then, if we allocate the array or the simulator body gives the array whose first element is -1, we initialize `nbor`$[k]$ by the followings where $\mathit{\Pi}_d =$ `pcoord`$[d]$ to specify $\mathit{\Pi}_0 \times \cdots \times \mathit{\Pi}_{D-1}$ integer coordinate space in which MPI processes for $\boldsymbol{\pi} = (\pi_0, \ldots, \pi_{D-1})$ are laid out with the rank $\mathit{rank}(\boldsymbol{\pi}) = \mathit{rank}(\pi_0, \ldots, \pi_{D-1})$ and the local node of rank n is at $\boldsymbol{\pi}(n)$, after checking if $N = \mathit{\Pi}_0 \times \cdots \times \mathit{\Pi}_{D-1}$ and abort the execution by `errstop()` if it is not satisfied.

$$
\begin{aligned}
r_{D-1}(\boldsymbol{\pi}) = \pi_{D-1} \qquad r_d(\boldsymbol{\pi}) &= r_{d+1}(\boldsymbol{\pi})\mathit{\Pi}_d + \pi_d \qquad \mathit{rank}(\boldsymbol{\pi}) = r_0(\boldsymbol{\pi}) \\
k &= \sum_{d=0}^{D-1} \nu_d 3^d \quad (\nu_d \in \{0,1,2\}) \\
\texttt{nbor}[k] &= \mathit{rank}(\boldsymbol{\pi}(n) + \boldsymbol{\nu} - (1, \ldots, 1))
\end{aligned}
$$

The implementation assumes that $D \leq 3$ and is comprehensive for the case of $D = 3$. However, by making $\nu_d = 0$, $\pi_d = 0$ and $\mathit{\Pi}_d = 1$ for all d s.t. $D \leq d < 3$, we can cope with the cases with $D < 3$.

```
if (!nb) {
  if (NeighborsShadow) {
    nb = *nbor = (int*)mem_alloc(sizeof(int), OH_NEIGHBORS, "Neighbors");
  } else {
    nb = *nbor = (int*)mem_alloc(sizeof(int), 3*OH_NEIGHBORS, "Neighbors");
  }
  nb[0] = -1;
}
if (nb[0]==-1) {
  int p=pcoord[0];
  int q=(OH_DIMENSION>1)?pcoord[1]:1, r=(OH_DIMENSION>2)?pcoord[2]:1;
  int j, k, l;
  int yplus=(OH_DIMENSION>1)?2:0, zplus=(OH_DIMENSION>2)?2:0;
  int xoff, yoff, zoff;
  if (nn!=p*q*r || p<0 || q<0 || r<0)
    errstop("<# of x-nodes>(%d) * <# of y-nodes>(%d) * <# of z-nodes>(%d) "
            "should be equal to <# of nodes>(%d)", p, q, r, nn);
  i = me % p;  j = (me/p) % q;  k = me / (p*q);
  for (l=0,zoff=-1; zoff<zplus; zoff++) {
    for (yoff=-1; yoff<yplus; yoff++) {
      for (xoff=-1; xoff<2; xoff++,l++) {
        nb[l] = (i+xoff+p)%p + (((j+yoff+q)%q) + ((k+zoff+r)%r)*q)*p;
      }
    }
  }
```

On the other hand, if the simulator body gives the array `nbor` setting its elements, we check its consistency. That is, we check inter-node consistency by sending k to the process `nbor`$[k]$ if non-negative and/or receivng it from `nbor`$[3^D-1-k]$ if non-negative to examine the received index is k, with `MPI_Sendrecv()`, `MPI_Send()` and `MPI_Recv()` depeding on the existance of neighbors. This consistency check may cause a deadlock but it is less harmful than occuring in later simulation phase.

```
} else {
  for (i=0; i<OH_NEIGHBORS; i++) {
    int n=nb[i], m=nb[(OH_NEIGHBORS-1)-i], k;
    MPI_Status st;
    if (m>=0) {
      if (n>=0)
        MPI_Sendrecv(&i, 1, MPI_INT, n, 0, &k, 1, MPI_INT, m, 0, MCW, &st);
      else
        MPI_Recv(&k, 1, MPI_INT, m, 0, MCW, &st);
      if (k!=i)
        local_errstop("rank-%d's %d-th neighbor rank-%d says "
                      "rank-%d is not %d-th neighbor but %d-th",
                      me, (OH_NEIGHBORS-1)-i, m, me, i, k);
    } else if (n>=0) {
      MPI_Send(&i, 1, MPI_INT, n, 0, MCW);
    }
  }
}
```

Now we have neighbor information of the local node in `*nbor`$[3^D]$ and let `NeighborsTemp` be the pointer for it so that `oh1_neighbors()` will refer to it afterward to make `NeighborsShadow`$[0][]$ consistent with `NeighborsTemp`$[]$. On the other hand, if `oh1_neighbors()` has been called beforehand to let `NeighborsShadow` $\neq$ `NULL` and two neighborhood arrays are different, we have to initialize `NeighborsShadow`$[0][]$ copying all elements in `*nbor`$[]$ into them.

```
NeighborsTemp = nb;
if (NeighborsShadow && nb!=(int*)NeighborsShadow)
  for (i=0; i<OH_NEIGHBORS; i++)  NeighborsShadow[0][i] = nb[i];
```

Now we initialize `DstNeighbors`$[k]$ = `Neighbors`$[0][k]$ and `SrcNeighbors`$[3^D-1-k]$ so that they have the followings where $\mu[k] =$ `nbor`$[k]$.

$$\texttt{DstNeighbors}[k] = \begin{cases} -(N+1) & \mu[k] < 0 \\ \mu[k] & \mu[k] \geq 0 \ \wedge \ \forall k' < k(\mu[k'] \neq \mu[k]) \\ -(\mu[k]+1) & \mu[k] \geq 0 \ \wedge \ \exists k' < k(\mu[k'] = \mu[k]) \end{cases}$$

$$\texttt{SrcNeighbors}[3^D-1-k] = \begin{cases} -(N+1) & \mu[k] < 0 \\ \mu[k] & \mu[k] \geq 0 \ \wedge \ \forall k' > k(\mu[k'] \neq \mu[k]) \\ -(\mu[k]+1) & \mu[k] \geq 0 \ \wedge \ \exists k' > k(\mu[k'] = \mu[k]) \end{cases}$$

For the occurence check whether there is k' such that $\mu[k'] = \mu[k]$, we use `TempArray`$[N]$. We make the bit-0 of the element of the array $[n]$ be 1 if and only if there is $k' < k$ such that $n = \mu[k']$ and make its bit-1 be 1 if and only if there is $k' > k$ such that $n = \mu[k']$, and look it up when we have k such that $n_d = \mu[k]$ and $n_s = \mu[3^D-1-k]$. That is, after

initializing `TempArray`$[m]$ to be 0 for all $m \in [0, N-1]$, we examine its elements $[n_d]$ and $[n_s]$ whenever we have k such that $n_d = \mu[k] \geq 0$ and $n_s = \mu[3^D-1-k] \geq 0$, and turn bit-0 and bit-1 of the elements to 1 respectively.

```
    DstNeighbors = Neighbors[0];
    for (i=0; i<nn; i++) TempArray[i] = 0;
    for (i=0; i<OH_NEIGHBORS; i++) {
      int dst=nb[i],  src=nb[(OH_NEIGHBORS-1)-i];
      if (dst<0)  DstNeighbors[i] = -(nn+1);
      else {
        DstNeighbors[i] = (TempArray[dst]&1) ? -(dst+1) : dst;
        TempArray[dst] |= 1;
      }
      if (src<0)
        SrcNeighbors[i] = -(nn+1);
      else {
        SrcNeighbors[i] = (TempArray[src]&2) ? -(src+1) : src;
        TempArray[src] |= 2;
      }
    }
```

Finally, we finish the function setting variables for statistics, `stats` into `statsMode` and `repiter` into `reportIteration`.

```
    statsMode = stats;
    reportIteration = repiter;
  }
```

4.3.4 mem_alloc()

`mem_alloc()` The function `mem_alloc()`, called from `init1()`, `transbound1()`, `init2()`, `init3()`, `init_subdomain_passively()` and `init_fields()`, allocates the memory region whose byte-size is specified by its arguments, namely $e \times c$ where $e =$ `esize` for the element size and $c =$ `count` for the number of elements, and returns the base pointer to the allocated region. The allocation is done by `malloc()` whose failure stops the execution by `mem_alloc_error()` which produces an error message containing the name of the variable to be allocated given by `varname` and the required byte-size $e \times c$.

```
  void*
  mem_alloc(int esize, int count, char* varname) {

    size_t size = (size_t)esize*(size_t)count;
    void* ptr = malloc(size);
    if (!ptr) mem_alloc_error(varname, size);
    return(ptr);
  }
```

4.3.5 mem_alloc_error()

`mem_alloc_error()` The function `mem_alloc_error()`, called from `mem_alloc()` and `oh2_max_local_particles()`, aborts the execution due to the memory shortage with an error message showing its cause given by the arguments, the variable name `varname` and the required byte-size `size`, by `errstop()`.

```
void
mem_alloc_error(char* varname, size_t size) {
  errstop("out of virtual memory for %s(%lld)", varname, size);
}
```

4.3.6 errstop() and local_errstop()

errstop() The funtion errstop(), called from init1(), mem_alloc_error(), try_stable1(), oh2_max_local_particles(), init_subdomain_actively(), init_subdomain_passively() and init_fields(), stops the execution gracefully by MPI_Finalize() and exit() after showing an error message given through its variable number arguments following format. The message printing is done solely by the node of rank 0 by vprintf() and fprintf() with macros for variable number arguments va_start() and va_end().

local_errstop() On the other hand, the funciton local_errstop() is for errors detected by the local process in init1(), transbound1(), sched_comm(), oh2_inject_particle() and init_subdomain_actively(). Therefore, the error message is printed by the local process itself, and the execution is stopped ungracefully by MPI_Abort().

```
void
errstop(char* format, ...) {
  va_list v;
  va_start(v, format);

  if (myRank==0) {
    vfprintf(stderr, format, v);
    fprintf(stderr, "\n");
  }
  va_end(v);
  MPI_Finalize();  exit(1);
}
void
local_errstop(char* format, ...) {
  va_list v;
  va_start(v, format);

  vfprintf(stderr, format, v);
  fprintf(stderr, "\n");
  va_end(v);
  MPI_Abort(MCW, 1);
}
```

4.3.7 oh1_neighbors()

oh1_neighbors_() oh1_neighbors() The API functions oh1_neighbors_() for Fortran and oh1_neighbors() for C provide a simulator body calling them with an access to neighbor information kept in the library through its argument nbor. The Fortran API simply calls its C counterpart to which it gives the pointer to its argument nbor.

```
void
oh1_neighbors_(int *nbor) {
  oh1_neighbors(&nbor);
}
void
oh1_neighbors(int **nbor) {
  int *nb = *nbor;
  int i;
```

First, if `*nbor` = `NULL` requiring the allocation of the neighborhood array, we allocate that of $[3][3^D]$ by `mem_alloc()` and *return* the pointer to it through `nbor`. Then, if `NeighborsTemp` $\neq$ `NULL` to mean `init1()` has already been called as expected but `*nbor` arguments of `init1()` and this function are different, we copy `NeighborsTemp`[] into `*nbor`[0][] to make them consistent. Othewise we leave `*nbor`[0][] unchanged because `init1()` will initialize them afterward referring to `NeighborsShadow` or, most usually, it has done for its `*nbor` being equivalent to `*nbor` of this function. Finally, we save `*nbor` into `NeighborsShadow` so that `build_new_comm()` will upadate elements in [1][] and [2][] and, if `init1()` has not been called yet, it will initialize those in [0][].

```
  if (!nb)
    nb = *nbor = (int*)mem_alloc(sizeof(int), 3*OH_NEIGHBORS, "Neighbors");
  if (NeighborsTemp && nb!=NeighborsTemp)
    for (i=0; i<OH_NEIGHBORS; i++)  nb[i] = NeighborsTemp[i];
  NeighborsShadow = (int(*)[OH_NEIGHBORS])nb;
}
```

4.3.8 `oh1_families()`

`FamIndex` `FamMembers`

Prior to give the definition of API functions `oh1_families[_]()`, we have to declare two pointer variables `FamIndex` and `FamMembers` being global but private to ohhelp1.c for the communiations among `oh1_families()`, `try_primary1()` and `build_new_comm()`. These pointers are made equivalent of the arrays pointed by `famindex` and `members` arguments of `oh1_families()` by the function so that other functions updates the arrays to show the family configuration to a simulator body through the arrays. More specifically, the element $[m]$ of `FamIndex`$[N{+}1]$ has the index i_m of the array `FamMembers`$[2N]$ whose elements $\{[j] \,|\, j \in [i_m, i_m{+}1)\}$ are the ranks of the members in the family whose helpand is m, which is always registered in `FamMembers`$[i_m]$. In addition `FamMembers`$[2N{-}1]$ has the rank of the root of the helpand-helper tree.

```
static int *FamIndex = NULL;
static int *FamMembers = NULL;
```

`oh1_families_()` `oh1_families()`

The API functions `oh1_families_()` for Fortran and `oh1_families()` for C provide a simulator body calling them with an access to family information kept in the library through its arguments `famindex` and `members`. The Fortran API simply calls its C counterpart to which it gives the pointers to its arguments `famindex` and `members`.

```
void
oh1_families_(int *famindex, int *members) {
  oh1_families(&famindex, &members);
```

```
}
void
oh1_families(int **famindex, int **members) {
  int *fidx = *famindex,  *fmem = *members;
  int nn, i;
```

First, we call `MPI_Comm_size()` to obtain N because `init1()` may have not been called yet and thus `nOfNodes` can be undefined, though very unlikely. Then if `*famindex` = `NULL` and/or `*members` = `NULL` requiring the allocation of both or either of the arrays for family configuration, we allocate those of $[N{+}1]$ and/or $[2N]$ by `mem_alloc()` and *return* the pointers to them through `famindex` and/or `members`. Next we initialize the arrays so that the elements $[m]$ of them commonly have m for all $m \in [0, N)$ because $F(m) = \{m\}$ for all m in the initial primary mode. We also initialize `*famindex`$[N] = N$ to make `*famindex`$[m{+}1]$ − `*famindex`$[m] = |F(m)| = 1$ for all $m \in [0, N)$ including $m = N - 1$. Finally, we save `*famindex` and `*members` into `FamIndex` and `FamMembers` so that `try_primary1()` and `build_new_comm()` refer to them for update.

```
  MPI_Comm_size(MCW, &nn);
  if (!fidx)
    fidx = *famindex = (int*)mem_alloc(sizeof(int), nn+1, "FamIndex");
  if (!fmem)
    fmem = *members = (int*)mem_alloc(sizeof(int), nn*2, "FamMembers");
  for (i=0; i<nn; i++)  fidx[i] = fmem[i] = i;
  fidx[nn] = nn;
  FamIndex = fidx;  FamMembers = fmem;
}
```

4.3.9 `set_total_particles()`

`set_total_particles()`

The function `set_total_particles()`, called from `transbound1()` and level-2 API functions `oh2_set_total_particles[_]()`, is to allocate `TotalP`$[p][s]$ by `mem_alloc()` if it is `NULL`, and to calculate `TotalP`$[p][s]$, `primaryParts` and `totalParts` as follows.

$$\texttt{TotalP}[0][s] = \sum_{m=0}^{N-1} q(n)[0][s][m]$$

$$\texttt{TotalP}[1][s] = \begin{cases} \sum_{m=0}^{N-1} q(n)[1][s][m] & \texttt{currMode} \bmod 2 \neq 0 \ \wedge \ \mathit{parent}(n) \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\texttt{primaryParts} = \sum_{s=0}^{S-1} \texttt{TotalP}[0][s]$$

$$\texttt{totalParts} = \texttt{primaryParts} + \sum_{s=0}^{S-1} \texttt{TotalP}[1][s]$$

The necessity of the function is based on the fact these substance variables are usually calculated by `transbound1()` and/or `transbound2()` but they are undefined at or before the first call of these functions. Therefore it is intended to call this function at or before the first call of `transbound1()` to let it know the very initial state of the particles in the local node, but the function is designed to work well in other occasions. The values calculated for `TotalP`$[p][s]$ is also stored in its shadow `TotalPNext`$[p][s]$ as the reasonable output of `oh2_set_total_particles()`.

```
void
set_total_particles() {
  int ns=nOfSpecies, nn=nOfNodes, nnns=nn*ns;
  int cm=(Mode_PS(currMode))&&(RegionId[1]>=0);
  int s, i, j, tpp, tps;

  if (!TotalP)  TotalP = (int*)mem_alloc(sizeof(int), 2*ns, "TotalP");
  primaryParts = 0;  totalParts = 0;
  for (s=0,j=0; s<ns; s++) {
    for (i=0,tpp=0,tps=0; i<nn; i++,j++) {
      tpp += NOfPLocal[j];  tps += NOfPLocal[nnns+j];
    }
    if (!cm)  tps = 0;
    TotalP[s] = TotalPNext[s] = tpp;  TotalP[ns+s] = TotalPNext[ns+s] = tps;
    primaryParts += tpp;  totalParts += tps;
  }
  totalParts += primaryParts;
}
```

4.3.10 `oh1_transbound()` and `transbound1()`

`oh1_transbound_()` `oh1_transbound()`

The API functions `oh1_transbound_()` for Fortran and `oh1_transbound()` for C provide a simulator body calling them with the core mechanism of level-1 library. The functions have the following arguments.

- The input argument `currmode` should be an integer in $\{0, 1\}$ to mean one of the followings[29].
  - 0 means we are in primary mode.
  - 1 means we are in secondary mode.
- The input argument `stats` is usually non-zero (just 1 is sufficient) to mean that statistics data will be collected if `statsMode` is non-zero. However, the simulator body can give 0 through this argument to disable statistics processing temporalily when it calls the function, for example, for the initial particle distribution.

The API functions simply call `transbound1()` and pass its return value in $\{-1, 0, 1\}$ to their callers to notify them that the next simulation step is performed in primary (0) or secondary mode with (-1) or without (1) (re)building the helpand-helper configuration. The call of `transbound1()`, however, needs an additional argument `level` being 1 to indicate that the function is called from level-1 API functions. This `level` argument can be 2 (or larger) for the call from functions in level-2 (or higher) library for which setting `NOfRecv`/`RecvCounts` and `NOfSend`/`SendCounts` are unnecessary because particles are transferred in the library.

```
int
oh1_transbound_(int *currmode, int *stats) {
```

[29] In earlier versions, we needed information more detailed than simply showing the current execution mode and thus this argument, but what we need in the current version is just the mode and it is stored in the global variable `currMode`. However we are keeping this almost unnecessary argument for backward compatibility, which also require us to check the consistency with `currMode` and to extract its bit-0 to obtain the mode.

```
  return(transbound1(*currmode, *stats, 1));
}
int
oh1_transbound(int currmode, int stats) {
  return(transbound1(currmode, stats, 1));
}
```

`transbound1()` When the function `transbound1()` is called, from `oh1_transbound_()` or `oh1_transbound()`, or higher level counterparts `transbound2()` or `transbound3()`, each element of the histogranm array `NOfPLocal`$[p][s][m] = q(n)[p][s][m]$ has the number of particles accommodated by the local node n for each $p \in \{0,1\}$, $s \in [0, S-1]$ and $d \in [0, N-1]$.

The first job of the function, besides starting time measurement and verbose messaging, is to call `set_total_particles()` if `TotalP` is `NULL` to indicate that it is the first call of the function and thus `TotalP`, `primaryParts` and `totalParts` should be initialized according to `NOfPLocal`. We also check if LSB (mode indicator), extraced by `Mode_PS()`, of the argument `currmode` is equal to that of `currMode` to confirm the simulator body and library agree on the execution mode and abort execution by `local_errstop()` unless they are equal[30].

```
int
transbound1(int currmode, int stats, int level) {
  int ret=MODE_NORM_SEC, nn=nOfNodes, ns=nOfSpecies, nnns=nn*ns, nnns2=2*nnns;
  int i, j, k, s, p, tp, tpn, *nbor;
  dint nofp;

  Verbose(1,vprint("oh_transbound"));
  if ((stats=statsMode&&stats)) oh1_stats_time(STATS_TRANSBOUND, 0);
  if (!TotalP)  set_total_particles();
  currmode = Mode_PS(currmode);
  if (currmode!=Mode_PS(currMode))
    local_errstop("currmode given to oh_transbound() does not match with "
                  "that the library maintains");
```

Next we calculate the followings to have temporary local values of `TotalPGlobal` for the local node n.

$$Q_n^m = \texttt{TotalPGlobal}[m] = \sum_{p\in\{0,1\}} \sum_{s=0}^{S-1} q(n)[p][s][m]$$

$$Q_n = \sum_{m=0}^{N-1} Q_n^m$$

$$nbor(p) = \{m \,|\, m \in \texttt{Neighbors}[p],\ m \geq 0\}$$

$$Q'_n = \sum_{p\in\{0,1\}} \sum_{m\in nbor(p)}^{N-1} \sum_{s=0}^{S-1} q(n)[p][s][m]$$

$$\texttt{TotalPGlobal}[N] = \begin{cases} 0 & Q_n = Q'_n \ \wedge\ \texttt{currMode} < 2 \\ 1 & \text{otherwise} \end{cases}$$

[30] We can simply ignore the argument `currmode` and use `currMode` instead, but checking the consistency could help to debug the simulator body.

That is, Q_n^m is the number of particles in the subdomain m and currently accomadate by the local node n which has Q_n particles in total. On the other hand, Q'_n is the total number of primary and secondary particles residing in the primary and secondary subdomains themselves or in their neighboring subdomains. Therefore, `TotalPGlobal`$[N] = 0$ if and only if the outgoing particles from the local node's primary or secondary subdomain will be transferred to their neighbors, providing that `Mode_Is_Norm()` for `currMode` is true to mean unnatural particle accommodation is not forced. We refer to this natural accommodation of particles as *normal accommodation*. Otherwise, i.e., if `TotalPGlobal`$[N] = 1$, the local node has particles which should be transferred some distant subdomains and is in the state of *anywhere accommodation*, or is forced to be considered so.

Then we gather $q(m)[p][s][n]$, i.e., `NOfPLocal`$[p][s][n]$ of node m to have `NOfPrimaries`$[p][s][m]$ of the local node n by `MPI_Alltoall()` using the MPI data-type `T_Histogram` for the slice `NOfPLocal`$[*][*][n]$[31]. We also obtain;

$$P_m = \texttt{TotalPGlobal}[m] = \sum_{k=0}^{N-1} Q_k^m$$

for all $m \in [0, N-1]$ by `MPI_Allreduce()` on `TotalPGlobal` which also give us;

$$\texttt{TotalPGlobal}[N] \begin{cases} = 0 & \forall m : Q_m = Q'_m \\ > 0 & \exists m : Q_m \neq Q'_m \end{cases}$$

to make bit-1 (accommodation indicator) of `currmode` indicate whether we have normal (0) or anywhere (1) accommodation globally.

```
  for (i=0; i<nn; i++) TotalPGlobal[i] = 0;
  for (p=0,j=0,tp=0,tpn=0; p<=currmode; p++) {
    for (s=0; s<ns; s++) {
      for (i=0; i<nn; i++,j++) {
        int np=NOfPLocal[j];
        TotalPGlobal[i] += np;  tp += np;
      }
    }
    for (i=0,nbor=Neighbors[p]; i<OH_NEIGHBORS; i++) {
      int n=nbor[i];
      if (n>=0)
        for (s=0,k=(p==0)?n:n+nnns; s<ns; s++,k+=nn)  tpn+= NOfPLocal[k];
    }
  }
  TotalPGlobal[nn] = (tp==tpn && Mode_Is_Norm(currMode)) ? 0 : 1;

  MPI_Alltoall(NOfPLocal, 1, T_Histogram, NOfPrimaries, 1, T_Histogram, MCW);
#ifndef INTEL_MPI_BUG_FIXED
  for (p=0,k=myRank; p<2; p++)  for (s=0; s<ns; s++,k+=nn)
    NOfPrimaries[k] = NOfPLocal[k];
#endif
  MPI_Allreduce(MPI_IN_PLACE, TotalPGlobal, nn+1, MPI_LONG_LONG_INT, MPI_SUM,
                MCW);
  if (TotalPGlobal[nn])  currmode = Mode_Set_Any(currmode);
```

[31] Since Intel MPI has a bug in `MPI_Alltoall()` with $12 \leq N \leq 16$ and the data type `T_Histogram` that `NOfPrimaries`$[p][s][n]$ of the local node n is not updated for $p > 0$ or $s > 0$, we have to copy `NOfPLocal`$[p][s][n]$ to it explicitly until the bug is fixed.

Then we calculate the followings.

$$P = \texttt{nOfParticles} = \sum_{m=0}^{N-1} P_m$$
$$P_{\max} = \texttt{nOfLocalPMax} = \lfloor P(100+\alpha)/(100N) \rfloor$$

We also let `accMode` have the accommodation mode according to `currmode` by `Mode_Is_Any()`.

```
for (i=0,nofp=0; i<nn; i++)  nofp += TotalPGlobal[i];
nOfParticles = nofp;
nOfLocalPMax = nofp*(maxFraction+100)/100/nn;
accMode = Mode_Is_Any(currmode) ? 1 : 0;
```

Then here is the heart of the balancing examination, but we skip it if `transbound1()` is called from level-2 or higher library, leaving the examination to level-2 counterparts of `try_primary1()`, `try_stable1()` and `rebalance1()`, i.e. `try_primary2()`, `try_stable2()` and `rebalance2()`, giving them `currmode` to show the current execution mode and accommodation type as the return value. Otherwise, we first call `try_primary1()` to examine if we can stay in or turn to primary mode. If so, the return value is set to `MODE_NORM_PRI` to indicate we will be in primary mode in the next step. Otherwise, if we have been in secondary mode, we check if the particle movement still allows to keep the helpand-helper configuration by `try_stable1()`. If this examination fails and thus we need reconfiguration, or we have been in primary mode and thus need to build the configuration from scratch, we call `rebalance1()` to do that setting the return value to `MODE_REB_SEC` to show we have new configuration, while the return value is `MODE_NORM_SEC`, being the initial default value of the local variable `ret`, if success.

```
if (level>1) return(currmode);
if (try_primary1(currmode, 1, stats))  ret = MODE_NORM_PRI;
else if (!Mode_PS(currmode) || !try_stable1(currmode, 1, stats)) {
  rebalance1(currmode, 1, stats);  ret = MODE_REB_SEC;
}
```

Finally, we clear `NOfPLocal[][][]` to give the histogram base for the next simulation step, and copy `NOfRecv[][][]` and `NOfSend[][][]` to their shadows `RecvCounts[][][]` and `SendCounts[][][]`, and also copy `TotalPNext[][]` to its substance `TotalP[][]`. Note that `NOfPLocal[1][][]` should be cleared even when we will be in primary mode in the next step, because at least they are referred to by the `rebalance1()` itself. Then we return to the simulator body with return value indicating that the mode in the next step is primary (0), secondary without reconfiguration (1), or secondary with reconfiguration (-1), also setting it into `currMode`.

```
for (i=0; i<nnns2; i++) {
  NOfPLocal[i] = 0;  RecvCounts[i] = NOfRecv[i];  SendCounts[i] = NOfSend[i];
}
for (s=0; s<ns*2; s++) TotalP[s] = TotalPNext[s];
return((currMode=ret));
}
```

4.3.11 try_primary1()

try_primary1()

The function try_primary1(), called from transbound1() and try_primary2() being the level-2 counterpart of this function, examines if we can stay in or turn to primary mode. If so, we set NOfRecv[][][] and NOfSend[][][] from NOfPrimaries[][][] and NOfPLocal[][][] respectively. The function has three arguments currmode, level and stats whose meanings are almost as same as those of transbound1(), but a little bit different from them in the following points; currmode has the particle accommodation type in its bit-1; and stats is the logical conjunction of statsMode and that given to transbound1().

```
int
try_primary1(int currmode, int level, int stats) {
  int nn=nOfNodes, ns=nOfSpecies, nnns=nn*ns, me=myRank, nlpmax=nOfLocalPMax;
  int i, j, s;
```

After verbose messaging, we perform the main job of this function to check if $P_m =$ TotalPGlobal$[m] \leq P_{\max} =$ nOfLocalPMax for all m. We continue the execution including particle transfer statistics calculation and another verbose messaging if it is satisfied, or return to the caller with FALSE to tell it to do try_stable1() and/or rebalance1().

```
  Verbose(2,vprint("try_primary(%s,%s)",
                   Mode_PS(currmode)?"secondary":"primary",
                   Mode_Acc(currmode)?"anywhere":"normal"));
  for (i=0; i<nn; i++) {
    if (TotalPGlobal[i]>nlpmax) return(FALSE);
  }
  if (stats) stats_primary_comm(currmode);
  Verbose(2,vprint("try_primary=TRUE"));
```

Then if the function is called from level-2 or higher library, we simply return to try_primary2() with TRUE to let it perform primary mode particle transfers, after setting RegionId[1] and its shadow SubdomainId[1] to -1 to indicate the local node does not have a secondary subdomain, and letting FamIndex$[m] =$ FamMembers$[m] = m$ for all $m \in [0, N)$ to represent $F(m) = \{m\}$ if the previous mode is secondary and oh1_families() has been called beforehand to make these pointers non-NULL.

```
  SubdomainId[1] = RegionId[1] = -1;
  if (Mode_PS(currmode) && FamIndex) {
    int *fidx = FamIndex,  *fmem = FamMembers;
    for (i=0; i<nn; i++)  fidx[i] = fmem[i] = i;
    fidx[nn] = nn;
  }
  if (level>1) return(TRUE);
```

Otherwise, we set the element values of NOfSend, NOfRecv and TotalPNext of the local node n as follows.

$$\begin{aligned}
\texttt{NOfSend}[0][s][m] &= \sum_{p\in\{0,1\}} q(n)[p][s][m] = \sum_{p\in\{0,1\}} \texttt{NOfPLocal}[p][s][m] \quad (m \neq n) \\
\texttt{NOfSend}[0][s][n] &= q(n)[1][s][n] = \texttt{NOfPLocal}[1][s][n] \\
\texttt{NOfSend}[1][s][m] &= 0
\end{aligned}$$

$$\texttt{NOfRecv}[0][s][m] = \sum_{p\in\{0,1\}} q(m)[p][s][n] = \sum_{p\in\{0,1\}} \texttt{NOfPrimaries}[p][s][m] \quad (m \neq n)$$
$$\texttt{NOfRecv}[0][s][n] = q(n)[1][s][n] = \texttt{NOfPrimaries}[1][s][n]$$
$$\texttt{NOfRecv}[1][s][m] = 0$$
$$\texttt{TotalPNext}[0][s] = \sum_{m=0}^{N-1} \sum_{p\in\{0,1\}} q(m)[p][s][n] = \sum_{m=0}^{N-1} \sum_{p\in\{0,1\}} \texttt{NOfPrimaries}[p][s][m]$$

The equations above mean that we simply send all particles accommodated by the local node to other nodes responsible for subdomains they reside, while also simply gather all particles residing in the local node's subdomain from other nodes. Note that we consider the particle amount $q(n)[1][s][n] = \texttt{NOfPLocal}[1][s][n] = \texttt{NOfPrimaries}[1][s][n]$ as transferred in the local node itself because it should require moves from the store of secondary particles to that of primary.

Then we return to the caller with `TRUE` to indicate that we will be in primary mode in the next simulation step.

```
    for (i=0,s=0; s<ns; s++) {
      int t = 0;
      for (j=0; j<nn; j++,i++) {
        NOfSend[i] = NOfPLocal[i] + NOfPLocal[i+nnns];
        t += (NOfRecv[i] = NOfPrimaries[i] + NOfPrimaries[i+nnns]);
        NOfSend[i+nnns] = NOfRecv[i+nnns] = 0;
        if (j==me) {
          NOfSend[i] -= NOfPLocal[i];  NOfRecv[i] -= NOfPrimaries[i];
        }
      }
      TotalPNext[s] = t;  TotalPNext[ns+s] = 0;
    }
    return(TRUE);
  }
```

4.3.12 Macro `Special_Pexc_Sched()`

`Special_Pexc_Sched()` The macro `Special_Pexc_Sched(LEVEL)`, used in `try_stable1()` and `rebalance1()`, is expanded to true iff the argument `LEVEL` is negative to mean that these functions are called from their higher-level counterparts which have their own particle exchange scheduling mechanism (for position-aware particle management) and thus those in the level-1 functions should be skipped.

```
  #define Special_Pexc_Sched(LEVEL) (LEVEL<0)
```

4.3.13 `try_stable1()`

`try_stable1()` The function `try_stable1()`, called from `transbound1()` and `try_stable2()` being the level-2 counterpart of this function, examines if the current helpand-helper configuration sustains the particle movements crossing subdomain boundaries which can bring intolerable load imbalance. If so, we make an all-to-all type particle transfer schedule to keep the balanced situation and set `NOfRecv[][][]` and `NOfSend[][][]` according to the schedule. The function has three arguments `currmode`, `level` and `stats` whose meanings are as same as those of `try_primary1()`.

```
int
try_stable1(int currmode, int level, int stats) {
  int nn=nOfNodes;
  int nlpmax=nOfLocalPMax;
  struct S_node *node, *ch;
  int i;
```

The first job of the function, besides starting time measurement and verbose messaging, is to call `count_stay()` to have the followings for all $n \in [0, N-1]$.

$$Q_n^n = \texttt{Nodes}[n]\texttt{.stay.prime} = \sum_{s=0}^{S-1} q(n)[0][s][n]$$

$$Q_n^{parent(n)} = \texttt{Nodes}[n]\texttt{.stay.sec} = \sum_{s=0}^{S-1} q(n)[1][s][parent(n)]$$

$$\texttt{NOfPToStay}[n] = \sum_{s=0}^{S-1} \left(q(n)[0][s][n] + \sum_{c \in H(n)} q(c)[1][s][n] \right)$$

$$\texttt{Nodes}[n]\texttt{.get.prime} = \texttt{Nodes}[n]\texttt{.get.sec} = 0$$

That is, $\texttt{Nodes}[n]\texttt{.stay.}\{\texttt{prime}, \texttt{sec}\}$ is set to the number of primary/secondary particles currently accommodated by the node n including those injected into primary/secondary subdomains, and $\texttt{NOfPToStay}[n]$ is set to the sum of these `stay`ing particles in the family of n (i.e., $F(n)$), while $\texttt{Nodes}[n]\texttt{.get.}\{\texttt{prime}, \texttt{sec}\}$ are intialized to 0.

```
  if (stats) oh1_stats_time(STATS_TRY_STABLE, 0);
  Verbose(2,vprint("try_stable"));
  count_stay();
```

Now we examine if we can keep the helpand-helper configuration by traversing the family tree in bottom-up (leaf-to-root) manner. The traversal is done by scanning $\texttt{NodeQueue}[i]$ for all $i \in [0, N-1]$ ascendingly because for any helpand-helper pair stored in $\texttt{NodeQueue}[i_p]$ and $\texttt{NodeQueue}[i_c]$ it is guaranteed that $i_p > i_c$. Thus we perform the followings for each node $\texttt{Nodes}[n] = \texttt{NodeQueue}[i]$.

(1) At the visit of n, $\texttt{Nodes}[n]\texttt{.get.prime}$ is 0 if n is a leaf node, or the sum of the *rooms* in n's helpers to which n can move its primary particles if required. That is, this value $\texttt{putprimemax} = P_n^{\text{put}}$ is defined as follows.

$$P_m^{\min} = \begin{cases} P_m & H(m) = \emptyset \\ \max(0,\ P_m - \sum_{c \in H(m)} (P_{\max} - P_c^{\min})) & H(m) \neq \emptyset \end{cases}$$

$$Q_m^{\text{get}} = P_{\max} - (P_m^{\min} + Q_m^n)$$

$$P_n^{\text{put}} = \sum_{m \in H(n)} \max(0, Q_m^{\text{get}})$$

(2) `NOfPToStay`$[n]$ has remained unchanged from its initial value if all helpers have some such rooms, i.e., $Q_m^{\text{get}} \geq 0$ for all $m \in H(n)$. However, if some helper m has no rooms but should put its secondary partiles out to other family members to make room for its own primary particles, `NOfPToStay`$[n]$ has been decremented by the number of the overflown particles to be put out, namely $Q_m^n - (P_{\max} - P_m^{\min})$. Therefore the current value of `NOfPToStay`$[n]$, namely Q_n^{stay} is defined as follows.

$$Q_n^{\text{stay}} = Q_n^n + \sum_{m \in H(n)} \min(Q_m^n,\ P_{\max} - P_m^{\min})$$

Since P_n = `TotalPGlobal`$[n]$ has the system-wide total number of particles in the subdomain n, `getprime` $= P_n - Q_n^{\text{stay}}$ is the number of particles n's family members have to get not only from non-family members but possibly from n's helpers having overflown secondary particles.

(3) We calculate `putprime` $= -P_n^{\text{get}} = \min(P_n^{\text{put}} - (P_n - Q_n^{\text{stay}}),\ Q_n^n)$ being the maximum number of primary particles that the node n can put out to its helpers if positive, or reversed miminum one that n have to get from its helpers if negative. Then we record P_n^{get} in `Nodes`$[n]$`.get.prime` so that we refer to it when we determine the real number of particles to be put or gotten afterward.

(4) We calculate the maximum *room* in n, namely `room` $= Q_n^{\text{get}} = P_{\max} - (Q_n^n + P_n^{\text{get}} + Q_n^{parent(n)})$ which is equivalent to the defition $Q_n^{\text{get}} = P_{\max} - (P_n^{\min} + Q_n^{parent(n)})$ because $Q_n^n + P_n^{\text{get}} = P_n^{\min}$ as proven as follows.

$$\begin{aligned}
P_n^{\text{get}} &= -\min(P_n^{\text{put}} - (P_n - Q_n^{\text{stay}}),\ Q_n^n) \\
&= \max((P_n - Q_n^{\text{stay}}) - P_n^{\text{put}},\ -Q_n^n) \\
&= \max((P_n - (Q_n^n + \sum_{m \in H(n)} \min(Q_m^n,\ P_{\max} - P_m^{\min}))) - P_n^{\text{put}},\ -Q_n^n)
\end{aligned}$$

$$P_n^{\text{get}} + Q_n^n = \max(P_n - \sum_{m \in H(n)} \min(Q_m^n,\ P_{\max} - P_m^{\min}) - P_n^{\text{put}},\ 0)$$

$$\begin{aligned}
&\sum_{m \in H(n)} \min(Q_m^n,\ P_{\max} - P_m^{\min}) + P_n^{\text{put}} \\
&= \sum_{m \in H(n)} \min(Q_m^n,\ P_{\max} - P_m^{\min}) + \sum_{m \in H(n)} \max(0, Q_m^{\text{get}}) \\
&= \sum_{m \in H(n)} \min(Q_m^n,\ P_{\max} - P_m^{\min}) + \sum_{m \in H(n)} \max(0, P_{\max} - P_m^{\min} - Q_m^n) \\
&= \sum_{m \in H(n)} \big(\min(Q_m^n,\ P_{\max} - P_m^{\min}) + \max(0,\ P_{\max} - P_m^{\min} - Q_m^n)\big) \\
&= \sum_{m \in H(n)} \big(\min(Q_m^n,\ P_{\max} - P_m^{\min}) + \max(Q_m^n, P_{\max} - P_m^{\min}) - Q_m^n\big) \\
&= \sum_{m \in H(n)} \big(Q_m^n + P_{\max} - P_m^{\min} - Q_m^n\big) \\
&= \sum_{m \in H(n)} \big(P_{\max} - P_m^{\min}\big)
\end{aligned}$$

$$P_n^{\text{get}} + Q_n^n = \max(P_n - \sum_{m \in H(n)} (P_{\max} - P_m^{\min}),\ 0) = P_n^{\min}$$

Note that $Q_n^{\rm get}$ can be negative even if $P_n^{\rm get}$ is negative to mean n can put out some particles to decrease the number of particles currently accommodating which can be larger than $P_{\rm max}$ by one or more of the following reasons[32].

(a) Some particles are injected into n's primary/secondary subdomain to make Q_n^n and/or $Q_n^{parent(n)}$ larger than those set in the previous call of `transbound1()`.

(b) Some particles are removed to make $P_{\rm max}$ less than that examined in the previous call of `transbound1()`.

(c) The position-aware particle management is in effect so that Q_n^n and/or $Q_n^{parent(n)}$ can be a little bit larger than those set in the previous call of `try_primary1()`, `try_stable1()` or `rebalance1()`.

On the other hand, $Q_n^{\rm get}$ can be positive with a positive $P_n^{\rm get}$ of course, because Q_n^n and/or $Q_n^{parent(n)}$ could be made small by the particle movement crossing subdomain boundaries.

(5) If $Q_n^{\rm get} \geq 0$, i.e., $Q_n^{\rm get} = \max(0, Q_n^{\rm get})$, we add it to `Nodes`[$parent(n)$]`.get.prime` = $P_{parent(n)}^{\rm put}$ to show the contribution from n.

(6) If $Q_n^{\rm get} < 0$, the node n has to put a number of secondary particles out, namely $-Q_n^{\rm get}$, to accommodate its floating primary particles. Thus we record the room $Q_n^{\rm get}$ itself (i.e., a negative number) into `Nodes`[n]`.get.sec` to indicate the negative amount to be gotten. Then we also add $Q_n^{\rm get}$ to `NOfPToStay`[k], or decrement it by $-Q_n^{\rm get}$ in other words, to state that the $-Q_n^{\rm get}$ secondary particles in the node n cannot stay in it and thus have to be put out. It is proven that this operation correctly contributes to $Q_k^{\rm stay}$ by $Q_{k,n}^{\rm stay} = \min(Q_n^k,\ P_{\rm max} - P_n^{\rm min})$, where $k = parent(n)$, as follows.

$$\begin{aligned} Q_{k,n}^{\rm stay} &= \begin{cases} Q_n^k & Q_n^{\rm get} \geq 0 \\ Q_n^k + Q_n^{\rm get} & Q_n^{\rm get} < 0 \end{cases} \\ &= Q_n^k + \min(Q_n^{\rm get}, 0) \\ &= Q_n^k + \min(P_{\rm max} - (P_n^{\rm min} + Q_n^k),\ 0) \\ &= \min(P_{\rm max} - P_n^{\rm min},\ Q_n^k) \\ &= \min(Q_n^k,\ P_{\rm max} - P_n^{\rm min}) \end{aligned}$$

Finally we check if $-Q_n^{\rm get} \leq Q_n^k =$ `Nodes`[n]`.stay.sec` meaning that the number of overflown secondary particles is not greater than that of secondary particles the node n has. If it does not hold meaning that the node n cannot accommodate primary particles the node should do, we stop the procedure returning `FALSE` to the caller `transbound1()` or `try_stable2()`. The equivalence of this inequality to that more comprehensible one, $P_{\rm max} \geq P_n^{\rm min}$ being the negation of the failure condition shown in §2.3, is proven as follows.

$$\begin{aligned} &Q_n^k + Q_n^{\rm get} = Q_n^k + P_{\rm max} - (P_n^{\rm min} + Q_n^k) = P_{\rm max} - P_n^{\rm min} \\ &\Rightarrow Q_n^k \geq -Q_n^{\rm get} \ \leftrightarrow\ P_{\rm max} \geq P_n^{\rm min} \end{aligned}$$

[32] The fix in v0.9.8 coped with the reason (a) by excluding injected particles from Q_n^m, but ignored (b) and (c) which we cannot cope with by adjusting Q_n^m.

Note that adding to `Nodes`[$parent(n)$]`.get.prime` in (3) and (4), and updating `NOfPToStay` [$parent(n)$] in (5) should not be done for the node n being the root of the family tree, which should be at the bottom of `NodeQueue`[]. However, checking the satisfaction of $-Q_n^{\rm get} \le Q_n^k$, the righthand side should be 0 for the root, in (5) must be done for the root as other non-root nodes.

```
for (i=0; ; i++) {                    /* bottom up traversal of node tree */
  int nid, stayprime, staysec;
  dint putprimemax, floating, putprime, room, getsec=0;
  struct S_node *parent;
  node = NodeQueue[i];
  nid = node->id;
  putprimemax = node->get.prime;      /* 0 for leaf, or max number of p's
                                         children can accommodate for non
                                         leaf */
  stayprime = node->stay.prime;
  staysec = node->stay.sec;
  parent = node->parent;
  floating = TotalPGlobal[nid] - NOfPToStay[nid];
  putprime = putprimemax - floating;
  if (putprime>stayprime)  putprime = stayprime;
  node->get.prime = -putprime;
  room = nlpmax - (stayprime + staysec - putprime);
  if (room<0) {                       /* have to put some secondaries to get
                                         primaries */
    node->get.sec = getsec = room;    /* getsec is negative to mean to put */
    if (room+staysec<0) return(FALSE);
                                      /* nlpmax<stay.prime+getprime */
  }
  if (!parent) break;
  if (getsec) {                       /* getsec is negative to mean to put
                                         and thus number of parant's to-stay
                                         is decremented to make its getprime
                                         larger in the result */
    NOfPToStay[node->parentid] += getsec;
  } else {                            /* getsec is 0 to mean the node has
                                         some room to get secondaries */
    parent->get.prime += room;
  }
}
Verbose(2,vprint("try_stable=TRUE"));
```

Now we have confirmed that the helpand-helper configuration can be kept. In the confirmation process above, `Nodes`[n]`.{stay,get}.{prime,sec}` have been set to the following numbers of particles for each $n \in [0, N-1]$

- `stay.prime` $= Q_n^n$ is the number of currently staying in primary subdomain (excluding secondary to primary movement).
- `stay.sec` $= Q_n^{parent(n)}$ is the number of currently staying in secondary subdomain (exculding primary to secondary movement).
- `get.prime` $= P_n^{\rm get}$ is the minmum number to get from other nodes if positive, or the reversed maximum number to put to helpers if negative.

- `get.sec` is the reversed number of secondary overflows if negative and thus $Q_n^{\rm get}$, or 0 otherwise.

The elelemnt `NOfPToStay`$[n] = Q_n^{\rm stay}$ has also been set to the number of particles in the subdomain n, which can stay in the family nodes rooted by n excluding those overflown from helpers. Now we perform a top-down (root-to-leaf) traversal of the family tree by scanning `NodeQueue`$[i]$ descendingly from `NodeQueue`$[N-1]$ for the root, skipping leaf nodes. We perform the followings for each non-leaf node `Nodes`$[n]$ = `NodeQueue`$[i]$.

(1) At the visit of n, $Q_n^{\rm get}$ = `Nodes`$[n]$`.get.sec` has been set to the real number of particles the node should get (if positive) from other nodes or put (if negative) to its helpand or sibling helpers. Thus we calculate the minimum number of particles which the node n must accommodate, namely `nproot` $= \rho_n$ by the following.

$$\begin{aligned}\rho_n &= \texttt{Nodes}[n]\texttt{.stay.prime} + \texttt{Nodes}[n]\texttt{.stay.sec} + \texttt{Nodes}[n]\texttt{.get.sec}\\ &= Q_n^n + Q_n^{parent(n)} + Q_n^{\rm get} = Q_n^n + R_n\end{aligned}$$

Note that this number does not include positive `Nodes`$[n]$`.get.prime` $= P_n^{\rm get}$ and thus may be less than real must, but it is assured that $\rho_n + P_n^{\rm get} \le P_{\max}$ by the calculation of $Q_n^{\rm get}$.

(2) ρ_n can be greater than $P_{\max}$ with negative $P_n^{\rm get}$. This means that the node has to put a number of particles namely $\rho_n - P_{\max}$ which is assuredly not greater than $-P_n^{\rm get}$. Thus we set the reverse of this difference namely $R_n^{\rm get} = P_{\max} - \rho_n$ into `Nodes`$[n]$`.get.prime` to indicate the negative amount of particles to be gotten. We also calculate `floating` $= R_n^{\rm flt}$ by the following to have the number of floating particles in the family including the movement from the node n to its chldren.

$$\begin{aligned}R_n^{\rm flt} &= P_n - Q_n^{\rm stay} + \rho_n - P_{\max}\\ &= P_n - \Big(Q_n^n + \sum_{m\in H(n)} \min(Q_m^n,\ P_{\max} - P_m^{\min})\Big) + (Q_n^n + R_n) - P_{\max}\\ &= \Big(\sum_{k\notin F(n)} Q_k^n + \sum_{m\in H(n)} Q_m^n + Q_n^n\Big) - \Big(Q_n^n + \sum_{m\in H(n)} \min(Q_m^n,\ P_{\max} - P_m^{\min})\Big)\\ &\qquad + (Q_n^n + R_n - P_{\max})\\ &= \sum_{k\notin F(n)} Q_k^n + \sum_{m\in H(n)} \max(0,\ Q_m^n + P_m^{\min} - P_{\max}) + (Q_n^n + R_n - P_{\max})\\ &= \sum_{k\notin F(n)} Q_k^n + \sum_{m\in H(n)} \max(0,\ Q_m^n + P_m^{\min} - P_{\max}) + \max(0,\ Q_n^n + R_n - P_{\max})\\ &\qquad (\rho_n = Q_n^n + R_n > P_{\max})\end{aligned}$$

We also do $\rho_n \leftarrow P_{\max}$ to minimize the number of particle movement from the node to its helpers.

(3) If $\rho_n \le P_{\max}$, the node n has some room to get primary particles. In this case, the number of floating particles $R_n^{\rm flt}$ is simply;

$$\begin{aligned}R_n^{\rm flt} &= P_n - Q_n^{\rm stay} = \sum_{k\notin F(n)} Q_k^n + \sum_{m\in H(n)} \max(0,\ Q_m^n + P_m^{\min} - P_{\max})\\ &= \sum_{k\notin F(n)} Q_k^n + \sum_{m\in H(n)} \max(0,\ Q_m^n + P_m^{\min} - P_{\max}) + \max(0,\ Q_n^n + R_n - P_{\max})\\ &\qquad (\rho_n = Q_n^n + R_n \le P_{\max})\end{aligned}$$

and we initialize $R_n^{\rm get}$ = `Nodes`$[n]$`.get.prime` $= 0$ because the node may not get anything if it is heavily loaded and thus the oringinal `Nodes`$[n]$`.get.prime` $= P_n^{\rm get}$ is negative. In fact, in an extreme case with $R_n^{\rm flt} = 0$, we may skip the following because $R_n^{\rm get}$ is set to 0 correctly and it is assured that $Q_m^{\rm get}$ is 0 for any n's helper nodes m.

(4) We call `assign_particles()` with ρ_n, positive $R_n^{\rm flt}$, the head of helpers of the node n and the forth argument `incgp` $= 0$ meaing we try to keep helpers from putting their own primary particles to *grand-helpers*. If it returns a positive number `nptotal` $= T_n$, it successfuly found that floating particles can be assigned to k_n lightly loaded nodes in the family and the well balancing will be achieved by letting them have T_n particles in total. Otherwise, we have to call `assign_particles()` again setting `incgp` $= 1$ but keeping other arguments same to have T_n and k_n allowing movements of helpers to grand-helpers.

(5) Now we have the set of lightly loaded nodes $F_l(n) \subseteq F(n)$ in the family and $k_n = |F_l(n)|$. For each node $m \in F(n)$, we define its *base-loads* B_m as follows.

- $B_n = \rho_n = \min(P_{\max},\ Q_n^n + R_n)$.
- If we called `assign_particles()` twice and thus `incgp` $= 1$, for all $m \in H(n)$;

$$\begin{aligned} B_m &= Q_m^m + Q_m^n + \min(0, Q_m^{\rm get}) + P_m^{\rm get} \\ &= (Q_m^m + P_m^{\rm get}) + Q_m^n + \min(0,\ P_{\max} - (P_m^{\min} + Q_m^n)) \\ &= P_m^{\min} + \min(Q_m^n,\ P_{\max} - P_m^{\min}) \\ &= \min(Q_m^n + P_m^{\min},\ P_{\max}) \end{aligned}$$

- If we called `assign_particles()` only once and thus `incgp` $= 0$;
  - For all $m \in \{m \in H(n) \,|\, P_m^{\rm get} \geq 0\}$;

$$B_m = Q_m^m + Q_m^n + \min(0, Q_m^{\rm get}) + P_m^{\rm get} = \min(Q_m^n + P_m^{\min},\ P_{\max})$$

  - For all $\min\{m \in H(n) \,|\, P_m^{\rm get} < 0\}$;

$$B_m = Q_m^m + Q_m^n + \min(0, Q_m^{\rm get}) = Q_m^m + \min(Q_m^n,\ P_{\max} - P_m^{\min})$$

    Note that, since $P_m^{\rm get} < 0$ means $Q_m^m > P_m^{\min}$, $B_m = Q_m^m + Q_m^n$ if $B_m \leq P_{\max}$.

That is, B_m for $m \in H(n)$ is the sum of `stay.prime`, `stay.sec` and `get.sec` of `Nodes`$[m]$, possibly adding its `get.prime` if positive or `incgp` $= 1$. Note that the definitions above assure that a node $m \in F(n)$ really has room to have $P_{\max} - B_m$ particles if $B_m < P_{\max}$. We also know the followings.

$$T_n = R_n^{\rm flt} + \sum_{l \in F_l(n)} B_l \qquad \forall h \notin F_l(n) : T_n \leq k_n B_h \qquad T_n \leq k_n P_{\max}$$
$$\forall l \in F_l(n) : B_l < P_{\max}$$

In other words, $T_n/k_n > B_l$ only for all $l \in F_l(n)$ and we can achieve a good balancing by making the node l get particles of $T_n/k_n - B_l$.

However, since T_n/k_n is not necessary to be an integer, we have to make some rounding as follows. Let `npave` $= q_a = \lceil (T_n/k_n) - 1 \rceil$ and `npfac` $= q_f = T_n - q_a k_n$ which should be in $[1, k_n]$. Since $T_n/k_n > B_l$ is equivalent $q_a \geq B_l$, we can find nodes in $F_l(n)$ by this condition. Since some nodes, q_f nodes more specfically, should have $q_a + 1$ particles but it should be greater than neither any B_h such that $h \notin F_l(n)$ nor $P_{\max}$, we may allocate this slightly heavier load to arbitrary nodes in $F_l(n)$ chosen by the scanning order from the node n and through the sibling chain. Therefore, we let;

- $R_n^{\text{get}} = \texttt{Nodes}[n]\texttt{.get.prime} = q_a + 1 - B_n$ if $n \in F_l(n)$
- $Q_l^{\text{get}} = \texttt{Nodes}[l]\texttt{.get.sec} = q_a + 1 - B_l$ for first q_f or $q_f - 1$ nodes $l \in F_l(n)$, and
- $Q_l^{\text{get}} = \texttt{Nodes}[l]\texttt{.get.sec} = q_a - B_l$ for remaining nodes $l \in F_l(n)$.

Now we have set $R_n^{\text{get}} = \texttt{Nodes}[n]\texttt{.get.prime}$ to be the real number of primary particles to be gotten by (5) if it is lightly loaded, that to be put by (2) if overloaded, or 0 otherwise by (3). We also have set $Q_m^{\text{get}} = \texttt{Nodes}[m]\texttt{.get.sec}$ for all helper nodes $m \in H(n)$ to be the real number of particles to be gotten by (5) if it is lightly loaded, that to be put by the first bottom-up tree traveral if overloaded, or initial 0 unchanged otherwise.

Since each leaf node l has the exact number of primary particles to be gotten P_l^{get} in $\texttt{Nodes}[l]\texttt{.get.prime}$, and the root node r has of course no secondary particles to get and thus $Q_r^{\text{get}} = \texttt{Nodes}[r]\texttt{.get.sec} = 0$ unchanged from its initial value, we have exact number of primary/secondary particles to get/put to/from any node n in $\texttt{Nodes}[n]\texttt{.get.prime}$ and $\texttt{Nodes}[n]\texttt{.get.sec}$ at the end of the traversal loop.

```
for (i=nn-1; i>=0; i--) {             /* top down traversal of node tree */
  int nid, k, npfrac, incgp;
  dint nproot, floating, nptotal, npave;
  node = NodeQueue[i];
  if (!(ch=node->child))  continue;   /* a leaf may reside below some non-
                                         leaves in NodeQueue when its number
                                         of primaries is equal to the
                                         average */
  nid = node->id;
  floating = TotalPGlobal[nid] - NOfPToStay[nid];
                                      /* # of transboundaries + overflows */
  nproot = node->stay.prime + node->stay.sec + node->get.sec;
  if (nproot>nlpmax) {                /* secondary assignment made primary
                                         overflow */
    dint getprime = nlpmax - nproot;  /* getprime<0 to mean to put */
    node->get.prime = getprime;
    floating -= getprime;
    nproot = nlpmax;
  } else {
    node->get.prime = 0;
  }
  if (floating==0)  continue;
  incgp = 0;
  if ((nptotal=assign_particles(nproot, floating, ch, 0, &k))<0) {
                                      /* try to avoid moving primaries of
                                         children to their children */
    incgp = 1;
    if ((nptotal=assign_particles(nproot, floating, ch, 1, &k))<0)
                                      /* allow moving primaries of
                                         children to their children */
      errstop("SECONDARY PARTICLE ASSIGNMENT STABILITY CHECK ERROR");
  }
  npave = nptotal / k;
  npfrac = nptotal - npave*k;         /* should be faster than nptotal%k */
  if (npfrac==0) {
    npave--;  npfrac = k;
  }                                   /* npave = ceil(average)-1 */
  if (nproot<=npave) {
```

```
      if (npfrac-- > 0)  nproot--;      /* get to have ceil(average) */
      node->get.prime = npave - nproot;
    }
    for (ch=node->child; ch; ch=ch->sibling) {
      int npch = ch->stay.prime + ch->stay.sec + ch->get.sec;
      int gp = ch->get.prime;
      if (gp>0 || incgp)  npch += gp;
      if (npch<=npave) {
        if (npfrac-- > 0)  npch--;
        ch->get.sec = npave - npch;
      }
    }
  }
```

Finally, if `Special_Pexc_Sched()` is true for the argument `level` meaning the caller of `try_stable1()` has its own particle exchange scheduling mechanism, we simply return to the caller with `TRUE` to indicate rebalancing is not necessary. Otherwise, we make particle transfer schedule by calling `schedule_particle_exchange()` with an argument `reb` $= 0$ (or -1) if `currmode` $= 1$ (or 3) to mean floating partciles for a subdomain must be (are not necessary to be) found only in the node responsible for the subdomain or its neighbors, that is, we have normal (anywhere) accommodation. Then we call `make_comm_count()` to build `NOfRecv[][][]` and `NOfSend[][][]` as the output of `oh1_transbound()` if `level` $= 1$, or for non-neighboring particle transfers if `currmode` $= 3$. Otherwise, `make_comm_count()` works to initialize `TotalPNext[][]` with the number of particles to be received. And then, we return to the caller `transbound1()` or `try_stable2()` with `TRUE` to indicate rebalancing is not necessary.

```
  if (Special_Pexc_Sched(level)) return(TRUE);
  schedule_particle_exchange(currmode==MODE_NORM_SEC ? 0 : -1);
  make_comm_count(currmode, level, 0, Nodes[myRank].parentid, stats);
  return(TRUE);
}
```

4.3.14 `count_stay()`

`count_stay()` The function `count_stay()`, called only from `try_stable1()` without any arguments, calculates the followings for all $n \in [0, N-1]$.

$$Q_n^n = \texttt{Nodes}[n]\texttt{.stay.prime} = \sum_{s=0}^{S-1} q(n)[0][s][n]$$

$$Q_n^{parent(n)} = \texttt{Nodes}[n]\texttt{.stay.sec} = \sum_{s=0}^{S-1} q(n)[1][s][parent(n)]$$

$$\texttt{NOfPToStay}[n] = \sum_{s=0}^{S-1} \Big(q(n)[0][s][n] + \sum_{c \in H(n)} q(c)[1][s][n] \Big)$$

That is, $\texttt{Nodes}[n]\texttt{.stay.}\{\texttt{prime}, \texttt{sec}\}$ is set to the number of primary/secondary particles currently accommodated by the node n, and $\texttt{NOfPToStay}[n]$ is set to the sum of these `stay`ing particles in the family of n, $F(n)$. The function also gives initial value 0 to $\texttt{Nodes}[n]\texttt{.get.}\{\texttt{prime}, \texttt{sec}\}$.

```
static void
count_stay() {
  int nn=nOfNodes, ns=nOfSpecies, me=myRank;
  int *np, *stay=TempArray;
  struct S_node *node;
  int i, s, sec;
```

For the calculation of `Nodes[].stay.{prime,sec}` and `NOfPToStay[]`, first we make `TempArray`$[n] = \sum_{s=0}^{S-1} q(n)[0][s][n]$ for the local node n, and perform `MPI_Allgather()` to have `TempArray`$[m]$ for all m. Then `TempArray`$[m]$ are copied to `NOfPToStay`$[m]$ and `Nodes`$[m]$`.stay.prime`.

```
  stay[me] = 0;
  for (s=0,np=NOfPLocal; s<ns; s++,np+=nn)  stay[me] += np[me];
                                            /* NOfPLocal[0][s][me] */
  MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, stay, 1, MPI_INT, MCW);
  for (i=0,node=Nodes; i<nn; i++,node++) {
      node->stay.prime = NOfPToStay[i] = stay[i];
      node->get.prime = 0;
  }
```

Next, we make `TempArray`$[n] = \sum_{s=0}^{S-1} q'(n)[1][s][parent(n)]$ for the local node n if it is not the root of family tree, or `TempArray`$[n] = 0$ otherwise. Then and finally, we perform `MPI_Allgather()` for `TempArray[]` again but this time gathered `TempArray`$[m]$ is copied to `Nodes`$[m]$`.stay.sec` and added to `NOfPToStay`$[parent(m)]$ if m is non-root.

```
  sec = RegionId[1];
  stay[me] = 0;
  if (sec>=0)                               /* np=&NOfPLocal[1][0][0] */
    for (s=0; s<ns; s++,np+=nn)  stay[me] += np[sec];
                                            /* NOfPLocal[1][s][me] */
  MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, stay, 1, MPI_INT, MCW);
  for (i=0,node=Nodes; i<nn; i++,node++) {
    sec = node->parentid;
    if (sec>=0)  NOfPToStay[sec] += (node->stay.sec = stay[i]);
    else         node->stay.sec = 0;
    node->get.sec = 0;
  }
}
```

4.3.15 `assign_particles()`

`assign_particles()` This function, called only from `try_stable1()`, examines the load of the nodes in a helpand-helper family, whose helpand n will have a number of particles specified by the argument `npr` $= \rho_n$ at least, and helpers are listed from the argument pointer `ch` to form $H(n)$. Then it finds the set of nodes, whose size is k_n to be *returned* through the argument `nget`, and to which we move a number of particles specified by the argument `npt` $= R_n^{\mathrm{flt}}$ in the subdomain of the family from its member and other nodes to achieve a good tradeoff between the load balancing and the communication cost reduction. More specifically, we perform the followings.

First, we build a histogram of particle populations, or the *base-load* B_m for each member $m \in F(n)$ as follows.

$$\begin{aligned}
B_n &= \rho_n = \min(P_{\max},\ Q_n^n + R_n) \\
B_m &= Q_m^m + Q_m^n + \min(0, Q_m^{\text{get}}) + \begin{cases} P_m^{\text{get}} & \texttt{incgp} = 1 \ \vee \ P_m^{\text{get}} > 0 \\ 0 & \text{otherwise} \end{cases} \\
&= \begin{cases} \min(P_{\max},\ Q_m^n + P_m^{\min}) & \texttt{incgp} = 1 \ \vee \ P_m^{\text{get}} > 0 \\ Q_m^m + \min(Q_m^n,\ P_{\max} - P_m^{\min}) & \text{othewise} \end{cases} \qquad (m \in H(n))
\end{aligned}$$

where;

$$\begin{aligned}
\{Q_m^m, Q_m^n\} &= \texttt{Nodes}[m]\texttt{.stay.}\{\texttt{prime}, \texttt{sec}\} \\
P_m^{\text{get}} &= \texttt{Nodes}[m]\texttt{.get.prime} \qquad \min(0, Q_m^{\text{get}}) = \texttt{Nodes}[m]\texttt{.get.sec}
\end{aligned}$$

and $\texttt{incgp} \in \{0, 1\}$ is the argument given to the function.

With the definitions above, we prove if $\texttt{incgp} = 1$ and $R_n \le P_{\max} - P_n^{\min}$ the following is satisfied to assure the family has room to have R_n^{flt} particles in total.

$$R_n^{\text{flt}} + \sum_{m \in F(n)} B_m \le P_{\max} |F(n)|$$

First we calulate the lefthand side of the inequality above.

$$\begin{aligned}
&\max(0,\ Q_m^n + P_m^{\min} - P_{\max}) + \min(P_{\max},\ Q_m^n + P_m^{\min}) \\
&\qquad = (\max(P_{\max},\ Q_m^n + P_m^{\min}) - P_{\max}) + \min(P_{\max},\ Q_m^n + P_m^{\min}) \\
&\qquad = (P_{\max} + (Q_m^n + P_m^{\min})) - P_{\max} = Q_m^n + P_m^{\min} \\
&\max(0,\ Q_n^n + R_n - P_{\max}) + \min(P_{\max},\ Q_n^n + R_n) \\
&\qquad = Q_n^n + R_n \\
&R_n^{\text{flt}} + \sum_{m \in F(n)} B_m = \sum_{k \notin F(n)} Q_k^n + \sum_{m \in H(n)} \max(0,\ Q_m^n + P_m^{\min} - P_{\max}) + \max(0,\ Q_n^n + R_n - P_{\max}) + \\
&\qquad \sum_{m \in H(n)} \min(P_{\max},\ Q_m^n + P_m^{\min}) + \min(P_{\max},\ Q_n^n + R_n) \\
&\qquad = \sum_{k \notin F(n)} Q_k^n + \sum_{m \in H(n)} (Q_m^n + P_m^{\min}) + Q_n^n + R_n \\
&\qquad = P_n + \sum_{m \in H(n)} P_m^{\min} + R_n
\end{aligned}$$

Then the satisfaction of the inequality is proven as follows.

$$\begin{aligned}
(\sum_{m \in F(n)} B_m + R_n^{\text{flt}}) - P_{\max} |F(n)| &= \Big(P_n + \sum_{m \in H(n)} P_m^{\min} + R_n) - P_{\max} |F(n)| \\
&= ((P_n + R_n) - P_{\max}) - \sum_{m \in H(n)} (P_{\max} - P_m^{\min}) \\
&= \Big(P_n - \sum_{m \in H(n)} (P_{\max} - P_m^{\min})\Big) + (R_n - P_{\max}) \\
&\le \max(0, P_n - \sum (P_{\max} - P_m^{\min})) + (R_n - P_{\max})
\end{aligned}$$

$$
\begin{aligned}
&= P_n^{\min} + (R_n - P_{\max}) \\
&= R_n - (P_{\max} - P_n^{\min}) \le 0 \\
\Leftrightarrow\ & R_n \le P_{\max} - P_n^{\min}
\end{aligned}
$$

Since $P_{\max} \ge P_n^{\min}$ is satsfied for all n and $R_r = 0$ for the root node r to assure $P_{\max} - P_r^{\min} \ge R_r = 0$, if we can distribute R_n^{flt} particles in the members of $F(n)$ keeping $R_m \le P_{\max} - P_m^{\min}$ for all $m \in H(n)$ and keep resulting Q_n not greater then $P_{\max}$, good load balancing should be kept. In fact, the proven inequality assures that we have the following subset $F'(n)$ of $F(n)$, whose member m may receive at most r'_m particles as a part of distribution of R_n^{flt} particles.

$$
F'(n) = \{m \in F(n) \,|\, B_m < P_{\max}\} \qquad r'_m = P_{\max} - B_m
$$

Note that r'_m is sufficient to cover R_n^{flt} by its sum over the members in $F'(n)$, and $R_m \le P_{\max} - P_m^{\min}$ for $m \in F'(n) - \{n\}$ and $Q_n \le P_{\max}$ for $n \in F'(n)$ are satisfied as follows.

$$
\begin{aligned}
&\sum_{m \in F'(n)} r'_m = \sum_{m \in F'(n)} (P_{\max} - B_m) = \sum_{m \in F(n)} (P_{\max} - B_m) = P_{\max}|F(n)| - \sum_{m \in F'(n)} B_m \ge R_n^{\text{flt}} \\
&R_m \le r'_m + Q_n^m = (P_{\max} - B_m) + Q_n^m = P_{\max} - (Q_n^m + P_m^{\min}) + Q_n^m = P_{\max} - P_m^{\min} \\
&Q_n \le r'_n + B_n = (P_{\max} - B_n) + B_n = P_{\max}
\end{aligned}
$$

On the other hand, the required conditions of R_m and Q_n are also satisfied for the members in $F(n) - F'(n)$ as follows.

$$
\begin{aligned}
&R_m = Q_n^m + \min(0, Q_n^{\text{get}}) = Q_n^m + Q_n^{\text{get}} = Q_n^m + (P_{\max} - (P_m^{\min} + Q_m^n)) = P_{\max} - P_m^{\min} \\
&Q_n = Q_n^n + R_n + P_n^{\text{get}} = Q_n^n + R_n + (P_{\max} - (Q_n^n + R_n)) = P_{\max}
\end{aligned}
$$

Now we have assured existence of $F'(n)$ and;

$$
\sum_{m \in F'(n)} (P_{\max} - B_m) \ge R_n^{\text{flt}} \quad \text{or equivalently} \quad R_n^{\text{flt}} + \sum_{m \in F'(n)} B_m \le P_{\max}|F'(n)|
$$

if `incgp` $= 1$. Therefore, we can find a subset $F_l(n)$ of $F'(n)$ which satisfies;

$$
R_n^{\text{flt}} + \sum_{m \in F_l(n)} B_m \le P_{\max}|F_l(n)|
$$

as follows. First we sort B_m in ascending order using `TempArray[]` as a temporary sorting buffer and calling `qsort()` giving it the comparation funciont `compare_int()`, to have an ascending sequence B'_0, B'_1, ..., $B'_{|F(n)|-1}$. Next we find minimum k_n such that

$$
R_n^{\text{flt}} + \sum_{i=0}^{k_n - 1} B'_i \le k_n B'_{k_n}
$$

or let $k_n = |F(n)|$ if such k_n does not exist, and let $F_l(n) = \{m \,|\, B_m < B'_{k_n}\}$ where $B'_{|F(n)|} = P_{\max}$. Since $B_m = P_{\max}$ for $B_m \in F(n) - F'(n)$, it is assured that we can have $F_l(n) \subset F(n)$ if $F'(n) \subset F(n)$. Otherwise, i.e., if $F'(n) = F(n)$, it can be $F_l(n) = F(n)$ but is all right because it means $B_m < P_{\max}$ for all $m \in F(n)$. This process is to find k_n nodes having small number of particles and to make them get some particles for resulting loads balanced among them, while other heavily loaded nodes will not get any particles

(but may put some of them out if it is necessary not for load balancing but for keeping the helpand-helper configuration).

On the other hand, If `incgp` $= 0$ this particle assignnment could make load overflow resulting;

$$R_n^{\mathrm{flt}} + \sum_{i=0}^{k_n-1} B'_i > k_n P_{\max}$$

(k_n may be less than $|F(n)|$) because we make overesitimation for B_m neglecting negative P_m^{get}, and thus find we don't have room enough to accommodate R_n^{flt} particles without pushing helper's primary particles down to their grand-helpers. In this case we return -1 to report the failure so that the caller `try_stable1()` call this function again with `incgp` $= 1$. Otherwise we return

$$T_n = R_n + \sum_{i=0}^{k-1} B'_i = R_n + \sum_{m \in F_l(n)} B_m$$

to report that its average for k_n nodes is the target of the number of particles which lightly loaded nodes will have in total. We also *return* the value of k_n through the pointer argument `nget` to the caller.

```
static dint
assign_particles(dint npr, dint npt, struct S_node *ch, int incgp, int *nget) {
  int *np=TempArray;            /* used just for temporary sorting buffer */
  int n, i;
  dint nlpmax = nOfLocalPMax;

  np[0] = npr;
  for (n=1; ch; ch=ch->sibling, n++) {
    int gp=ch->get.prime;
    np[n] = ch->stay.prime + ch->stay.sec + ch->get.sec;
    if (gp>0 || incgp)  np[n] += gp;
  }
  qsort(np, n, sizeof(int), compare_int);       /* sort ascendingly */
  for (i=0; i<n; i++) {
    dint npc=np[i];
    if (npt<=npc*i)  break;
    npt += npc;
  }
  *nget = i;
  return(npt>nlpmax*i ? -1 : npt);
}
```

4.3.16 `compare_int()`

`compare_int()` The function `compare_int()`, called from `qsort()` in `assign_particles()` with two `void` poiter arguments `x` and `y`, returns the following r for the comparison of an integer pair X and Y pointed by the arguments in the array to be sorted.

$$r = \begin{cases} -1 & X < Y \\ 0 & X = Y \\ 1 & X > Y \end{cases}$$

```
static int
compare_int(const void* x, const void* y) {
  int xx=*((int*)x), yy=*((int*)y);

  if (xx<yy)  return(-1);
  if (xx>yy)  return(1);
  return(0);
}
```

4.3.17 schedule_particle_exchange()

schedule_particle_exchange()

The function schedule_particle_exchange(), called from try_stable1() or rebalance1 () with one argument reb, makes the schedule of the inter-node transfer of particles which reside in the primary subdomain of the local node in the next step. The argument reb has one of the followings.

- -1 means we were in secondary mode with anywhere accommodation in the last step and try_stable1() found the helpand-helper configuration is sustainable. Floating particles for a subdomain may be found in any nodes due to, for example, particle injections.
- 0 means we were in secondary mode with normal accommodation in the last step and try_stable1() found the helpand-helper configuration is sustainable. Floating particles for a subdomain must be found in nodes responsible for the subdomain or its neighbors.
- 1 means we were in secondary mode with normal accommodataion in the last step but try_stable1() found the helpand-helper configuration must be reformed by rebalance1(). Floating particles for a subdomain must be found in nodes which were responsible for the subdomain or its neighbors.
- 2 means we were in primary mode with normal accommodataion in the last step but try_primary1() found we cannot continue primary mode execution and a new helpand-helper configuration must be established by rebalance1(). Floating particles for a subdomain must be found in nodes which were responsible for the subdomain or its neighbors as primary ones.
- 3 means a new helpand-helper configuration was established by rebalance1() and floating particles for a subdomain may be found in any nodes due to, for example, initial particle distribution or particle injections, to mean we have anywhere accommodation.

Before this function is called, the callers have determined the numbers of primary and secondary particles which should be gotten in or put out from/to each node n and have set them in $R_n^{\mathrm{get}} = NN[n]$.get.prime and $Q_{parent(n)}^{\mathrm{get}} = NN[n]$.get.sec, where NN is Nodes if reb ≤ 0 or NodesNext otherwise. That is, if a new family tree is build by rebalance1(), NodesNext[] has the new configuration while Nodes[] keeps the old configuration.

```
static void
schedule_particle_exchange(int reb) {
  int me=myRank, nn=nOfNodes, ns=nOfSpecies, nnns=nn*ns;
```

```
struct S_node *mynode, *ch;
int i, slidx;
struct S_commsched_context context;
```

First we build the sequence of `S_commlist` records in `CommList[]` for sending/receiving particles in the local primary subdomain n. The sequnce is built by a loop to scan the the family members rooted by the local node n in the next step, that is $NN[n]$, calling `sched_comm()` for each member with the following arguments.

- `toget` is $R_n^{\rm get}$ = $NN[n]$`.get.prime` for the local node n or $Q_n^{\rm get}$ = $NN[m]$`.get.sec` for its helper m. If it is positive, `sched_comm()` adds records into `CommList[]`. Otherwise, i.e., the node receives no particles in the subdomain n but put some of them out, `sched_comm()` will do nothing.
- `rid` is the receiving node ID n or m.
- `tag` is 0 for `rid` $= n$ meaning primary particles, and S for others meaning secondary particles. An MPI communication tag will be given by the sum of this argument and the species of the particles transferred, so that the receiver recognizes whether the particles are primary or secondary as well as their species. Moreover, the tag value of $pS+s$ where $p \in \{0, 1\}$ can be used for the one-dimensional index of a two-dimensional array of $[2][S]$ to access its element $[p][s]$.
- `reb` is simply equal to the argument `reb` of this function. It notifies `sched_comm()` where it can find particle senders, only in current neighbor families (0), both in old and new neighbor families (1), only in current neighbors (2), or in any nodes (-1 or 3). It also shows that `sched_comm()` should refer to `Nodes[]` if `reb` ≤ 0 or to `NodesNext[]` if `reb` > 0 for the helpand-helper configuration. Note that if `reb` $= -1$ it is set to 3 after the call of `sched_comm()` to mean anywhere accommodation.
- `context` is a `S_commsched_context` structure having the following elements to hold the execution context of `sched_comm()`, whose initial value is shown in parens.
  - `neighbor` is the index of `DstNeighbors[]` currently processed (0).
  - `sender` is the sender node ID currently processed (0 if `reb` $= -1$ or `reb` $= 3$, or `DstNeighbors`$[0]$ otherwise).
  - `comidx` is the index of `CommList[]` at which the next `S_commlist` record will be stored (0).
  - `spec` is the species of the sender node currently processed (0).
  - `dones` is the number of already processed particles in `spec` (0).
  - `donen` is the number of already processed primary/secondary particles accommodated by the sender which is a family member (0).

Note that `sched_comm()` also consults `TempArray`$[m]$ to check double visiting of the sender node m, and thus we clear `TempArray[]` and then turns on the entry of the first sender. Also note that `sched_comm()` let `RLIndex`$[k]$ be the starting index of `CommList[]` for senders in the family rooted by `DstNeighbors`$[k]$ for all $k > 0$, and thus we let `RLIndex`$[0] = 0$.

Another remark is that we have to adjust $Q_m^{\rm get}$ = $NN[m]$`.get.sec` for all $m \in F(n)$, if `reb` > 0 meaning the function is called from `rebalance1()` which set $Q_m^{\rm get}$ assuming m does not have any secondary particles. Therefore, we have

to calculate $\sum_{s=0}^{S} q(m)[1][s][n] = \sum_{s=0}^{S}$ `NOfPrimaries`$[1][s][m]$ for $m \in H(n)$ and $\sum_{s=0}^{S} q(n)[1][s][parent(n)] = \sum_{s=0}^{S}$ `NOfPLocal`$[1][s][parent(n)]$ for n by `count_real_stay()`, to set it in $NN[\{m,n\}]$`.stay.sec` for further references[33], and to subtract it from $Q^{\mathrm{get}}_{\{m,n\}}$ so that they reflects the number of secondary particles accommodated by those nodes because their secondary subdomain is unchanged or they have secondary particles in new secondary subdomain accidentally.

After we finish the loop to call `sched_comm()` and let `SLHeadTail`$[0]$ = `context.comidx` to record the end of the primary receiving block, we return to the caller `try_stable1()` or `rebalance1()` if `reb` $= 3$ (including the case updated from -1) because we cannot notify senders of the schedule by neighboring communication and broadcast in neighboring families.

```
RLIndex[0] = 0;
context.neighbor = 0;
context.sender = (reb<0 ||reb==3) ? 0 : DstNeighbors[0];
context.comidx = 0;
context.spec = 0;  context.dones = 0;  context.donen = 0;
for (i=0; i<nn; i++) TempArray[i] = 0;
TempArray[context.sender] = 1;
if (reb>0) {
  mynode = NodesNext + me;
  for (ch=mynode->child; ch; ch=ch->sibling)
    ch->get.sec -= (ch->stay.sec=count_real_stay(NOfPrimaries+nnns+ch->id));
                                              /* NOfPrimaries[1][0][cid] */
  sched_comm(mynode->get.prime, me, 0, reb, &context);
  for (ch=mynode->child; ch; ch=ch->sibling)
    sched_comm(ch->get.sec, ch->id, ns, reb, &context);
  if (mynode->parent)
    mynode->get.sec -=
      (mynode->stay.sec=count_real_stay(NOfPLocal+nnns+mynode->parentid));
                                              /* NOfPLocal[1][0][pid] */
} else {
  mynode = Nodes + me;
  sched_comm(mynode->get.prime, me, 0, reb, &context);
  for (ch=mynode->child; ch; ch=ch->sibling)
    sched_comm(ch->get.sec, ch->id, ns, reb, &context);
  if (reb<0)  reb = 3;
}
SLHeadTail[0] = slidx = context.comidx;
if (reb==3) return;
```

Now we have the transfer schedule for the local primary subdomain in `CommList`$[i]$ where $i \in [0, \sigma)$ and $\sigma =$ `context.comidx` $=$ `slidx` which has sub-blocks starting from `RLIndex`$[k]$ for senders in each neighboring families rooted by `DstNeighbors`$[k]$. Since the largest k namely $k_{\max} =$ `context.neighbor` could be less than `OH_NEIGHBORS` $= 3^D$ and we let `RLIndex`$[k]$ be σ for all $k \in [k_{\max}, 3^D]$, which could be left unassigned (very unlikely, but ...). The value of σ was also stored into `SLHeadTail`$[0]$ from which the local node receives sending schedules from its neighbors to form primary sending block.

Then we perform `MPI_Sendrecv()`, `MPI_Send()` or `MPI_Recv()` to send `T_Commlist` type data in `CommList`$[i]$ for $i \in$ [`RLIndex`$[k]$, `RLIndex`$[k+1]-1$] to the k-th neighbor

[33] $NN[m]$`.stay.sec` is referred to by level-4p/4s function `make_recv_list()`, while $NN[n]$'s is not referred to so far but we set the value to it for consistency.

DstNeighbors[k] from the local node, and/or to receive the schedule of (3^D-1-k)-th neighbor subdomain SrcNeighbors[k] which is stored in the block starting from CommList[SLHeadTail[0]]. The size of each received schedule for sending, which must be less than $N + NS$, is obtained by MPI_Get_count() and its total is added to SLHeadTail[0] and is stored in SLHeadTail[1]. Note that since we omit sending/receiving for negative DstNeighbors[k] and SrcNeighbors[k] respectively, the schedule for a process is transferred only once.

Now the local node has the particle sending schedule, i.e., primary sending block, for particles accommodated by it and its helpers. Then if reb = 2, since no nodes have helpers because we are in primary mode, we finish this function and return to its caller rebalance1().

```
  for (i=context.neighbor+1; i<=OH_NEIGHBORS; i++)  RLIndex[i] = slidx;
  for (i=0; i<OH_NEIGHBORS; i++) {
    int dst=DstNeighbors[i];
    int src=SrcNeighbors[i];
    int rc;
    MPI_Status st;
    if (dst==me) continue;
    if (src>=0) {
      if (dst>=0)
        MPI_Sendrecv(CommList+RLIndex[i], RLIndex[i+1]-RLIndex[i], T_Commlist,
                     dst, 0,
                     CommList+slidx, nn+nnns, T_Commlist, src, 0, MCW, &st);
      else
        MPI_Recv(CommList+slidx, nn+nnns, T_Commlist, src, 0, MCW, &st);
      MPI_Get_count(&st, T_Commlist, &rc);
      slidx += rc;
    } else if (dst>=0)
      MPI_Send(CommList+RLIndex[i], RLIndex[i+1]-RLIndex[i], T_Commlist,
               dst, 0, MCW);
  }
  SLHeadTail[1] = slidx;
  if (reb==2) return;
```

Now the local node broadcasts the transfer schedules, that created by itself and those received from neighbors, to its helpers by oh1_broadcast(). First the local node broadcasts its SLHeadTail[0, 1] to show its helpers the size of the primary receiving and primary sending blocks, and stores that received from its helpand into SecSLHeadTail[0, 1], which are initialized to be 0 for the family tree root because it does not receive anything. Then both blocks of T_Commlist type are broadcasted but, if rebalanced in secondary mode, the primary receiving block will be ignored by the helpers whose helpand remains unchanged by the rebalance (i.e., the local node is their old and new helpand) because they only refer to the duplicated block broadcasted in the newly established family afterward.

```
  SecSLHeadTail[0] = SecSLHeadTail[1] = 0;
  oh1_broadcast(SLHeadTail, SecSLHeadTail, 2, 2, MPI_INT, MPI_INT);
  oh1_broadcast(CommList, CommList+slidx, slidx, SecSLHeadTail[1],
                T_Commlist, T_Commlist);
}
```

4.3.18 count_real_stay()

count_real_stay() The function count_real_stay(), called from schedule_particle_exchange() and rebalance1(), calculatates $\sum_{s=0}^{S} q(k)[p][s][l]$ for (k,l,p) being $(n,n,0)$, $(n,parent(n),1)$, or $(m,n,1)$ where $m \in H(n)$, and return the sum, being the number of primary/ secondary particles really accommodated by the local node n or its helper m, to the caller. Since the targets of the summation are NOfPLocal$[0][s][n]$ or NOfPrimaries$[0][s][n]$, NOfPLocal$[1][s][parent(n)]$ and NOfPrimaries$[1][s][m]$ respectively, the callers specify the pointer to the element of $s = 0$ through the sole argument np of this function.

```
static int
count_real_stay(int *np) {
  const int ns=nOfSpecies, nn=nOfNodes;
  int stay, s;

  for (s=0,stay=0; s<ns; s++,np+=nn)  stay += *np;
  return(stay);
}
```

4.3.19 sched_comm()

sched_comm() The function sched_comm(), called only from schedule_particle_exchange() with the arguments discussed in §4.3.17, adds S_commlist records to CommList[] from its index context->comidx, for the transfer of $p_{\rm get}$ = toget (possibly non-positive) particles in the primary subdomain n of the local node to the node n_r = rid, which is the local node itself or one of its helpers, from the node n_s = context->sender and its successors which are explained later. Each element of S_commlist record is set to the following.

- rid is the argument rid $= n_r$ always.
- sid is context->sender $= n_s$ or its successor.
- region is myRank $= n$ always.
- tag is $t + s$ for sepcies s where t is the argument tag which is 0 for if $n_r = n$, i.e., n_r is the helpand of the subdomain n, or S otherwise, i.e., n_r is a helper.
- count is the number of particles of the species s in the subdomain n transferred from the node n_s to node n_r.

Note that it is possible that $n_r = n_s$ to make a on-node communication for the particle transfer from the primary to the secondary subdomain of the node $n_r = n_s$ and vice versa. The transfer starts from the particles of species s set to the argument context->spec and accommodated by the node n_s, but some of them whose amount q_s is set in the argument context->dones have already been processed.

```
static void
sched_comm(int toget, int rid, int tag, int reb,
           struct S_commsched_context *context) {
  int neighbor = context->neighbor, sid = context->sender;
  int comidx = context->comidx;
  int s=context->spec, havedones=context->dones, havedonen=context->donen;
```

```
int me=myRank;
int nn=nOfNodes, ns=nOfSpecies, nnns=nn*ns;
int i=nn*s+sid;                          /* [0][s][sid] */
struct S_node *nodesnext = reb>0 ? NodesNext : Nodes;
```

The heart of this funciton is to determine the `count` element of the `S_commlist` records. To do that, we scan `NOfPrimaries`$[0][s][n_s]$ and `NOfPrimaries`$[1][s][n_s]$ from initial setting of n_s being `context->sender` and s being `context->spec`, incrementing s and, when s goes back to 0 cyclicly, advancing n_s to successors, while $p_{\rm get}$ is positive. The `count` for the species s and the sender node n_s, namely $p_{\rm put}$ = `toput`, is basically

$$p_{\rm put} = \texttt{NOfPrimaries}[0][s][n_s] + \texttt{NOfPrimaries}[1][s][n_s] - q_s$$

representing the number of unprocessed particles which was accommodated by the node n_s but moved into the subdomain n. However, we have the following two exceptions where NN is `NodesNext` if the argument `reb` > 0 indicating the family is newly established by rebalancing, or `Nodes` otherwise.

- If $n = n_s$, i.e., the node n_s is local node and thus the helpand of the family for n, we have to replace `NOfPrimaries`$[0][s][n_s]$ with 0 if $NN[n_s]$`.get.prime` $= R^{\rm get}_{n_s} > 0$ meaning no primary particles are put out from n_s, or with $-R^{\rm get}_{n_s}$ otherwise indicating pushing-down some particles to its helpers. Let $g = \max(0, -R^{\rm get}_{n_s})$.
- If $n = \mathit{parent}(n_s)$, i.e., the node n_s is a helper of the family for n, we have to replace `NOfPrimaries`$[1][s][n_s]$ with 0 if $NN[n]$`.get.sec` $= Q^{\rm get}_{n_s} > 0$ meaning no secondary particles are put out from n_s, or with $-Q^{\rm get}_{n_s}$ otherwise indicating secondary overflow. Let $g = \max(0, -Q^{\rm get}_{n_s})$.

In both cases above, g has the number of total particles, if any, to be put out regardless of species. Thus we maintain the already-processed number of particles q_n whose initial value is given by the argument `context->donen`. Thus if $q_n < g$ we still have $g - q_n$ particles to be processed. Therefore, number of particles of s to be put is $\min($`NOfPrimaries`$[p][s][n_s], g - q_n)$ where $p = 0$ if $n_s = n$ or $p = 1$ otherwise, and q_n should be incremented by this amount after all the particles of s are processed.

Then if $p_{\rm put} > p_{\rm get}$, i.e., the node n_s still has particles in question more than those to be sent to the node n_r, $p_{\rm put}$ is set to $p_{\rm get}$, q_s is incremented by $p_{\rm get}$ to indicate this amount have been processed, and $p_{\rm get}$ is cleared to be 0 (by subtracting $p_{\rm put} = p_{\rm get}$ from it) to finish the scanning loop. Otherwise, $p_{\rm get}$ is decremented by $p_{\rm put}$ (and possibly becomes 0) and we completed the process for the species s. That is, q_s is cleared to be 0, q_n is incremented as discussed above, and, after adding the records to `CommList`[], s is incremented to process new species. Then, if s becomes S and goes back to 0, n_s is advanced to process new sender node.

```
while (toget>0) {
  struct S_node *snoden=nodesnext+sid;
  int npp = NOfPrimaries[i], nps = NOfPrimaries[i+nnns];
                                                  /* [0/1][s][sid] */
  int toput, hdninc = 0;
  int next=1;
  if (sid<0)  local_errstop("PARTICLE TRANSFER SCHEDULING ERROR");
  if (sid==me) {
    int nput = snoden->get.prime + havedonen;
```

```
  nput = nput<0 ? -nput : 0;
  if (nput<npp) npp = nput;
  hdninc = npp;
}
else if (snoden->parentid==me) {
  int nput = snoden->get.sec + havedonen;
  nput = nput<0 ? -nput : 0;
  if (nput<nps) nps = nput;
  hdninc = nps;
}
toput = npp + nps - havedones;
if (toput>0) {
  struct S_commlist *cptr = CommList+(comidx++);
  if (toput>toget) {
    havedones += toget;
    toput = toget;
    next = 0;
  }
  cptr->rid = rid;  cptr->sid = sid;  cptr->region = me;
  cptr->count = toput;  cptr->tag = tag + s;
  toget -= toput;
}
if (next) {
  havedones = 0;  havedonen += hdninc;
  s++;  i += nn;
```

The advancement of n_s is performed in the families rooted by neighboring subdomains of n and the family of n itself. That is, if n_s is the root of a family of currently processed k-th neighboring subdomain whose initial value is given by `context->neighbor`, i.e., n_s = `DstNeighbors`$[k]$, the argument `reb` ≤ 1 to indicate that we are in secondary mode currently, and n_s has some helpers, we advance to n_s's first helper. On the other hand, if n_s is a helper of a family of the k-th neighboring subdomain, `reb` ≤ 1, and n_s has a sibling, we advance to n_s's sibling. Otherwise, i.e., n_s is the last family member of the k-th neighboring subdomain (helpand without helper, possiblily due to that we are in primary mode, or the last helper), we advance k to its first succeeding neighboring subdomain which has not been visited (i.e., `DstNeighbors`$[k] \geq 0$) or k becomes 3^D, each time setting `RLIndex`$[k]$ to the next index of `CommList`[]. In the former case, we simply set k to the neighbor which has not been visited yet (as the root). In the latter case, we have to visit the newly established family for n if the argument `reb` is 1 (in secondary mode) or 2 (in primary mode), since the neighboring subdomain families are those before rebalancing. (The case of 3 will be explained later.) In this case we start the scan from its first helper because there should be `DstNeighbors`$[k] = n$ and thus we must have already visited n itself. This extra family scan with $k = 3^D$ will stop after the last helper is processed, or does not eventually start if n's family does not have helpers or `reb` $= 0$, resulting in $n_s = -1$.

The advancement of n_s should take care of the possibility that a sender may be scanned twice, as the root of a neighboring family and as a helper of another neighboring family. To detect the second visit, `TempArray`$[n_s]$ is cleared to be zero by the caller `schedule_particle_exchange()` and then turned to 1 when n_s is visited. Thus if we encounter a node n_s with `TempArray`$[n_s] = 1$, we skip the node and advance n_s further.

The family scan above is not performed when `reb` $\in \{-1, 3\}$ to indicate that we have anywhere accommodation. In this case, we simply scans all nodes from 0 to $N - 1$.

```
      if (s==ns) {
        havedonen = 0;
        s = 0;
        if (reb>=0 && reb!=3) {
          struct S_node *nodes = (neighbor<OH_NEIGHBORS ? Nodes : NodesNext);
          struct S_node *snode = nodes + sid;
          while (sid>=0) {
            if (neighbor==OH_NEIGHBORS) {
              snode = snode->sibling;  sid = snode ? snode - nodes : -1;
            }
            else if (sid==DstNeighbors[neighbor] && reb<2 && snode->child) {
              snode = snode->child;  sid = snode - Nodes;
            }
            else if (sid!=DstNeighbors[neighbor] && reb<2 && snode->sibling) {
              snode = snode->sibling;  sid = snode - Nodes;
            }
            else {
              RLIndex[++neighbor] = comidx;
              while(neighbor<OH_NEIGHBORS && (sid=DstNeighbors[neighbor])<0)
                RLIndex[++neighbor] = comidx;
              if (neighbor==OH_NEIGHBORS) {
                nodes = NodesNext;
                snode = nodes[me].child;
                sid = (snode && reb) ? snode - nodes : -1;
              } else {
                snode = Nodes + sid;
              }
            }
            if (sid>=0 && TempArray[sid]==0) {
              TempArray[sid] = 1;  break;
            }
          }
        } else {
          sid++;
        }
        i = sid;
      }
    }
  }
```

Finally, after we complete the process for all particles the node n_r receives, we store the currently visiting node n_s to `context->sender` toghether with the neighboring subdomain index k of the family which n_s belongs to into `context->neighbor`. We also store s, q_s and q_n into `context->{spec, dones, donen}` respectively, together with the next index of `CommList[]` into `context->comidx`. By *returning* these values, we can continue the scan specified by them in the next call of this function.

```
  context->neighbor = neighbor;  context->sender = sid;
  context->comidx = comidx;
  context->spec = s;  context->dones = havedones;  context->donen = havedonen;
}
```

4.3.20 make_comm_count()

make_comm_count()

The function make_comm_count, called from try_stable1() or rebalance1(), sets (the base of) TotalPNext[][] unconditionally and NOfRecv[][][] and NOfSend[][][] if necessary. Besides the arguments of callers themselves, i.e., currmode, level and stats, it has two additional arguments as follows.

- reb = 0 if called from try_stable1(), or reb = 1 otherwise, i.e., from rebalance1().
- oldparent has the node ID of the local node's helpand, in the configuration before rebalancing if done. More specifically, this aregument for the local node n has the following value where $parent(n)$ and $parent_{old}(n)$ are n's *new* (or current) and *old* helpand before rebalancing.

$$\texttt{oldparent} = \begin{cases} parent(n) & \texttt{reb} = 0 \\ parent_{old}(n) & \texttt{reb} = 1 \ \wedge \ \texttt{currmode} = 1 \\ -1 & \texttt{reb} = 1 \ \wedge \ \texttt{currmode} \neq 1 \end{cases}$$

```
static void
make_comm_count(int currmode, int level, int reb, int oldparent, int stats) {
  int nn=nOfNodes, ns=nOfSpecies, nnns=nn*ns, nnns2=nnns*2, me=myRank;
  struct S_node *mynode=Nodes+me;
  int newparent=mynode->parentid;
  int ps, s, i;
```

The first job of this function is to broadcast the primary receiving block CommList[SLHeadTail[0]] to the (new) family members to have the block in SecRList[SecRLSize]. This broadcast is necessary for the following cases.

- The case reb = 1 and currmode = MODE_NORM_PRI (primary mode) in which schedule_particle_exchange() did not broadcast the block because we do not have old families and have not yet build new family communicators. SecRList is placed following the primary sending block and thus starts from CommList[SLHeadTail[1]].
- The case reb = 1 and currmode = MODE_NORM_SEC (secondary mode) in which the secondary receiving block was given from old helpand. SecRList is placed following the secondary sending block and thus starts from CommList[SLHeadTail[1] + SecSLHeadTail[1]].
- The case reb = 1 and Mode_Is_Any() for currmode is true, or currmode = MODE_ANY_SEC, in which schedule_particle_exchange() did not broadcast the block because any node can be a sender to any subdomain and thus old family configuration is useless if rebalanced. SecRList is placed following the primary receiving block and thus starts from CommList[SLHeadTail[0]].

As done in schedule_particle_exchange(), the broadcast is done by two successive calls of oh1_broadcast(), the former sending the size SLHeadTail[0] and the latter sending CommList[SLHeadTail[0]] of T_Commlist data type.

On the other hand, if the broadcasting is not necessary (reb = 0 and currmode = 1), SecRList and SecRLSize are set to represent the secondary receiving block which has already obtained.

```
if (reb || currmode==MODE_ANY_SEC) {
  SecRLSize = 0;
  oh1_broadcast(SLHeadTail, &SecRLSize, 1, 1, MPI_INT, MPI_INT);
  if (currmode==MODE_NORM_PRI)
    SecRList = CommList + SLHeadTail[1];
  else if (currmode==MODE_NORM_SEC)
    SecRList = CommList + SLHeadTail[1] + SecSLHeadTail[1];
  else
    SecRList = CommList + SLHeadTail[0];
  oh1_broadcast(CommList, SecRList, SLHeadTail[0], SecRLSize,
                T_Commlist, T_Commlist);
} else {
  SecRList = CommList + SLHeadTail[1];
  SecRLSize = SecSLHeadTail[0];
}
```

Next, if `level` $= 1$, `Mode_Is_Any()` for `currmode` is true, or `stats` $\neq 0$ meaning that we need to return the shadow of `NOfRecv[][][]` and `NOfSend[][][]`, namely `RecvCounts[][][]` and `SendCounts[][][]`, as level-1 API, or to have them for all-to-all particle transfers or statistics, we count receiving and sending particles to set their amounts into the arrays by scanning `CommList[]`. First we scan the primary receiving block by `make_recv_count()` and then the secondary receiving block if the local node is not the root. Note that these scans not only for `NOfRecv[][][]` but also for `TotalPNext[][]` and `NOfSend[][][]`. The former is to give the *base* number of the particles the local node has in the next step by counting the receiving particles. The latter is necessary because receivings block may have paritlce ejections to family members.

Another remark is that we have to scan the secondary receiving block given by the old helpand before rebalancing which is different from the new (current) helpand. That is, in this case the local node ejects all the old secondary particles to the members in the new family rooted by the old helpand, but this ejection may not be recoreded in sending blocks. This operation must be performed only when we were in secondary mode and rebalancing were taken place switching the helpand of the local node. As discussed at the beginning of this section, this condition for the local node n is confirmed by `oldparent` $\neq$ *parent*(n) $\wedge$ `oldparent` ≥ 0.

Then if `Mode_Is_Any()` for `currmode` is true requiring all-to-all particle transfers, we globally exchange `NOfRecv[][][]` by `MPI_Alltoall()` with `T_Histogram` data type to get `NOfSend`$[p][s][m]$ of the local node n from `NOfRecv`$[p][s][n]$ of the node m[34].

Otherwise, we can set elements of `NOfSend[][][]` by scanning sending blocks by calling `make_send_count()` (usually) twice, once giving primary sending block, and then with secondary sending block if the local node is not the root of the *old* family tree. Note that in this scan we may encounter a `S_commlist` record which has already been found in the scan of receiving blocks, but it is not hazardous because it simply overwrites an element of `NOfSend[][][]` with the same value which has been stored.

If the operations above are not performed because `level` > 1, `currmode` < 2 and `stats` $= 0$, the case in which we need to have neither `NOfRecv[][][]|` nor `NOfSend[][][]`, we only count the base value of `TotalPNext[][]` by calling `count_next_particles()` (usually) twice, once giving primary receiving block, and then with secondary receiving block if the local node is not the root of the family tree.

[34]Since Intel MPI has a bug in `MPI_Alltoall()` with $12 \leq N \leq 16$ and the data type `T_Histogram` that `NOfSend`$[p][s][n]$ of the local node n is not updated for $p > 0$ or $s > 0$, we have to copy `NOfRecv`$[p][s][n]$ to it explicitly until the bug is fixed.

```
  for (s=0; s<ns*2; s++) TotalPNext[s] = 0;             /* TotalPNext[p][s] */
  if (level==1 || Mode_Is_Any(currmode) || stats) {
    for (i=0; i<nnns2; i++)  NOfRecv[i] = NOfSend[i] = 0;
    make_recv_count(CommList, SLHeadTail[0]);
    if (newparent>=0)
      make_recv_count(SecRList, SecRLSize);
    if (oldparent!=newparent && oldparent>=0)
      make_recv_count(CommList+SLHeadTail[1], SecSLHeadTail[0]);
    if (Mode_Is_Any(currmode)) {
      MPI_Alltoall(NOfRecv, 1, T_Histogram, NOfSend, 1, T_Histogram, MCW);
#ifndef INTEL_MPI_BUG_FIXED
      for (ps=0,i=me; ps<2; ps++)  for (s=0; s<ns; s++,i+=nn)
        NOfSend[i] = NOfRecv[i];
#endif
    } else {
      make_send_count(CommList+SLHeadTail[0], SLHeadTail[1]-SLHeadTail[0]);
      if (oldparent>=0)
        make_send_count(CommList+SLHeadTail[1]+SecSLHeadTail[0],
                        SecSLHeadTail[1]-SecSLHeadTail[0]);
    }
  } else {
    count_next_particles(CommList, SLHeadTail[0]);
    if (newparent>=0)
      count_next_particles(SecRList, SecRLSize);
  }
  if (stats) stats_secondary_comm(currmode, reb);
```

The last operation performed if `level` $= 1$ is to complete the setting of `TotalPNext[][]` by adding the following $q(p,s)$ to its element $[p][s]$ for the local node n, while this operation is left to level-2 function `move_to_sendbuf_secondary()` if `level` > 1.

$$q_{\rm put}(p) = \begin{cases} \min(0, -R_n^{\rm get}) & p = 0 \\ \min(0, -Q_n^{\rm get}) & p = 1 \end{cases}$$

$$q_{\rm stay}(p,s) = \begin{cases} \texttt{NOfPLocal}[0][s][n] & p = 0 \\ \texttt{NOfPLocal}[1][s][parent(n)] & p = 1 \end{cases}$$

$$q(p,s) = \max\left(0,\ q_{\rm stay}(p,s) - \max\left(0,\ q_{\rm put}(p) - \sum_{t=0}^{s-1} q_{\rm stay}(p,t)\right)\right)$$

That is, if the local node puts some primary/secondary particles out to its family members, this ejection is done in smaller-first manner of species identifiers to reduce their particle amounts or to make them empty. Otherwise, the number of particles currently accommodated by the local node is simply added to the base value.

```
  if (level==1) {
    for (ps=0,i=0; ps<(newparent<0?1:2); ps++) {
      int putme = ps==0 ? -mynode->get.prime : -mynode->get.sec;
      int *mynps = ps==0 ? NOfPLocal+me : NOfPLocal+nnns+newparent;
      if (putme<0) putme = 0;
      for (s=0; s<ns; s++,i++,mynps+=nn) {
        int stay=*mynps;
        int tpni=TotalPNext[i];
        if (putme<stay) {
```

```
            TotalPNext[i] = tpni + stay - putme;  putme = 0;
          }
          else putme -= stay;
        }
      }
    }
  }
```

4.3.21 make_recv_count()

make_recv_count()

The function make_recv_count(), called only from make_comm_count(), scans a part of CommList whose head and size are specified by the arguments rlist and rlsize. The scan is to set the primary ($p = 0$) or secondary ($p = 1$) particle count c of species s which the local node n will receive from the node m into NOfRecv$[p][s][m]$. Therefore, when we find a S_commlist record whose rid is n and which has tag $= t = pS + s$ and sid $= m$, we set its count $= c$ into NOfRecv$[p][s][m]$. Note that since the one-dimensional index of $[p][s][m]$ for an array $[2][S][N]$ is $(pS + s)N + m$, that index can be calculated by $tN + m$.

The receiving count c is also added to the element of the array TotalPNext$[p][s]$ whose one-dimensional index is $pS + s$ or simply t, to have the total number of primary/secondary receiving particles of species s.

In addition, the function also sets the number of particles c of species s which the local node n will send to its family member node m as its primary ($p = 0$) or secondary ($p = 1$) particles into NOfSend$[p][s][m]$. That is, when we find a record whose sid is n and which has tag $= t = pS + s$ and rid $= m$, we set its count $= c$ into NOfSend$[p][s][m]$ whose one-dimensional index is $tN + m = (pS + s)N + m$.

```
static void
make_recv_count(struct S_commlist* rlist, int rlsize) {
  int me=myRank, nn=nOfNodes;
  int i;

  for (i=0; i<rlsize; i++) {
    int rid=rlist[i].rid, sid=rlist[i].sid;
    int tag=rlist[i].tag, count=rlist[i].count;
    if (rid==me) {
      NOfRecv[tag*nn+sid] = count;
      TotalPNext[tag] += count;
    }
    if (sid==me)
      NOfSend[tag*nn+rid] = count;
  }
}
```

4.3.22 make_send_count()

make_send_count()

The function make_send_count(), called only from make_comm_count(), scans a part of CommList whose head and size are specified by the arguments slist and slsize. The scan is to set the primary ($p = 0$) or secondary ($p = 1$) particle count c of species s which the local node n will send to the node m into NOfSend$[p][s][m]$. Therefore, when we find a S_commlist record whose sid is n and which has tag $= t = pS + s$ and rid $= m$, we set

its count $= c$ into NOfSend$[p][s][m]$. Note that since the one-dimensional index of $[p][s][m]$ for an array $[2][S][N]$ is $(pS + s)N + m$, that index can be calculated by $tN + m$.

```
static void
make_send_count(struct S_commlist* slist, int slsize) {
  int me=myRank, nn=nOfNodes;
  int i;

  for (i=0; i<slsize; i++) {
    if (slist[i].sid==me)
      NOfSend[slist[i].tag*nn+slist[i].rid] = slist[i].count;
  }
}
```

4.3.23 count_next_particles()

count_next_particles()

The function count_next_particles(), called only from make_comm_count(), scans a part of CommList whose head and size are specified by the arguments rlist and rlsize. The scan is to count the number of primary ($p = 0$) or secondary ($p = 1$) particles of species s which the local node n will receive from nodes and to set the count into TotalPNext$[p][s]$. Therefore, when we find a S_commlist record whose rid is n and which has tag $= t = pS + s$, we add its count to TotalPNext$[p][s]$. Note that since the one-dimensional index of $[p][s]$ for an array $[2][S]$ is $pS + s$, that index is simply t.

```
static void
count_next_particles(struct S_commlist* rlist, int rlsize) {
  int me=myRank, i;

  for (i=0; i<rlsize; i++) {
    if (rlist[i].rid==me)
      TotalPNext[rlist[i].tag] += rlist[i].count;
  }
}
```

4.3.24 oh1_broadcast()

oh1_broadcast_()
oh1_broadcast()

The API functions oh1_broadcast_() for Fortran and oh1_broadcast() for C provide a simulator body calling them with a safe broadcast communications in a family. The function oh1_broadcast() is also used in library functions schedule_particle_exchange() and make_comm_count() to broadcast blocks in CommList[], and build_new_comm() to do it for Neighbors[0] and NeighborsShadow[0]. The functions have the following arguments.

- The input argument pbuf is the pointer to the buffer of data which the local node broadcast to its helpers.
- The output argument sbuf is the pointer to the buffer of data which the local node receives from its helpand by broadcast.
- The input argument pcount is the size (number of data elements) of the data to broadcast to the helpers.

- The input argument `scount` is the size (number of data elements) of the broadcasted data to be received from the helpand. The value of this argument must be equal to the `pcount` argument of corresponding call of the function in the helpand.
- The input argument `ptype` is the MPI data-type of the data to broadcast to the helpers.
- The input argument `stype` is the MPI data-type of the broadcasted data to be received from the helpand. The value of this argument must be equal to the `ptype` argument of corresponding call of the function in the helpand.

The Fortan API `oh1_broadcast_()` simply calls its C counterpart `oh1_broadcast()` which does what we have to do, translating its `ptype` and `stype` arguments into C representation by `MPI_Type_f2c()`.

```
void
oh1_broadcast_(void* pbuf, void* sbuf, int *pcount, int *scount,
               int *ptype, int *stype) {
  oh1_broadcast(pbuf, sbuf, *pcount, *scount,
                MPI_Type_f2c(*ptype), MPI_Type_f2c(*stype));
}
void
oh1_broadcast(void* pbuf, void* sbuf, int pcount, int scount,
              MPI_Datatype ptype, MPI_Datatype stype) {
```

The broadcast in a family consists of a pair of `MPI_Bcast()`, one as the helpand sending the data spceified by `pbuf`, `pcount` and `ptype` to helpers in the communicator `MyComm->prime` with its rank `MyComm->rank`, and the other as a helper receiving the data specified by `sbuf`, `scount` and `stype` from the helpand whose rank is `MyComm->root` in the communicator `MyComm->sec`. Thus, if we performed these two boradcasting without care about their order, e.g., first as the helpand then as a helper, they should be unnecessarily serialized waiting the completions of those for bottommost families, then the second bottommost, and so on, in this example.

Therefore we perform broadcasts in two phases with red-black ordering. That is, each node is assigned a color red (`MyComm->black` = 0) or black (`MyComm->black` = 1) so that a black (red) helpand has red (black) helpers. Then first we perform the broadcasts from black helpands as roots safely to their red helpers which dedicate only receiving in this phase. Then in the second phase, red helpands also safely broadcast to their black helpers.

Note that if a helpand does not have nothing to broadcast (`pcount` = 0), corresponding broadcast is not performed because its helpers know the fact (`scount` = 0). Also note that the root of the family tree does not have a helpand and thus its `MyComm->sec` is `MPI_COMM_NULL`, while leaves does not have helpers with their `MyComm->prime` being `MPI_COMM_NULL`.

```
  if (MyComm->black) {
    if (MyComm->prime!=MPI_COMM_NULL && pcount)
      MPI_Bcast(pbuf, pcount, ptype, MyComm->rank, MyComm->prime);
    if (MyComm->sec!=MPI_COMM_NULL && scount)
      MPI_Bcast(sbuf, scount, stype, MyComm->root, MyComm->sec);
  } else {
    if (MyComm->sec!=MPI_COMM_NULL && scount)
      MPI_Bcast(sbuf, scount, stype, MyComm->root, MyComm->sec);
    if (MyComm->prime!=MPI_COMM_NULL && pcount)
```

```
      MPI_Bcast(pbuf, pcount, ptype, MyComm->rank, MyComm->prime);
    }
  }
```

4.3.25 rebalance1()

rebalance1() The function rebalance1(), called from transbound1() and rebalance2() being the level-2 counterpart of this function, builds the new family tree to rebalance the load among nodes. It also makes an all-to-all type particle transfer schedule to make particles in a subdomain accommodated by the family members responsible for the subdomain, and set NOfRecv[][][] and NOfSend[][][] according to the schedule. The function has three arguments currmode, level and stats whose meanings are as same as those of try_primary1().

```
void
rebalance1(int currmode, int level, int stats) {
  int nn=nOfNodes;
  dint nofp=nOfParticles;
  dint npavefloor=nofp/nn;
  dint npfracin=nofp-npavefloor*nn, npfracout=npfracin;
  dint npavein=npavefloor+(npfracin==0 ? 0 : 1), npaveout=npavein;
  int ns=nOfSpecies;
  int i, j, k, s, bot, pm=Mode_PS(currmode)-1, me=myRank;
  struct S_node *node, *mynode=NodesNext+me, *root;
```

The first job of the function, besides starting time measurement and verbose messaging, is to initialize LessHeap and GreaterHeap by emptying them and clear GreaterHeap.index[] to state that any subdomains are not in GreaterHeap. Note that LessHeap.index[] is never referred to.

```
  if (stats) oh1_stats_time(STATS_REBALANCE, 0);
  Verbose(2,vprint("rebalance"));

  LessHeap.n = GreaterHeap.n = 0;
  for (i=0; i<nn; i++) GreaterHeap.index[i] = 0;
```

Next we split subdomains according to their particle populations and, by push_heap(), push subdomain n such that TotalPGlobal$[n] = P_n$ is less than average to LessHeap and thus make it a leaf of the family tree, and push others to GreaterHeap. Note that the average P/N can be a non-integer and thus we examine with the ceiling of the average $\lceil P/N \rceil$ until we push (P mod N) subdomians to LessHeap, while remainders are examined with the floor $\lfloor P/N \rfloor$. We also push NodesNext$[n]$ such that n goes to LessHeap into NodeQueue[] so that its first members are leaves. In addtion, after copying Nodes$[n]$ into NodesNext$[n]$, we initialize NodesNext$[n]$.child to be NULL indicating no children, and, if we are in primary mode, NodesNext$[n]$.parentid to be -1 to indicate that n does not have a parent currently.

```
  for (i=0,bot=0,node=NodesNext; i<nn; i++,node++) {
    dint npg=TotalPGlobal[i];
    if (npg<npavein) {
      if (--npfracin==0) npavein--;
      push_heap(i, &LessHeap, 0);
```

```
    NodeQueue[bot++] = node;
  } else {
    push_heap(i, &GreaterHeap, 1);
  }
  *node = Nodes[i];
  node->child = NULL;
  if (pm) node->parentid = -1;
}
```

Now we repeatedly pick subdomains from `LessHeap` by `pop_heap()` and thus in lightest-first manner untill `LessHeap` becomes empty. The node responsible for each popped subdomain becomes a helper and is assigned secondary particles so that it becomes accommodating the average number of particles in total. Again the average can be a non-integer and thus first ($P \bmod N$) subdomains will be loaded with ceiling of the average $\lceil P/N \rceil$ while remainders with floor $\lfloor P/N \rfloor$. The helpand of a node n remains unchanged if we are in secondary mode and the subdomain of n's helpand is in `GreaterHeap` from which the helpand's subdomain is removed by `remove_heap()`. Otherwise, the helpand is that having the heaviest load popped from `GreaterHeap` by `pop_heap()`. In both cases, the node n is assigned $a - P_n$ secondary particles, where $a = \lceil P/N \rceil$ for first ($P \bmod N$) nodes or $a = \lfloor P/N \rfloor$ for remainders, and the diffence is set into `NodesNext`$[n]$`.get.sec` $= Q_n^{\mathrm{get}}$. This value is not the actual numbers of particles to be received by n but it will be adjusted according to the number of particles currently accommodated by n as its secondary particle (maybe incidentally) by `schedule_particle_exchange()` afterward to possibly result in negative one meaning to put.

Then, after linking the node n and its helpand m by making elements of `NodesNext`$[n]$ and `NodesNext`$[m]$ for the helpand-helper linkage have appropriate values, $P_m =$ `TotalPGlobal`$[m]$ is decremented by the number of secondary particles assigned to the n, namely $a - P_n$. If the resulting amount is less than the average, which is the average for pushing to `LessHeap` rather than for popping from it, the node m is pushed to `LessHeap` by `push_heap()` and to `NodeQueue`[]. Otherwise, it is returned to `GreaterHeap` also by `push_heap()`.

```
while (LessHeap.n) {
  struct S_node *parent;
  dint npg;
  int get, pid, h;
  j = pop_heap(&LessHeap, 0);
  node = NodesNext + j;
  get = npaveout - TotalPGlobal[j];
  if (--npfracout==0) npaveout--;
  if ((k=node->parentid)>=0 && (h=GreaterHeap.index[k]))
    remove_heap(&GreaterHeap, 1, h);
  else
    k = pop_heap(&GreaterHeap, 1);
  node->get.sec = get;
  parent = NodesNext + k;
  node->parentid = k;  node->parent = parent;
  node->sibling = parent->child;
  parent->child = node;
  npg = (TotalPGlobal[k] -= get);
  if (npg<npavein) {
    if (--npfracin==0) npavein--;
    push_heap(k, &LessHeap, 0);  NodeQueue[bot++] = parent;
```

```
    } else {
      push_heap(k, &GreaterHeap, 1);
    }
  }
```

When we complete the helper assignment, we have at least one node remaining in `GreaterHeap`. We pick the first element `GreaterHeap.node`[1] to make it the root of the family tree. If `GreaterHeap` has two or more elements whose particle amounts are incidentally tie the root's, we make them root's helpers without secondary particle assignment (i.e., $Q_n^{\rm get}$ = `get.sec` = 0) pushing them to `NodeQueue`[]. Finally, the root is pushed to `NodeQueue`[] as its last element.

Note that in this final root-family member addition, a leaf node may be pushed to some midst entry (i.e., not in the topmost sequence) of `NodeQueue`[]. This can be happen if P_n for a node n is accidentaly equal to $\lceil P/N \rceil$ or $\lfloor P/N \rfloor$ and thus is pushed into `GreaterHeap` by definition. Therefore, we cannot stop a top-down traversal of the tree, i.e., tail-to-head scan of `NodeQueue`[] when we find a leaf node.

```
root = NodesNext + GreaterHeap.node[1];
root->parentid = -1;  root->parent = root->sibling = NULL;
root->get.sec = 0;
k = root->id;
for (i=2; i<=GreaterHeap.n; i++) {
  j = GreaterHeap.node[i];
  node = NodesNext + j;
  node->get.sec = 0;
  node->parentid = k;  node->parent = root;
  node->sibling = root->child;  root->child = node;
  NodeQueue[bot++] = node;
}
NodeQueue[bot] = root;
```

Now we have the family tree whose every node n has correct settings of $Q_n^{\rm get}$ in `NodesNext`[n]`.get.sec` with respect to the number of secondary particles the node should accommodate. From now we temporalily switches to local operations. For the local node n, We let

$$\texttt{NodesNext}[n]\texttt{.get.prime} = R_n^{\rm get} = P_n - \sum_{s=0}^{S} q(n)[0][s][n] = P_n - \sum_{s=0}^{S} \texttt{NOfPLocal}[0][s][n]$$

to represent the number of primary particles to be received using `count_real_stay()` to calculate $\sum_{s=0}^{S} q(n)[0][s][n]$[35]. Note that we set the calculation result into `NodesNext`[n]`.stay.prime` so that it is referred to by a level-4p function `make_recv_list()`.

Then, with the setting of $R_n^{\rm get}$ for the local node n and $Q_m^{\rm get}$ for all nodes $m \in [0, N-1]$ (especially $m \in H(n)$), we call `schedule_particle_exchange()` with the argument `reb` as follows, unless `Special_Pexc_Sched()` is true with the argument `level` to mean that the caller of `rebalance1()` will do its own particle exchange scheduling.

- 1 means we are in secondary mode with normal accommodataion (`currmode` = `MODE_NORM_SEC`) and thus the old family tree kept in `Nodes`[] is correct. Thus

[35] We may rely on `stay.prime` if we were in secondary mode but we always call `count_real_stay()` for the sake of simplicity.

`schedule_particle_exchange()` and `sched_comm()` consult the tree to find old family menbers for neighboring regions.

- 2 means we are in primary mode with normal accommodation (`currmode` = `MODE_NORM_PRI`) and thus the old family tree is obsolete but particles moving to a subdomain n should be found only in the nodes whose primary subdomain is a neighbor of the subdomain n. Thus `schedule_particle_exchange()` and `sched_comm()` scan these nodes as senders.
- 3 means we are in primary or secondary mode with anywhere accommodation and thus any nodes may have particles in any subdomain (`Mode_Is_Any()` for `currmode` is true) because of initial particle distribution or particle injections and so on. Thus `schedule_particle_exchange()` and `sched_comm()` scan all nodes as potential senders.

Now we have the transfer schedule for the local node in `CommList[]` a part of which is obtained by broadcast in the old family, and thus we can now create communicators for newly created families by `build_new_comm()`.

```
    mynode->get.prime = TotalPGlobal[me] -
                        (mynode->stay.prime=count_real_stay(NOfPLocal+me));
    if (Special_Pexc_Sched(level)) return;
    schedule_particle_exchange(currmode==MODE_NORM_SEC ?
                               1 : (currmode==MODE_NORM_PRI ? 2 : 3));
    build_new_comm(currmode, level, 1, stats);
  }
```

4.3.26 `build_new_comm()`

`build_new_comm()` The function `build_new_comm()`, called from `rebalance1()` and level-4 (or higher) library functions with their own particle exchange scheduling mechanisms, creates MPI communicators for new family tree created by `rebalance1()`. The arguments of this function, except for `nbridx`, are exactly same as those of `rebalance1()`.

First of all, since the old family tree is no more useful[36], we exchange `Nodes[]` and `NodesNext[]`.

```
  void
  build_new_comm(int currmode, int level, int nbridx, int stats) {
    int bot=nOfNodes-1, me=myRank;
    struct S_node *node, *ch;
    struct S_node *mynode=NodesNext+me, *root=NodeQueue[bot];
    int oldparent=Mode_PS(currmode) ? Nodes[me].parentid : -1;
    int i, j;
    MPI_Group grpw=GroupWorld, grp;

    node = Nodes;  Nodes = NodesNext;  NodesNext = node;
```

[36] If without position-aware paritcle management. If with it, the old family tree will be referred to after the call of `build_new_comm` but anyway the tree is kept in `NodesNext` for the referencce.

Then we creates communicators as follows. Each family should have its own MPI communicator for the broadcast subdomain field data and/or that of its borders and particle transfer schedule from its helpand to helpers, and the (all-)reduce of the current and/or charge density of the subdomain. Since a node may belong two families, one as the helpand and the other as a helper, if collective communications of both families were performed in a careless order we could have unnecessarily heavy serialization. For example, if all nodes perform a collective communication as the helpand first and then do as a helper, the communication is serialized in the bottom-up manner. Reversing the order does not help us simply causing top-down serialization. Thus we assign one of two colors, black and red, to families so that the color of a family is differnt from that of its direct ancenstral and direct descendant families. By this coloring, first we perform the collective communications of black families which have no dependencies each other, and then those of red families without serialization, as discussed in §4.3.24.

To build the communicators and to assign colors, we traverse the helper tree in top-down manner starting from the bottom of `NodeQueue[]` as done in `try_stable1()`, after freeing communicators for the old tree. To create the communicator for the family rooted by the node n visited in the i-th ($i \geq 0$) iteration for non-leaf nodes, we gather the MPI ranks of the family members into `TempArray[]` from which the MPI group of the family is created by `MPI_Group_incl()` with `GroupWorld`, and then the MPI communicator is created from the group by `MPI_Comm_create()`. Then the created communicator stored in `Comms.body`$[i]$, and `Nodes`$[n]$`.comm.prime` and `Nodes`$[c]$`.comm.sec` are set to i for all n's helper nodes c. The rank of the node n in the family communicator is obtained by `MPI_Group_translate_ranks()` from n in `GroupWorld`, and is stored in `Nodes`$[n]$`.comm.rank` to use it as the root node rank for collective communications. For the family tree root node r, `Nodes`$[r]$`.comm.sec` is set to -1, while leaf nodes l have `Nodes`$[l]$`.comm.prime` $= -1$, both of them meaning they don't belong to communicators as a helper or the helpand.

The color for the topmost familiy is red and thus `Nodes`$[r]$`.comm.black` for the root node r is 0 while `Nodes`$[c]$`.comm.black` is 1 for all r's helper nodes c. Then the colors are set into `Nodes`$[n]$`.comm.black` so that it is reversed from the `Nodes`$[parent(n)]$`.comm.black`.

```
if (stats) oh1_stats_time(STATS_REB_COMM, 0);
for (i=0; i<Comms.n; i++) {
  if (Comms.body[i] != MPI_COMM_NULL)
    MPI_Comm_free(Comms.body+i);
}
root->comm.black = 0;  root->comm.sec = -1;
for (i=0; bot>=0; bot--) {
  int black, rid;
  node = NodeQueue[bot];
  if (!(ch=node->child))  continue;   /* a leaf may reside below some non-
                                         leaves in NodeQueue when its number
                                         of primaries is equal to the
                                         average */
  node->comm.prime = i;
  rid = TempArray[0] = node->id;
  black = 1 - node->comm.black;
  for (j=1; ch; ch=ch->sibling, j++) {
    TempArray[j] = ch->id;
    ch->comm.prime = -1;  ch->comm.sec = i;  ch->comm.black = black;
  }
  MPI_Group_incl(grpw, j, TempArray, &grp);
```

```
  MPI_Group_translate_ranks(grpw, 1, &rid, grp, &(node->comm.rank));
  MPI_Comm_create(MCW, grp, Comms.body+i);
  MPI_Group_free(&grp);
  i++;
}
```

Then, after letting `Comms.n` be the number of families (non-leaf nodes) in the tree for the freeing operation in the next occasion, we let `FamIndex`[] and `FamMembers`[] represent $F(m)$ for all $m \in [0, N)$, which we have just built, if `oh1_families()` has been called beforehand to make these arrays non-`NULL`. That is, for all $m \in [0, N)$ we let;

$$
\begin{aligned}
&i_m = \texttt{FamIndex}[m] = \sum_{j=0}^{m-1} |F(j)| \\
&\texttt{FamMembers}[i_m] = m \\
&\{\texttt{FamMembers}[i_m{+}k] \,|\, k \in \{1, H(m)\}\} = H(m)
\end{aligned}
$$

visiting `Nodes`$[m]$ and the chain of $H(m)$ starting from `Nodes`$[m]$`.child`. We also let `FamMembers`$[2N{-}1] = r$ to show the family tree root.

```
Comms.n = i;

if (FamIndex) {
  int *fidx = FamIndex,  *fmem = FamMembers;
  int nn = nOfNodes, j;
  for (i=0,j=0; i<nn; i++) {
    fidx[i] = j;
    fmem[j++] = i;
    for (ch=Nodes[i].child; ch; ch=ch->sibling, j++)  fmem[j] = ch->id;
  }
  fidx[nn] = j;  fmem[j] = root->id;
}
```

Next, we set `MyComm` elements for the local node n referring to the element of `Nodes`$[n]$`.comm` and `Comms.body`[] having its communicators. Then if the library is called from a C-coded simulator body which needs `MyComm` through its shadow() `MyCommC` of non-`NULL`, we copy `MyComm` into `MyCommC`. Similary, if the library is called from Fortran and thus we have `MyCommF` of non-`NULL`, the elements in `MyComm` are copied into those in `MyCommF` translating communicators into Fortran forms by `MPI_Comm_c2f()`.

```
MyComm->prime =
  mynode->comm.prime<0 ? MPI_COMM_NULL : Comms.body[mynode->comm.prime];
MyComm->sec =
  mynode->comm.sec<0 ? MPI_COMM_NULL : Comms.body[mynode->comm.sec];
MyComm->rank = mynode->comm.prime<0 ? -1 : mynode->comm.rank;
MyComm->black = mynode->comm.black;
if ((node=mynode->parent))  MyComm->root = node->comm.rank;
else MyComm->root = -1;
if (MyCommC) *MyCommC = *MyComm;
if (MyCommF) {
  MyCommF->prime = MPI_Comm_c2f(MyComm->prime);
  MyCommF->sec   = MPI_Comm_c2f(MyComm->sec);
  MyCommF->rank  = MyComm->rank;
```

```
  MyCommF->root  = MyComm->root;
  MyCommF->black = MyComm->black;
}
```

Next, we broadcast `Neighbors`[0] = `DstNeighbors` to the helpers of the local node which receive it in `Neighbors`[i], where i = `nbridx` = 1 when the function is called from `rebalance1()`, while it can be i = 2 when called from a higher level function, e.g., `make_recv_list()` for position-aware particle management to keep the neighbors of the old parent. Similarly, we broadcast `NeighborsShadow`[0][] to the helpers to let them have the array elements in their `NeighborsShadow`[1][] after pushing their old values to `NeighborsShadow`[2][], if the array is non-`NULL` to mean `oh1_neighbors()` has been called beforehand to show the neighbor information to a simulator body.

We also let `RegionId`[1] = `Nodes`[n]`.parentid` to notify the simulator body of the MPI rank of the helpand through its shadow `SubdomainId`[1].

```
oh1_broadcast(Neighbors[0], Neighbors[nbridx], OH_NEIGHBORS, OH_NEIGHBORS,
              MPI_INT, MPI_INT);
if (NeighborsShadow) {
  int (*nb)[OH_NEIGHBORS] = NeighborsShadow;
  for (i=0; i<OH_NEIGHBORS; i++)  nb[2][i] = nb[1][i];
  oh1_broadcast(nb[0], nb[1], OH_NEIGHBORS, OH_NEIGHBORS, MPI_INT, MPI_INT);
}
SubdomainId[1] = RegionId[1] = mynode->parentid;
```

Finally, if `Special_Pexc_Sched()` is true for the argument `level` meaning the caller of `build_new_comm()` has its own particle exchange scheduling mechanism, we simply finish the function body. Otherwise, we call `make_comm_count()` giving its argument `reb` = 1 to indicate the family tree is rebuilt and passing *parent*(n) of the local node n in the old family tree, if it was valid and useful, through its argument `oldparent`, to have `NOfRecv`[][][] and `NOfSend`[][][] as the output of `oh1_transbound()` if `level` = 1, or for non-neighboring particle transfers if `Mode_Is_Norm()` for `currmode` is false.

```
  if (!Special_Pexc_Sched(level))
    make_comm_count(currmode, level, 1,
                    (Mode_Is_Norm(currmode) ? oldparent : -1), stats);
}
```

4.3.27 `push_heap()`

`push_heap()` The function `push_heap`, called only from `rebalance1()`, adds the element for the subdomain n to the heap h = `heap`, which is `GreaterHeap` if g = `greater` = 1 or `LessHeap` otherwise, according to the number of particles in n, i.e., P_n = `TotalPGlobal`[n]. The addtion to a heap whose number of elements becomes k is done by traversing the heap tree from its k-th node (leaf) to its root inserting the values to be added shifting those recorded on the path downward to keep the invariant of the tree, a parent is greater/less than its children.

The upward path from the tree node stored in h`.node`[k] is depicted by the binary representaion of k, namely $k(l-1)\dots k(0)$ where $k(l-1) = 1$. That is, the binary representation of the index of its parent of k is $k(l-1)\dots k(1)$, that of the grand parent is $k(l-1)\dots k(2)$ and so on to the root h`.node`[1] whose index is represented by $k(l-1) = 1$. Thus we go up the tree from k until we find minimum i such that, for all $j \in [1, i]$, $P_n > P_{n(j)}$ if

$g = 1$ or $P_n \leq P_{n(j)}$ otherwise where $n(j) = h.\texttt{node}[k/2^j]$, shifting $h.\texttt{node}[k/2^j]$ down to $h.\texttt{node}[k/2^{j+1}]$. (If such i is not found because $h.\texttt{node}[k/2]$ does not hold the inequation above, we let $i = 0$.) Then we store n into $h.\texttt{node}[k/2^i]$ and set $h.\texttt{index}[n] = k/2^i$.

```
static void
push_heap(int r, struct S_heap* heap, int greater) {
  int n=heap->n, *hnode=heap->node, *index=heap->index;
  dint np=TotalPGlobal[r];
  int m, q, g;

  heap->n = ++n;
  for (; n>1; n=m) {
    m = n>>1;  q = hnode[m];
    g = (np>TotalPGlobal[q]) ? 1 : 0;
    if (g!=greater) break;
    hnode[n] = q;  index[q] = n;
  }
  hnode[n] = r;  index[r] = n;
}
```

4.3.28 pop_heap()

pop_heap() The function pop_heap(), called only from rebalance1(), removes the root node of heap, which is GreaterHeap if the argument greater is 1 or LessHeap otherwise, by remove_heap() and returns the subdomain ID which had been stored in the root node.

```
static int
pop_heap(struct S_heap* heap, int greater) {
  int pop=heap->node[1];

  remove_heap(heap, greater, 1);
  return(pop);
}
```

4.3.29 remove_heap()

remove_heap() The function remove_heap(), called from rebalance1() directly or through pop_heap(), removes the element $r = \texttt{rem}$ of the heap $h = \texttt{heap}$ which is GreaterHeap if $g = \texttt{greater} = 1$ or LessHeap otherwise. The removal of $h.\texttt{node}[r]$ of the heap whose number of element becomes $k - 1$ is done by temporalily moving $h.\texttt{node}[k]$ to $h.\texttt{node}[r]$ and rearranging the subtree rooted by r so that it keeps the invariant that a parent is greater/less than its children.

Since a node $h.\texttt{node}[i]$ has two children in $h.\texttt{node}[2i]$ and $h.\texttt{node}[2i+1]$ (if $k > 2i+1$), the rearrangement of the subtree rooted by i is peformed as follows if $g = 1$ (or $g = 0$).

(1) If $2i + 1 > k$ and $P_{2i+1} = \texttt{TotalPGlobal}[2i+1]$ is the maximum (minimum) in three members, we exchange $h.\texttt{node}[2i+1]$ and $h.\texttt{node}[i]$ and rearrange the subtree rooted by $2i + 1$ recursively.

(2) Otherwise, if $2i > k$ and $P_{2i} > P_i$ ($P_{2i} \leq P_i$), we exchange $h.\texttt{node}[2i]$ and $h.\texttt{node}[i]$ and rearrange the subtree rooted by $2i$ recursively.

(3) Otherwise, we complete the procedure.

```
static void
remove_heap(struct S_heap* heap, int greater, int rem) {
  int n=heap->n, *hnode=heap->node, *index=heap->index;
  int id=hnode[n];
  dint np=TotalPGlobal[id];
  int i;

  heap->n = --n;  index[hnode[rem]] = 0;
  if (rem>n) return;
  for (i=rem; ; ) {
    int left=(i<<1), right=left+1;
    if (right<=n) {
      int lid=hnode[left], rid=hnode[right];
      dint lnp=TotalPGlobal[lid], rnp=TotalPGlobal[rid];
      int cgl=(np>lnp)?1:0, cgr=(np>rnp)?1:0, lgr=(lnp>rnp)?1:0;
      if (cgl==greater) {
        if (cgr==greater) {
          hnode[i] = id;  index[id] = i;  return;
        } else {
          hnode[i] = rid;  index[rid] = i;  i = right;
        }
      } else if (lgr==greater) {
        hnode[i] = lid;  index[lid] = i;  i = left;
      } else {
        hnode[i] = rid;  index[rid] = i;  i = right;
      }
    } else {
      if (left<=n) {
        int lid=hnode[left];
        int cgl=(np>TotalPGlobal[lid])?1:0;
        if (cgl==greater) {
          hnode[i] = id;  index[id] = i;
        } else {                /* we know left node has no children. */
          hnode[i] = lid;  index[lid] = i;
          hnode[left] = id;  index[id] = left;
        }
      } else {
        hnode[i] = id;  index[id] = i;
      }
      return;
    }
  }
}
```

4.3.30 `oh1_accom_mode()`

`oh1_accom_mode_()` `oh1_accom_mode()`

The API functions `oh1_accom_mode_()` for Fortran and `oh1_accom_mode()` for C simply retuns the value of `accMode` to a simulator body calling them to let it know the accommodation mode is normal or anywhere.

```
int
oh1_accom_mode_() {
  return(accMode);
}
int
oh1_accom_mode() {
  return(accMode);
}
```

4.3.31 oh1_all_reduce()

oh1_all_reduce_()
oh1_all_reduce()

The API functions `oh1_all_reduce_()` for Fortran and `oh1_all_reduce()` for C provide a simulator body calling them with a safe all-reduce communications in a family. The functions have the following arguments.

- The input arguments `pbuf` and `sbuf` are the pointers to the buffers of data to be all-reduced in the primary and secondary families which the local node belongs to respectively.
- The input arguments `pcount` and `scount` are the sizes (number of data elements) of the data to be all-reduced in the primary and secondary families which the local node belongs to respectively. The `pcount` for a helpand must be equal to `scount` for its helpers.
- The input arguments `ptype` and `stype` are the MPI data-types of the data to be all-reduced in the primary and secondary families which the local node belongs to respectively. The `ptype` for a helpand must be equal to `stype` for its helpers.
- The input arguments `pop` and `sop` are the MPI's reduction operators of the data to be all-reduced in the primary and secondary families which the local node belongs to respectively. The `pop` for a helpand must be equal to `sop` for its helpers.

The Fortan API `oh1_all_reduce_()` simply calls its C counterpart `oh1_all_reduce()` which does what we have to do, translating its `ptype` and `stype` arguments into C representation by `MPI_Type_f2c()`, and `pop` and `sop` arguments by `MPI_Op_f2c()`.

The function calls `MPI_Allreduce()` twice with `MPI_IN_PLACE` option, as the parent and a child, in the way similar to `oh1_broadcast()` to avoid serialization.

```
void
oh1_all_reduce_(void *pbuf, void *sbuf, int *pcount, int *scount,
                int *ptype, int *stype, int *pop, int *sop) {
  oh1_all_reduce(pbuf, sbuf, *pcount, *scount,
                 MPI_Type_f2c(*ptype), MPI_Type_f2c(*stype),
                 MPI_Op_f2c(*pop), MPI_Op_f2c(*sop));
}
void
oh1_all_reduce(void *pbuf, void *sbuf, int pcount, int scount,
               MPI_Datatype ptype, MPI_Datatype stype,
               MPI_Op pop, MPI_Op sop) {

  if (MyComm->black) {
    if (MyComm->prime!=MPI_COMM_NULL)
      MPI_Allreduce(MPI_IN_PLACE, pbuf, pcount, ptype, pop, MyComm->prime);
```

```
    if (MyComm->sec!=MPI_COMM_NULL)
      MPI_Allreduce(MPI_IN_PLACE, sbuf, scount, stype, sop, MyComm->sec);
  } else {
    if (MyComm->sec!=MPI_COMM_NULL)
      MPI_Allreduce(MPI_IN_PLACE, sbuf, scount, stype, sop, MyComm->sec);
    if (MyComm->prime!=MPI_COMM_NULL)
      MPI_Allreduce(MPI_IN_PLACE, pbuf, pcount, ptype, pop, MyComm->prime);
  }
}
```

4.3.32 oh1_reduce()

oh1_reduce_() oh1_reduce()

The API functions oh1_reduce_() for Fortran and oh1_reduce() for C provide a simulator body calling them with a safe one-way reduction communications in a family. The functions have the following arguments.

- The input arguments pbuf and sbuf are the pointers to the buffers of data to be reduced in the primary and secondary families which the local node belongs to respectively.
- The input arguments pcount and scount are the sizes (number of data elements) of the data to be reduced in the primary and secondary families which the local node belongs to respectively. The pcount for a helpand must be equal to scount for its helpers.
- The input arguments ptype and stype are the MPI data-types of the data to be reduced in the primary and secondary families which the local node belongs to respectively. The ptype for a helpand must be equal to stype for its helpers.
- The input arguments pop and sop are the MPI's reduction operators of the data to be reduced in the primary and secondary families which the local node belongs to respectively. The pop for a helpand must be equal to sop for its helpers.

The Fortan API oh1_reduce_() simply calls its C counterpart oh1_reduce() which does what we have to do, translating its ptype and stype arguments into C representation by MPI_Type_f2c(), and pop and sop arguments by MPI_Op_f2c().

The function calls MPI_Reduce() twice, as the helpand with MPI_IN_PLACE option and as a helper, in the way similar to oh1_broadcast() to avoid serialization.

```
void
oh1_reduce_(void *pbuf, void *sbuf, int *pcount, int *scount,
            int *ptype, int *stype, int *pop, int *sop) {
  oh1_reduce(pbuf, sbuf, *pcount, *scount,
             MPI_Type_f2c(*ptype), MPI_Type_f2c(*stype),
             MPI_Op_f2c(*pop), MPI_Op_f2c(*sop));
}
void
oh1_reduce(void *pbuf, void *sbuf, int pcount, int scount,
           MPI_Datatype ptype, MPI_Datatype stype, MPI_Op pop, MPI_Op sop) {

  if (MyComm->black) {
    if (MyComm->prime!=MPI_COMM_NULL)
      MPI_Reduce(MPI_IN_PLACE, pbuf, pcount, ptype, pop, MyComm->rank,
```

```
                MyComm->prime);
    if (MyComm->sec!=MPI_COMM_NULL)
      MPI_Reduce(sbuf, NULL, scount, stype, sop, MyComm->root, MyComm->sec);
  } else {
    if (MyComm->sec!=MPI_COMM_NULL)
      MPI_Reduce(sbuf, NULL, scount, stype, sop, MyComm->root, MyComm->sec);
    if (MyComm->prime!=MPI_COMM_NULL)
      MPI_Reduce(MPI_IN_PLACE, pbuf, pcount, ptype, pop, MyComm->rank,
                MyComm->prime);
  }
}
```

4.3.33 oh1_init_stats()

oh1_init_stats_()
oh1_init_stats()

The API functions oh1_init_stats_() for Fortran and oh1_init_stats() for C initialize the data structure Stats for statistics and start the first time measurement specified by the arguments key and ps, if statsMode $\neq 0$. The initialization is done by calling clear_stats() twice for subotal and total substructures in Stats. The time measurement is started by setting $2k + p$ into Stats.curr.time.key where $k =$ key and $p =$ ps indicating whether the first interval is for primary ($p = 0$) or secondary ($p = 1$) execution, and setting the current wall-clock time obtained by MPI_Wtime() into Stats.curr.time.value. We also clear all the elements in Stats.curr.time.ev[] to indicate no time measurement intervals have completed.

Another initialization is to create the MPI data-type T_StatsTime for the reduction on timing statistics, as a MPI_BYTE sequence of sizeof(struct␣S_statstime) by MPI_Type_contiguous() followed by MPI_Type_commit(). The pairwise reduction is peformed by the function stats_reduce_time() which MPI_Op_create() bind to Op_StatsTime to be given to MPI_Reduce(). The other reducer function is stats_reduce_part() for particle transfer statistics, which is bound to Op_StatsPart.

```
void
oh1_init_stats_(int *key, int *ps) {
  if (statsMode) oh1_init_stats(*key, *ps);
}
void
oh1_init_stats(int key, int ps) {
  int i;

  if (!statsMode)  return;
  clear_stats(&Stats.subtotal);
  clear_stats(&Stats.total);
  Stats.curr.time.key = (key<<1) + ps;
  Stats.curr.time.value = MPI_Wtime();
  for (i=0; i<(STATS_TIMINGS<<1)+1; i++) Stats.curr.time.ev[i] = 0;
  MPI_Type_contiguous(sizeof(struct S_statstime), MPI_BYTE, &T_StatsTime);
  MPI_Type_commit(&T_StatsTime);
  MPI_Op_create(stats_reduce_time, 1, &Op_StatsTime);
  MPI_Op_create(stats_reduce_part, 1, &Op_StatsPart);
}
```

4.3.34 clear_stats()

clear_stats() This function, called from oh1_init_stats() and oh1_show_stats(), clears statistics recorded in a S_statstotal structure specified by its argument stotal, which should be Stats.total or Stats.subtotal. Clearing its element arrays time[$2K_t$] and part[K_p], where K_t = STATS_TIMINGS and K_p = STATS_PARTS, is commonly done by letting leaf elements max and total be zero and min be the absolute maximum value, DBL_MAX for time[] and INT_MAX for part[]. For times[k], an additional zero-clearing is also done for the leaf element ev to indicate that the interval corresponging to the key k is not executed. As for part[], a special treatment is taken for keys STATS_PART_PRIMARY and STATS_PART_SECONDARY so that their elements min is set to 0 rather than INT_MAX because they act as counters of execution mode transition.

```
static void
clear_stats(struct S_statstotal *stotal) {
  int i;
  struct S_statstime *st = stotal->time;
  struct S_statspart *sp = stotal->part;

  for (i=0; i<STATS_TIMINGS<<1; i++) {
    st[i].ev = 0;
    st[i].min = DBL_MAX;
    st[i].max = 0.0;
    st[i].total = 0.0;
  }
  for (i=0; i<STATS_PARTS; i++) {
    sp[i].min = INT_MAX;
    sp[i].max = 0;
    sp[i].total = 0;
  }
  sp[STATS_PART_PRIMARY].min = sp[STATS_PART_SECONDARY].min = 0;
}
```

4.3.35 oh1_stats_time()

oh1_stats_time_() oh1_stats_time() The API functions oh1_stats_time_() for Fortran and oh1_stats_time() for C finishe the time measurement of an interval and start that of the next interval, if statsMode $\neq 0$. The function oh1_stats_time() is also called from level-1 library functions transbound1(), try_stable1() and rebalance1(), and level-2 functions try_primary2(), exchange_particles(), move_to_sendbuf_primary() and move_to_sendbuf_secondary() to measure time consumed in the libraray. The other function that calls oh1_stats_time() is oh1_show_stats() but it is not for starting measurement but only for finishing that of the last interval.

The key k_f for the interval whose time measurement is to be finished is recorded in Stats.curr.time.key and thus we calculate the time consumed in the interval as the difference of the starting time recorded in the leaf element value of Stats.curr.time and the current wall-time obtained by MPI_Wtime() which is then recorded into value. The calculated time is recorded into the leaf element val[k_f] while another leaf element ev[k_f] is turned to 1 to indicate the interval is executed in the current simulation step.

Finally, to start the measurement of the new interval specified by the arguments key = k_s and p = ps, we store the key for the interval $2k_s + p$ where p indicate whether the new interval is for primary ($p = 0$) or secondary ($p = 1$) execution.

```
void
oh1_stats_time_(int *key, int *ps) {
  if (statsMode)  oh1_stats_time(*key, *ps);
}
void
oh1_stats_time(int key, int ps) {
  double t;
  int k=Stats.curr.time.key;

  if (!statsMode)  return;
  t = MPI_Wtime();
  Stats.curr.time.val[k] = t - Stats.curr.time.value;
  Stats.curr.time.ev[k] = 1;
  Stats.curr.time.value = t;
  Stats.curr.time.key = (key<<1) + ps;
}
```

4.3.36 stats_primary_comm()

stats_primary_comm() The function stats_primary_comm(), called only from try_primary1() when it finds the next simulation step is executed in primary mode, calculates the local statistics of particle transfer to update Stats.curr.part[] by scanning NOfPrimaries[][][] and NOfPLocal[][][] by stats_comm(). Then the element Stats.curr.part[STATS_PART_PRIMARY] is set to either 1, 2 or 3, depending on we were in primary (1) or secondary (2) mode with normal accommodation, or anywhere accommodataion (3).

```
static void
stats_primary_comm(int currmode) {
  stats_comm(NOfPrimaries, NOfPLocal, Stats.curr.part, nOfSpecies*2);
  Stats.curr.part[STATS_PART_PRIMARY] =
    (currmode==MODE_ANY_PRI) ? 3 : Mode_PS(currmode)+1;
}
```

4.3.37 stats_secondary_comm()

stats_secondary_comm() The function stats_secondary_comm(), called only from make_comm_count(), calculates the local statistics of particle transfer to update Stats.curr.part[] by scanning at first NOfRecv[0][][] and NOfSend[0][][] by stats_comm() for primary particles, and then by scanning NOfRecv[1][][] and NOfSend[1][][] for secondary particles. Then Stats.curr.part[STATS_PART_SECONDARY] is set to either 1, 2 or 3, depending on we were in primary (1) mode, secondary mode without rebalancing (2), or with that (3).

```
static void
stats_secondary_comm(int currmode, int reb) {
  int ns=nOfSpecies, nnns=nOfNodes*ns;

  stats_comm(NOfRecv, NOfSend, Stats.curr.part, ns);
  stats_comm(NOfRecv+nnns, NOfSend+nnns,
             Stats.curr.part+STATS_PART_MOVE_SEC_MIN, ns);
  Stats.curr.part[STATS_PART_SECONDARY] =
```

```
    Mode_PS(currmode) ? (reb ? 3 : 2) : 1;
}
```

4.3.38 stats_comm()

stats_comm()

The function stats_comm(), called from stats_primary_comm() or stats_secondary_comm(), calculates the local statistics of particle transfer to update scp[] = σ[] which points Stats.curr.part[0] for the call from stats_primary_part() or the first call from stats_secondary_comm(), or Stats.curr.part[STATS_PART_MOVE_SEC_MIN] for the the second call from stats_secondary_comm(). The calculation is done by scanning argument arrays nrecv[S'][N] and nsend[S'][N], where S' = ns, which are set by callers as follows.

- When called from stats_primary_comm(), nrecv[S'][N] = NOfPrimaries[2][S][N] and nsend[S'][N] = NOfPLocal[2][S][N] because $S' = 2S$.
- When called from stats_secondary_comm() as its first call, nrecv[S'][N] = (NOfRecv[0])[S][N] and nsend[S'][N] = (NOfSend[0])[S][N] because $S' = S$.
- When called from stats_secondary_comm() as its second call, nrecv[S'][N] = (NOfRecv[1])[S][N] and nsend[S'][N] = (NOfSend[1])[S][N] because $S' = S$.

Then we define $recv(m)$ and $send(m)$ for the local node n as the numbers of particles receiving from and sending to the node m.

$$recv(m) = \sum_{s=0}^{S'-1} \texttt{nsend}[s][m] \qquad send(m) = \sum_{s=0}^{S'-1} \texttt{nrecv}[s][m]$$

With the definitions above, we update the statistics $\sigma[k]$ as follows.

- The elements $\sigma[k]$ for k = STATS_PART_MOVE_PRI_x ($x \in$ {MIN, MAX, AVE}) are set to the followings.

  $$\sigma[\texttt{STATS\_PART\_MOVE\_PRI\_MIN}] = \min\{recv(m) \,|\, m \neq n,\ recv(m) > 0\}$$
  $$\sigma[\texttt{STATS\_PART\_MOVE\_PRI\_MAX}] = \max\{recv(m) \,|\, m \neq n,\ recv(m) > 0\}$$
  $$\sigma[\texttt{STATS\_PART\_MOVE\_PRI\_AVE}] = |\{recv(m) \,|\, m \neq n,\ r(m) > 0\}|$$

  That is, they are the minimum and maximum numbers of particles the local node receives from other nodes, and the number of nodes from which the local node receives some (non-zero) particles.

- The elements $\sigma[k]$ for k = STATS_PART_GET_PRI_x ($x \in$ {MIN, MAX}) and k = STATS_PART_PG_PRI_AVE are commonly set to the following *get*, the total number of particles the local node n receives from other nodes.

  $$get = \sum_{m=0}^{N-1} recv(m) - recv(n)$$

- The elements $\sigma[k]$ for k = STATS_PART_PUT_PRI_x ($x \in$ {MIN, MAX}) are commonly set to the following *put*, the total number of particles the local node n sends to other nodes.

  $$put = \sum_{m=0}^{N-1} send(m) - send(n)$$

Note that the element σ[STATS_PART_y_PRI_x] corresponds to Stats.curr.part[STATS_PART_y_SEC_x] when this function is called from stats_secondary_comm() as its second call, because σ[] points Stats.curr.part + STATS_PART_MOVE_SEC_MIN. Also note that the local staticstics above will be gathered (reduced) from all nodes by update_stats().

```
static void
stats_comm(int* nrecv, int* nsend, dint* scp, int ns) {
  int i, s, nn=nOfNodes, me=myRank;
  int get=0, put=0, minmove=INT_MAX, maxmove=0, nmove=0;

  for (i=0; i<nn; i++) {
    int g=0, p=0, *npr=nrecv, *nps=nsend;
    for (s=0; s<ns; s++,npr+=nn,nps+=nn) {
      g += nrecv[i];                          /* nrecv[s][i] */
      p += nsend[i];                          /* nsend[s][i] */
    }
    if (i!=me) {
      get += g;  put += p;
      if (g>0) {
        if (g<minmove) minmove = g;
        if (g>maxmove) maxmove = g;
        nmove++;
      }
    }
    if (minmove>maxmove) minmove = 0;
    scp[STATS_PART_MOVE_PRI_MIN] = minmove;
    scp[STATS_PART_MOVE_PRI_MAX] = maxmove;
    scp[STATS_PART_MOVE_PRI_AVE] = nmove;
    scp[STATS_PART_GET_PRI_MIN] = scp[STATS_PART_GET_PRI_MAX]
                                = scp[STATS_PART_PG_PRI_AVE] = get;
    scp[STATS_PART_PUT_PRI_MIN] = scp[STATS_PART_PUT_PRI_MAX] = put;
  }
}
```

4.3.39 oh1_show_stats()

oh1_show_stats_()
oh1_show_stats()

The API functions oh1_show_stats_() for Fortran and oh1_show_stats() for C at first finishes the last interval of execution time measurement by oh1_stats_time() giving it a dummy interval key STATS_TIMINGS and then performs a barrier synchronization by MPI_Barrier(), if statsMode $\neq 0$. Then, if statsMode $= 2$ ordering periodical statistics printing, update the statistics recorded in Stats.subtotal by update_stats() giving it arguments including currmode of this function's own. Then, if we reach the end of period specified by r = reportIteration, i.e., (step mod r) $= 0$ with the argument step having the current simulation step number, we print statistics by print_stats() before clear the statistics by clear_stats(). Finally, regardless of the value of statsMode, we update total staticstics data recorded in Stats.total by update_stats() and do MPI_Barrier() again.

```
void
oh1_show_stats_(int *step, int *currmode) {
  if (statsMode)  oh1_show_stats(*step, *currmode);
```

```
}
void
oh1_show_stats(int step, int currmode) {

  if (!statsMode)  return;
  oh1_stats_time(STATS_TIMINGS,0);
  MPI_Barrier(MCW);
  if (statsMode==2) {
    update_stats(&Stats.subtotal, step, currmode);
    if (step%reportIteration == 0) {
      print_stats(&Stats.subtotal, step, reportIteration);
      clear_stats(&Stats.subtotal);
    }
  }
  update_stats(&Stats.total, step, currmode);
  MPI_Barrier(MCW);
}
```

4.3.40 Macro `Round()`

`Round()` The macro `Round()`, used in `update_stats()` and `print_stats()`, divides the numerator $\mathcal{N}$ = `NUM` by the denominator $\mathcal{D}$ = `DEN` and round the quotient to have

$$round(\mathcal{N}/\mathcal{D}) = \lfloor \mathcal{N}/\mathcal{D} + 0.5 \rfloor = \left\lfloor \frac{\mathcal{N} + \lfloor \mathcal{D}/2 \rfloor}{\mathcal{D}} \right\rfloor$$

whose correctness is proved as follows. Let $\mathcal{Q}$ and $\mathcal{R}$ be the followings.

$$\mathcal{Q} = \left\lfloor \frac{\mathcal{N} + \lfloor \mathcal{D}/2 \rfloor}{\mathcal{D}} \right\rfloor \qquad \mathcal{R} = \mathcal{N} + \lfloor \mathcal{D}/2 \rfloor - \mathcal{Q}\mathcal{D}$$

That is $\mathcal{R}$ is the remainder of the integer division. If $\mathcal{R} \geq \lfloor \mathcal{D}/2 \rfloor$, let r be $\mathcal{R} - \lfloor \mathcal{D}/2 \rfloor \geq 0$ to have $\mathcal{N} = \mathcal{Q}\mathcal{D} + r$, $\mathcal{Q} = \lfloor \mathcal{N}/\mathcal{D} \rfloor$ and $r + \lfloor \mathcal{D}/2 \rfloor = \mathcal{R} < \mathcal{D}$, and thus $\mathcal{Q} = \lfloor \mathcal{N}/\mathcal{D} \rfloor = round(\mathcal{N}/\mathcal{D})$. Otherwise, let r be $\mathcal{R} - \lfloor \mathcal{D}/2 \rfloor + \mathcal{D}$, being $r \geq 0$ but $r < \mathcal{D}$, to have $\mathcal{N} = (\mathcal{Q} - 1)\mathcal{D} + r$, $\mathcal{Q} - 1 = \lfloor \mathcal{N}/\mathcal{D} \rfloor$ and $r + \lfloor \mathcal{D}/2 \rfloor = \mathcal{R} + \mathcal{D} \geq \mathcal{D}$, and thus $\mathcal{Q} = \lfloor \mathcal{N}/\mathcal{D} \rfloor + 1 = round(\mathcal{N}/\mathcal{D})$. Note that the macro gives 0 if $\mathcal{D} = 0$ without division.

```
#define Round(NUM,DEN) (DEN ? (NUM+(DEN>>1))/DEN : 0)
```

4.3.41 `update_stats()`

`update_stats()` The function `update_stats()`, called only from `oh1_show_stats()`, updates the statistics data in `Stats.total` or `Stats.subtotal` specified by the argument `stotal` with the measured values of the current iteration recorded in `Stats.curr`. The update of the array element `time`$[k]$ for an event of key k is done only when the flag for the event `Stats.curr.time.ev`$[k] = 1$ indicating the event has occurred in the current step. The flag is then cleared if `stotal` = `Stats.total` meaning that the update for the event has completed for both `Stats.subtotal` and `Stats.total`.

On the other hand, the particle transfer statistics in `part[]` is not updated if `step` ≤ 0 meaning, e.g., `oh1_show_stats()` is called outside simulation loop. Otherwise, first we

reduce the local statistics in all nodes by `MPI_Reduce()` which calls `stats_reduce_part()` through `Op_StatsPart`, if this function is invoked as the first call in `oh1_show_stats()`, i.e., `statsMode` $= 1$ meaning this function is called only once, or `stotal` $\neq$ `Stats.total` meaning the first call with `Stats.subtotal`. Then, if the local node rank is 0, we calculate the average number of particle transfer among send/receive pairs and among nodes for keys updating `Stats.curr.part`$[k] = \sigma_p[k]$ for $k = k_p =$ `STATS_PART_MOVE_`x`_AVE` and $k = k_t =$ `STATS_PART_PG_`x`_AVE` where $x \in \{$`PRI`, `SEC`$\}$, referring to themselves and using `Round()`. That is, since the reduction results in that the total number of transferred particles in $\sigma_p[k_t]$ and the number of send/receive pairs in $\sigma_p[k_p]$, we update them as $\sigma_p[k_p] \leftarrow \sigma_p[k_t]/\sigma_p[k_p]$ and $\sigma_p[k_t] \leftarrow \sigma_p[k_t]/N$. Then we update the first half of `part[]` for primary particles which is always significant, while do the second half for secondaries when we are now in secondary mode, i.e. (`currmode` mod 2) $= 1$. Note that σ_p[`STATS_PART_`y] ($y \in \{$`PRIMARY`, `SECONDARY`$\}$) have the code of the previous mode rather than the number of transferred particles. Thus we simply increment `min`, `max` or `total` element of their counterparts in `part[]` according to the code being 1, 2 or 3 to sum up the number of each occasion.

```
static void
update_stats(struct S_statstotal *stotal, int step, int currmode) {
  int i, j, k, ev, nn=nOfNodes;
  int evclr = stotal==&Stats.total, reduce = statsMode==1 || !evclr;
  struct S_statstime *st = stotal->time;
  struct S_statspart *sp = stotal->part;
  int pm=Mode_PS(currmode)-1;
  int transkey=pm?STATS_PART_PRIMARY:STATS_PART_SECONDARY;
  dint trans=Stats.curr.part[transkey];

  for (i=0; i<STATS_TIMINGS<<1; i++) {
    if ((ev=Stats.curr.time.ev[i])) {
      double t = Stats.curr.time.val[i];
      st[i].ev++;
      if (t<st[i].min) st[i].min = t;
      if (t>st[i].max) st[i].max = t;
      st[i].total += t;
      if (evclr) Stats.curr.time.ev[i] = 0;
    }
  }
  if (step<=0) return;
  if (myRank!=0) {
    if (reduce)
      MPI_Reduce(st, NULL, STATS_PART_PRIMARY, MPI_LONG_LONG_INT, Op_StatsPart,
                 0, MCW);
    return;
  }
  if (reduce)
    MPI_Reduce(MPI_IN_PLACE, st, STATS_PART_PRIMARY, MPI_LONG_LONG_INT,
               Op_StatsPart, 0, MCW);
  for (i=0,j=0; i<(pm?1:2); i++,j+=STATS_PART_MOVE_SEC_MIN) {
    dint *scp=Stats.curr.part+j;
    struct S_statspart *spps=sp+j;
    if (reduce) {
      scp[STATS_PART_MOVE_PRI_AVE] = Round(scp[STATS_PART_PG_PRI_AVE],
                                           scp[STATS_PART_MOVE_PRI_AVE]);
```

```
      scp[STATS_PART_PG_PRI_AVE] = Round(scp[STATS_PART_PG_PRI_AVE], nn);
    }
    for (k=0; k<STATS_PART_MOVE_SEC_MIN; k++) {
      dint n = scp[k];
      if (n<spps[k].min) spps[k].min = n;
      if (n>spps[k].max) spps[k].max = n;
      spps[k].total += n;
    }
  }
  if      (trans==1) sp[transkey].min++;
  else if (trans==2) sp[transkey].max++;
  else if (trans==3) sp[transkey].total++;
}
```

4.3.42 Macro Stats_Reduce_Part_{Min,Max,Sum}()

Stats_Reduce_Part_Min()
Stats_Reduce_Part_Max()
Stats_Reduce_Part_Sum()

The macros Stats_Reduce_Part_x ($x \in \{\texttt{Min}, \texttt{Max}, \texttt{Sum}\}$), used only in stats_reduce_part(), peforms pairwise operations for reductions to obtain minimum, maximum and sum over the in/out array io[k] and input array in[k] to update io[k], where $k =$ KEY being the macro argument.

```
#define Stats_Reduce_Part_Min(KEY) { if (io[KEY]<in[KEY]) io[KEY] = in[KEY]; }
#define Stats_Reduce_Part_Max(KEY) { if (io[KEY]>in[KEY]) io[KEY] = in[KEY]; }
#define Stats_Reduce_Part_Sum(KEY) { io[KEY] += in[KEY]; }
```

4.3.43 stats_reduce_part()

stats_reduce_part()

The function stats_reduce_part(), called from MPI_Reduce() in update_stats() through Op_StatsPart, performs pairwise reduction of the local statistics of particle transfer stored in Stats.curr.part[] and given through its argument inarg and ioarg to have the minimum of the elemnts at STATS_PART_x_z_MIN, the maximum of those at STATS_PART_x_z_MAX, and the sum of those at STATS_PART_y_z_AVE, where $x \in \{\texttt{MOVE}, \texttt{GET}, \texttt{PUT}\}$, $y \in \{\texttt{MOVE}, \texttt{PG}\}$ and $z \in \{\texttt{PRI}, \texttt{SEC}\}$. The pairwise reduction is performed by the macros Stats_Reduce_Part_{Min,Max,Sum}().

```
static void
stats_reduce_part(void* inarg, void* ioarg, int* len, MPI_Datatype* type) {
  dint *in=(dint*)inarg, *io=(dint*)ioarg;
  int ps, statsbase=0;

  for (ps=0; ps<2; ps++,statsbase+=STATS_PART_MOVE_SEC_MIN) {
    Stats_Reduce_Part_Min(statsbase+STATS_PART_MOVE_PRI_MIN);
    Stats_Reduce_Part_Max(statsbase+STATS_PART_MOVE_PRI_MAX);
    Stats_Reduce_Part_Sum(statsbase+STATS_PART_MOVE_PRI_AVE);
    Stats_Reduce_Part_Min(statsbase+STATS_PART_GET_PRI_MIN);
    Stats_Reduce_Part_Max(statsbase+STATS_PART_GET_PRI_MAX);
    Stats_Reduce_Part_Min(statsbase+STATS_PART_PUT_PRI_MIN);
    Stats_Reduce_Part_Max(statsbase+STATS_PART_PUT_PRI_MAX);
    Stats_Reduce_Part_Sum(statsbase+STATS_PART_PG_PRI_AVE);
  }
}
```

4.3.44 print_stats()

print_stats() The function print_stats(), called from oh1_show_stats() and oh1_print_stats[_](), at first reduce timing statistics in the argument stotal which is Stats.total or Stats.subtotal for the nstep iteration steps to have the global minimum, maximum and total by MPI_Reduce() of T_StatsTime data, which calls stats_reduce_time() through Op_StatsTime.

Then, if the local node rank is 0, it prints the statistics in stotal, with the current step number is given by the argument step if stotal = Stats.subtotal. The statistics data stored in the element array time[] or part[] is judged valid if the leaf elements min and max satisfies min $\leq$ max. To print the meaning of each statistics data, we refer to StatsTimeStrings[k] for time[k] and StatsPartStrings[k] for part[k]. The average number of the leaf element part[].total is calculated by Round().

```
static void
print_stats(struct S_statstotal *stotal, int step, int nstep) {
  int i;
  struct S_statstime *st = stotal->time;
  struct S_statspart *sp = stotal->part;

  if (myRank!=0) {
    MPI_Reduce(st, NULL, STATS_TIMINGS<<1, T_StatsTime, Op_StatsTime, 0, MCW);
    return;
  }
  MPI_Reduce(MPI_IN_PLACE, st, STATS_TIMINGS<<1, T_StatsTime, Op_StatsTime,
             0, MCW);
  printf("\n");
  if (stotal==&Stats.subtotal)
    printf("# Subtotal Statistics for %d..%d\n",  step-nstep+1, step);
  else
    printf("# Total Statistics\n");
  printf("## Execution Times (sec)\n");
  for (i=0; i<STATS_TIMINGS<<1; i++) {
    if (st[i].ev==0)
      printf("  %-29s = -------- / -------- / -------- / ------------\n",
             StatsTimeStrings[i]);
    else
      printf("  %-29s = %8.3f / %8.3f / %8.3f / %12.3f\n",
             StatsTimeStrings[i],
             st[i].min, st[i].max, st[i].total/st[i].ev, st[i].total);
  }
  printf("## Particle Movements\n");
  for (i=0; i<STATS_PARTS; i++) {
    if (i<STATS_PART_PRIMARY && sp[i].min>sp[i].max)
      printf("  %-29s = -------- / -------- / -------- / ------------\n",
             StatsPartStrings[i]);
    else if (i<STATS_PART_PRIMARY)
      printf("  %-29s = %8lld / %8lld / %8lld / %12lld\n",
             StatsPartStrings[i],
             sp[i].min, sp[i].max, Round(sp[i].total,nstep), sp[i].total);
    else
      printf("  %-29s = %8lld / %8lld / %8lld / %12lld\n",
             StatsPartStrings[i],
```

```
                sp[i].min, sp[i].max, sp[i].total,
                sp[i].min+sp[i].max+sp[i].total);
    }
  }
```

4.3.45 stats_reduce_time()

stats_reduce_time()

The function stats_reduce_time(), called from MPI_Reduce() in print_stats() through Op_StatsTime, performs pairwise reduction of the local timing statistics stored in Stats.total[] or Stats.subtotal[] and given through the arguments inarg and ioarg. The reduction is performed on the leaf elements to obtain the minimum of min, the maximum of max and the total of total, if the element ev > 0.

```
static void
stats_reduce_time(void* inarg, void* ioarg, int* len, MPI_Datatype* type) {
  struct S_statstime *in=(struct S_statstime*)inarg;
  struct S_statstime *io=(struct S_statstime*)ioarg;
  int n=*len, i;

  for (i=0; i<n; i++) {
    if (in[i].ev>0) {
      io[i].ev += in[i].ev;
      if (in[i].min<io[i].min) io[i].min = in[i].min;
      if (in[i].max>io[i].max) io[i].max = in[i].max;
      io[i].total += in[i].total;
    }
  }
}
```

4.3.46 oh1_print_stats()

oh1_print_stats_()
oh1_print_stats()

The API functions oh1_print_stats_() for Fortran and oh1_print_stats() for C simply call print_stats() giving it Stats.total and the argment nstep to print statistics in Stats.total over the whole simulation steps nstep, if statsMode $\neq 0$. The second argument step of print_stats() can be anything and thus is set to 0.

```
void
oh1_print_stats_(int *nstep) {
  if (statsMode)  print_stats(&Stats.total, 0, *nstep);
}
void
oh1_print_stats(int nstep) {
  if (statsMode)  print_stats(&Stats.total, 0, nstep);
}
```

4.3.47 oh1_verbose()

oh1_verbose_()
oh1_verbose()

The API functions oh1_verbose_() for Fortran and oh1_verbose() for C simply invoke macro Verbose() giving it vprint() with the argument message as its second argument for verbose messaging. The first argument of Verbose() is set to 1 to indicate fundamental messaging.

```
void
oh1_verbose_(char *message) {
  Verbose(1, vprint(message));
}
void
oh1_verbose(char *message) {
  Verbose(1, vprint(message));
}
```

4.3.48 Macros `Vprint()` and `Vprint_Norank()`

`Vprint()` `Vprint_Norank()`

The macro `Vprint()`, used in `vprint()` and `dprint()`, constructs a message header including the local node's rank `myRank` into a buffer calling `sprintf()` with the argmument `RANKFORMAT`, and then concatenates it with the argument `FORMAT` and an end-of-line character by `strcat()` to pass it to `vprintf()` as the format argument. The remaining variable number arguments of the caller function, which is handled in this macro with `va_start()` and `va_end()`, are also passed to `vprintf()`. The concatenation of the format argument of `vprintf()` aims at minimizing the possibility of interleaving of the messages from multiple nodes. The macro also calls `fflush()` to flush the message.

The relative macro `VPrint_Norank()`, used solely in `vprint()`, works almost in same manner as `Vprint()` but `myRank` is not passed to `sprintf()` because its `RANKFORMAT` does not have the specifier for it.

```
#define Vprint(FORMAT, RANKFORMAT) {\
  char buf[1024];\
  va_list v;\
  sprintf(buf, RANKFORMAT, myRank);\
  strcat(buf, FORMAT);\
  strcat(buf, "\n");\
  va_start(v, FORMAT);\
  vprintf(buf, v);\
  fflush(stdout);\
  va_end(v);\
}
#define Vprint_Norank(FORMAT, RANKFORMAT) {\
  char buf[1024];\
  va_list v;\
  sprintf(buf, RANKFORMAT);\
  strcat(buf, FORMAT);\
  strcat(buf, "\n");\
  va_start(v, FORMAT);\
  vprintf(buf, v);\
  fflush(stdout);\
  va_end(v);\
}
```

4.3.49 `vprint()`

`vprint()`

The function `vprint()`, given to the macro `Verbose()` as its argument and called in it, prints verbose message specified by the variable number arguments following `format` by

the macro `Vprint()` or `Vprint_Norank()`. If `verboseMode` ≥ 3 to specify the most verbose execution with message printing from all nodes, the rank of the local node is also printed by `Vprint()`, while `Vprint_Norank()` is used otherwise.

```
void
vprint(char* format, ...) {

  if (verboseMode>=3) { Vprint(format, "*Starting[%d] "); }
  else                { Vprint_Norank(format, "*Starting "); }
}
```

4.3.50 `dprint()`

`dprint()` The function `dprint()`, defined only for debugging and thus not used in the production version of the library, prints a debug message specified by its variable number arguments following `format` by the macro `Vprint()`. The message always contains the rank of the local node.

```
void
dprint(char* format, ...) {

  if (nOfNodes>=1000)     { Vprint(format, "#Debug[%04d] "); }
  else if (nOfNodes>=100) { Vprint(format, "#Debug[%03d] "); }
  else if (nOfNodes>=10)  { Vprint(format, "#Debug[%02d] "); }
  else                    { Vprint(format, "#Debug[%d] "); }
}
```

4.4 Header File ohhelp2.h

The header file of level-2 library, ohhelp2.h, `#include`'s a C header file oh_part.h to declare a `struct`ured data type `S_particle` to represent a particle, declares global variables used in level-2 library codes to manipulate particles and buffers of them, and gives prototypes of API functions and those called by higher level codes.

4.4.1 Header File Inclusion

At very first, ohhelp2.h `#include`'s a header file named oh_part.h which defines the `struct` for a particle, `S_particle`. As discussed in §3.5.1, it has the following elements in default.

- `x`, `y` and `z` have the double-float three-dimensional coordinate at which the particle is located.
- `vx`, `vy` and `vz` have the double-float velocity elements of the particle.
- `pid` is the unique 64-bit integer identifier given to the particle.
- `nid` is the identifier of the subdomain in which the particle is residing, or will be resinding in the next simulation step.
- `spec` is the integer in $[0, S-1]$ in C codes or in $[1, S]$ in Fortran codes, to identify the species of the particle.

```
#include "oh_part.h"
```

4.4.2 Particle Buffers and Related Variables

Next we declare the following variables for particles and their transfer, using the trick of `EXTERN` to have their home in ohhelp2.c.

`nOfLocalPLimit`

- The integer variable `nOfLocalPLimit` have the absolute maximum number of particles which a local node can accommodate. Its value P_{lim} should be defined by the simulator body possibly by calling `oh2_max_local_particles()` and then should be given to `oh2_init()` through its argument `maxlocalp`. The value defines the size of the particle buffer `Particles[]` if it is not allocated by the simulator body, and particle send buffer `SendBuf[]`. The variable is referred to by `oh2_inject_particle()` to check if `Particles[]` has room to inject a particle, and by `oh2_remap_injected_particle()` and `oh2_remove_injected_particle()` to check if the argument particle is in the region for injected particles.

`Particles`

- Each element of the array `Particles`$[P_{lim}]$ has the `S_particle` structure of a paritlce accommodated by the local node. The array must be allocated by a Fortran-coded simulator body and the pointer to it must be given to `oh2_init()` through its argument `pbuf`. On the other hand, a C-coded body may do so by giving the double pointer to it through `pbuf`, or may give a pointer to `NULL` through `pbuf` to let `oh2_init()` allocate the array body. The buffer is partitioned into two parts, one for primary particles and the other for secondary ones. Then each part is further decomposed for species to have $2S$ blocks $pbuf(0,0)$, ..., $pbuf(0,S-1)$, $pbuf(1,0)$, ..., $pbuf(1,S-1)$ in this order and the size of $pbuf(p,s)$ is `TotalP`$[p][s]$.

  The array is directly referred to by the following functions;

move_to_sendbuf_secondary(), move_to_sendbuf_uw(), move_to_sendbuf_dw(), move_injected_to_sendbuf(), oh2_inject_particle(), oh2_remap_injected_particle() and oh2_remove_injected_particle();

and indirectly through RecvBufBases[][] by the following functions;

try_primary2(), exchange_particles(), move_injected_from_sendbuf() and receive_particles().

SendBuf

- The array SendBuf[P_{lim}] of S_particle structures is used to send particles from the local node to other nodes. The buffer is partitioned for species and then for receiver nodes to have SN blocks $sbuf(0,0)$, ..., $sbuf(0,N-1)$, ..., $sbuf(S-1,0)$, ..., $sbuf(S-1,N-1)$ in this order and the displacement of the head of $sbuf(s,n)$ from the head of SendBuf[] is SendBufDisps[s][n]. Since a node may send out all the particles in Particles[] and receives the same amount of particles from other nodes, we need to have SendBuf[] of the size P_{lim} same as Particles[]. However, the number of sending particles is significantly smaller than P_{lim} in usual cases, we could limit the size of SendBuf to a small fraction, say 10 %, of Particles[] if we can devise an *in-place* all-to-all communication required in, for example, initial particle distribution.

  The array is allocated by init2() or its counterpart in level-4p or higher library, and referred to by try_primary2(), exchange_particles(), move_to_sendbuf_uw(), move_to_sendbuf_dw(), move_injected_to_sendbuf(), move_injected_from_sendbuf(), receive_particles() and send_particles().

RecvBufBases

- The element [p][s] of the array RecvBufBases[2][S][37] is the S_particle type pointer to a block $rbuf(p,s)$ in Particles[] to which the local node receives primary ($p=0$) or secondary ($p=1$) particles of species s from other nodes. That is, the block is the receive buffer for particle transfer. The array is allocated by init2() and then its elements are initialized by move_to_sendbuf_uw() through move_to_sendbuf_primary() and move_to_sendbuf_secondary(). Then the elements are referred to by try_primary2(), exchange_particles(), and receive_particles() and are updated by them so that the elements point the receive buffer for each sender node. Similarly, move_injected_from_sendbuf() refers to and update the elements so that they reflects particle injection into the primary subdomain of the local node.

secondaryBase
totalLocalParticles

- The integer pointer secondaryBase and totalLocalParticles point the shadow variables of primaryParts and totalParts which are located in the argument array of oh2_init() namely pbase[1] and pbase[2]. That is, the shadow variable pbase[1] has the number of primary particles Q^n_n and thus the displacement of the base of $pbuf(1,0)$, the first block of secondary particles, from the head of Particles[]. On the other hand, the pbase[2] has the total number of particles Q_n and thus the displacement of the first unused entry of Particles[]. The poiters and pointed variables are initialized by init2(), and *secondaryBase is updated by try_primary2() and move_to_sendbuf_secondary() together with its substance primaryParts, while *totalLocalParticles is updated by transbound2().

SendBufDisps

- The element [s][m] of the integer array SendBufDisps[S][N] has the displacement of the block $sbuf(s,m)$ in SendBuf[] from its head. That is, prior to sending the particles of species s to the node m is, they are at first moved from Particles[] into the

[37] It has one extra element [2][0] for sort_received_particles().

block starting from SendBuf[SendBufDisps[s][m]]. The array is allocated by init2() and its elements are initialized by set_sendbuf_disps(). Then move_to_sendbuf_uw(), move_to_sendbuf_dw() and move_injected_to_sendbuf() increment the elements each time they move a particle from Particles[] to SendBuf[]. Finally, after reinitialized by set_sendbuf_disps(), the elements are referred to by try_primary2(), exchange_particles(), move_injected_from_sendbuf(), receive_particles() and send_particles() for particle sending and injection.

RecvBufDisps

- The integer array RecvBufDisps[N] has displacements of receive buffer blocks in Particles[] for particle transfer with anywhere accommodation. That is, the local node receives primary ($p = 0$) or secondary ($p = 1$) particles of species s from the node m into the block starting from RecvBufBases[p][s][RecvBufDisps[m]] by MPI_Alltoallv(). The array is allocated by init2() and its elements are set and referred to by try_primary2() and exchange_particles().

nOfInjections

- The integer varialbe nOfInjections $= Q_n^{\rm inj}$ has the number of particles injected by the local node n using oh2_inject_particle(). That is, after zero-cleared by init2() and transbound2(), it is incremented by oh2_inject_particle() to have the number of injected particles at the call of transbound2(). Then it is referred to by move_to_sendbuf_primary(), move_to_sendbuf_secondary() and move_injected_to_sendbuf() to send injected particles to other nodes or to keep primary ones in the local node, and by oh2_remap_injected_particle() and oh2_remove_injected_particle() to check if the argument particle is in the region for injected particles.

specBase

- The integer variable specBase has 0 if the library is called from C-coded simulator body through oh2_init() or oh3_init(), or has 1 if called from Fortran counterpart through oh2_init_() or oh3_init_(), to represent the identification number of the first species. Then it is referred to by move_injected_to_sendbuf(), oh2_inject_particle(), oh2_remap_injected_particle() and oh2_remove_injected_particle() to translate Fortran's 1-base representation in spec element of S_particle particle data into 0-base one used in the library.

T_Particle

- The MPI_Datatype variable T_Particle has the MPI data-type of a S_particle particle data for particle transfer. The value of this variable is created by MPI_Type_contiguous() called in init2(), and used for MPI communication in try_primary2(), exchange_particles(), receive_particles() and send_particles().

Requests
Statuses

- The array of MPI_Request data Requests[$4NS$] is to keep the requests of asynchronous communications MPI_Isend() and MPI_Irecv() performed in receive_particles() and send_particles(). Then the results of these requests are stored in the array Statuses[$4NS$] of MPI_Status type data by MPI_Waitall() in exchange_particles(). These arrays are allocated by init2() to have $2 \times 2 \times S \times N$ elements for sending/receiveing (2) primary and secondary particles (2) of S species to N nodes.

```
EXTERN int nOfLocalPLimit;
EXTERN struct S_particle *Particles;    /* [nOfLocalPLimit] */
EXTERN struct S_particle *SendBuf;      /* [nOfLocalPLimit] */
EXTERN struct S_particle **RecvBufBases;/* [2][nOfSpecies] */
EXTERN int *secondaryBase, *totalLocalParticles;
EXTERN int *SendBufDisps;               /* [nOfSpecies][nOfNodes] */
```

```
EXTERN int *RecvBufDisps;               /* [nOfNodes] */
EXTERN int nOfInjections;
EXTERN int specBase;
EXTERN MPI_Datatype T_Particle;
EXTERN MPI_Request *Requests;           /* [nOfNodes*nOfSpecies*2*2] */
EXTERN MPI_Status *Statuses;            /* [nOfNodes*nOfSpecies*2*2] */
```

4.4.3 Macro `Particle_Spec()`

`Particle_Spec()` The macro `Particle_Spec()`, used in `move_injected_to_sendbuf()`, `oh2_inject_particle()`, `oh2_remap_injected_particle()` and `oh2_remove_injected_particle()` and higher level library functions such as those in level-4p, is replaced with given argument expression `S` if `OH_HAS_SPEC` is defined, or with 0 otherwise. That is, if `S_particle` structure has the element `spec` as asserted by the fact that `OH_HAS_SPEC` is defined, the expression `S` having the reference to the element is used to have the species of the injected particle. Otherwise, it is assured that $S = 1$ and thus 0 is given for the species unconditionally.

```
#ifdef OH_HAS_SPEC
#define Particle_Spec(S) (S)
#else
#define Particle_Spec(S) (0)
#endif
```

4.4.4 Macros `Decl_Grid_Info()`, `Subdomain_Id()` and `Primarize_id()`

Fundamentally, a lower level library is designed to work independently of the use of higher level ones. However a small fraction of level-2 functions have to change its behavior with/without position-aware particle management, i.e., depending on whether `OH_POS_AWARE` is defined, because `nid` element of `S_particle` structure has level-dependent meaning. Therefore, we define three macros `Decl_Grid_Info()`, `Subdomain_Id()` and `Primarize_Id()` to cope with the level-dependency to let them act as follows if `OH_POS_AWARE`.

`Decl_Grid_Info()` The macro `Decl_Grid_Info()` is to declare an `OH_nid_t` variable `nidelement` and integer variables named `subdomid`, `gridmask` and `loggrid`. The variable `gridmask` is used only in level-4p and/or higher level libraries, while other three are used to extract subdomain identifier from `nid` of `S_particle` structured data for particles referring to a global variable `AbsNeighbors`$[2][3^D]$, if the functions `move_to_sendbuf_uw()`, `move_to_sendbuf_dw()` and `move_injected_to_sendbuf()` are used in a higher-level library for position-aware particle management with `OH_POS_AWARE` defined. The variables `gridmask` and `loggrid` are for *caching* `gridMask` and `logGrid` having the mask and bit-width of lower bits for particle grid-position and thus `loggrid` has the right-shift count to eliminate it and to extract *subdomain code* by the macro `Subdomain_Id()`.

`Subdomain_Id()` The macro `Subdomain_Id`(i, p) acts on the *subdomain code* of the `nid` element i of a primary ($p = 0$) or secondary ($p = 1$) particle, i.e., $\sigma = \lfloor i/2^{\Gamma} \rfloor$ where $\Gamma =$ `loggrid`, to give the subdomain m in which the particle will reside, as its expansion result if $i \geq 0$. The subdomain code σ for a subdomain m is $k \in [0, 3^D)$ if m is the k-th neighbor of the local node's primary ($p = 0$) or secondary ($p = 1$) subdomain definitely, or $m + 3^D$ otherwise[38].

[38] σ can be $m + 3^D$ for a neighbor subdomain m.

On the other hand if $i < 0$, the macro is replaced with -1 so that functions using the macro find eliminated particles by examining the result of the macro rather than the `nid` element, as we can do so without position-aware partilce management. Therefore, the macro is replaced with m as;

$$m = \begin{cases} -1 & i < 0 \\ \texttt{AbsNeighbors}[p][\sigma] & i \geq 0 \ \wedge \ \sigma < 3^D \\ \sigma - 3^D & i \geq 0 \ \wedge \ \sigma \geq 3^D \end{cases}$$

where `AbsNeighbors`$[p][k]$ is the subdomain identifier of k-th neighbor of the local node's primary ($p = 0$) or secondary ($p = 1$) subdomain, i.e., always-positive version of `Neighbors`$[p][k]$ without aware of multiple occurrences of subdomains in a neighbor set. Note that the local variables `nidelement` and `subdomid` declared by `Decl_Grid_Info()` are used in this macro to have i and σ temporarily in it.

`Primarize_Id()` The subdomain code of a particle can be $(N + 3^D) + \sigma$ if the particle is injected into a subdomain represented by σ as a secondary particle of the local node. The macro `Primarize_Id`(π, m), used only in `move_injected_to_sendbuf()` in the level-2 library, acts on the subdomain code $\sigma' = (N + 3^D) + \sigma$ of the particle pointed by π to let it $\sigma = \sigma' - (N + 3^D)$. It also let m be the subdomain identifier encoded in σ by `Subdomain_Id()` with $p = 1$ because the particle is a secondary one. Note that $(N + 3^D)$ is cast to `OH_nid_t` prior to the subtraction to have σ because the `nid` can be `long␣long␣int`.

If `OH_POS_AWARE` is not defined, on the other hand, `Decl_Grid_Info()` just declares a dummy variable `unusedvariable` which should not been referred to in the functions using the macro[39], while `Subdomain_Id()` is simply replaced with its first argument and `Primarize_Id()` is not defined[40].

```
#ifdef OH_POS_AWARE
EXTERN int gridMask, logGrid;
EXTERN int AbsNeighbors[2][OH_NEIGHBORS];
#define Decl_Grid_Info() \
  OH_nid_t nidelement;  int subdomid;\
  const int gridmask=gridMask, loggrid=logGrid
#define Subdomain_Id(ID, PS) \
  ((nidelement=(ID))<0 ? -1 :\
      ((subdomid=nidelement>>loggrid)<OH_NEIGHBORS ?\
          AbsNeighbors[PS][subdomid] : subdomid-OH_NEIGHBORS))
#define Primarize_Id(P, SD) {\
  const OH_nid_t nidelem =\
    ((P)->nid -= (OH_nid_t)(nOfNodes+OH_NEIGHBORS)<<loggrid);\
  SD = Subdomain_Id(nidelem, 1);\
}
#else
#define Decl_Grid_Info() int unusedvariable
#define Subdomain_Id(ID, PS) (ID)
#endif
```

[39] If we make the expansion result of the macro empty, the macro cannot be followed by any variable declarations with C89.

[40] `Primarize_Id()` is not used if `OH_POS_AWARE` is undefined and thus we leave it undefined too.

4.4.5 Function Prototypes

The next and last block is to declare function prototypes. First we declare the prototypes of the API function pairs each of which consists of API for Fortran and C, as listed below.

- The function `oh2_init[_]()` initializes data strucutures of the level-2 library.
- The function `oh2_transbound[_]()` at first performs what its level-1 counterpart `oh1_transbound[_]()` does to have particle transfer schedule, and then transfers particles from/to the particle buffer `Particles[]`.
- The function `oh2_max_local_particles[_]()` calculates P_{lim}, the size of the particle buffer `Particles[]`.
- The function `oh2_inject_particle[_]()` injects a particle and place it at the bottom of `Particles[]`.
- The function `oh2_remap_injected_particle[_]()` maintains `NOfPLocal[][][]` and `InjectedParticles[][][]` of an injected particle.
- The function `oh2_remove_injected_particle[_]()` removes an injected particle maintaining `NOfPLocal[][][]` and `InjectedParticles[][][]`.
- The function `oh2_set_total_particles()` is to initialize `TotalP`, `primaryParts` and `totalParts` with `NOfPLocal` after having initial particle setting in `Particles` and `NOfPLocal` but before injecting/removing particles into/from them prior to the first call of `oh2_transbound()`.

As done in §4.2.11, prior to showing the function prototypes, we show the second part of the header files ohhelp_c.h for C-coded simulators and ohhelp_f.h for Fortran-coded ones, which define the aliases of level-2 API functions[41]. First, they `#define` the aliases of level-2 API function which does not have higher level counterpart, in the `#else` part of `#if␣OH_LIB_LEVEL=1`.

```
#else
#define oh_set_total_particles() oh2_set_total_particles()
```

Then ohhelp_c.h gives the prototypes of the function above, which is also given in ohhelp2.h, after it `#include`'s the header file oh_part.h to define `S_particle`.

```
#include "oh_part.h"
void oh2_set_total_particles();
```

Next, they `#define` the aliases of level-2 API functions which have level-4p counterparts;

```
#if OH_LIB_LEVEL!=4
#define oh_max_local_particles(A1,A2,A3) oh2_max_local_particles(A1,A2,A3)
#define oh_inject_particle(A1)           oh2_inject_particle(A1)
#define oh_remap_injected_particle(A1)   oh2_remap_injected_particle(A1)
#define oh_remove_injected_particle(A1)  oh2_remove_injected_particle(A1)
```

[41] Aliases of `oh2_set_total_particles()` in ohhelp_c.h and ohhelp_f.h are silightly different, i.e., the former has "`()`" in both of the macro and definition while latter does not have it.

and function prototypes are given to C headers[42]. Note that the prototypes of `oh2_max_local_particles()`, `oh2_remap_injected_particle()` and `oh2_remove_injected_particle()` are not given if the level-4p extension is in effect because the functions are useless (and harmful), but that of `oh2_inject_particle()` is given regardless of the extension because it is may be called for injections followed by remapping.

```
int  oh2_max_local_particles(long long int npmax, int maxfrac, int minmargin);
void oh2_remap_injected_particle(struct S_particle *part);
void oh2_remove_injected_particle(struct S_particle *part);
#endif
void oh2_inject_particle(struct S_particle *part);
```

Then both headers `#define` the aliases level-2 specific API functions if `OH_LIB_LEVEL` is 2.

```
#if OH_LIB_LEVEL==2
#define oh_init(A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14) \
        oh2_init(A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14)
#define oh_transbound(A1,A2) oh2_transbound(A1,A2)
```

Finally, the prototypes of these functions are given in ohhelp_c.h and ohhelp1.h.

```
void oh2_init(int **sdid, int nspec, int maxfrac, int **nphgram,
              int **totalp, struct S_particle **pbuf, int **pbase,
              int maxlocalp, void *mycomm, int **nbor,
              int *pcoord, int stats, int repiter, int verbose);
int  oh2_transbound(int currmode, int stats);
```

On the other hand, the prototypes of Fortran API functions are solely given in ohhelp2.h, while their Fortran versions are given in oh_mod2.F90 as shown in §3.5.

```
void oh2_set_total_particles_();
int  oh2_max_local_particles_(dint *npmax, int *maxfrac, int *minmargin);
void oh2_inject_particle_(struct S_particle *part);
void oh2_remap_injected_particle_(struct S_particle *part);
void oh2_remove_injected_particle_(struct S_particle *part);
void oh2_init_(int *sdid, int *nspec, int *maxfrac, int *nphgram,
               int *totalp, struct S_particle *pbuf, int *pbase,
               int *maxlocalp, struct S_mycommf *mycomm, int *nbor,
               int *pcoord, int *stats, int *repiter, int *verbose);
int  oh2_transbound_(int *currmode, int *stats);
```

Next we declare the prototypes of the following functions used in the level-3 and/or higher level library.

- The function `init2()` is the body of `oh2_init()`.
- The function `transbound2()` is the body of `oh2_transbound()`.
- The function `exchange_primary_particles()` is the core of the particle transfer in primary mode.

[42] Prototypes of `oh2_max_local_particles()` in ohhelp_c.h and ohhelp2.h are slightly different, i.e., the type of its first argument is `long long int` in former, while in latter is `dint`.

- The functions `move_to_sendbuf_primary()` moves particles to be transferred from `Particles[]` to `SendBuf[]` and packs those remaining in `Particles[]` in primary mode.
- The function `set_sendbuf_disps()` calculates each entry of `SendBufDisps[][]`.
- The function `exchange_particles()` is the core of the particle transfer in secondary mode.

```
/* Prototypes for the functions called from higher-level library code */
void init2(int **sdid, int nspec, int maxfrac, int **nphgram,
           int **totalp, struct S_particle **pbuf, int **pbase, int maxlocalp,
           struct S_mycommc *mycommc, struct S_mycommf *mycommf,
           int **nbor, int *pcoord, int stats, int repiter, int verbose);
int  transbound2(int currmode, int stats, int level);
void exchange_primary_particles(int currmode, int stats);
void move_to_sendbuf_primary(int secondary, int stats);
void set_sendbuf_disps(int secondary, int parent);
void exchange_particles(struct S_commlist *secrlist, int secrlsize,
                        int oldparent, int neighboring, int currmode,
                        int stats);
```

4.5 C Source File ohhelp2.c

4.5.1 Header File Inclusion

The first job done in ohhelp2.c is the inclusion of the header files ohhelp1.h and ohhelp2.h. Before the inclusion of ohhelp1.h, we `#define` the macro `EXTERN` as `extern` so as to make variables declared in the file external, but after that we make it `#undef`'iend and then `#define` it as empty so as to provide variables declared in ohhelp2.h with their homes, as discussed in §4.2.3.

```
#define EXTERN extern
#include "ohhelp1.h"
#undef  EXTERN
#define EXTERN
#include "ohhelp2.h"
```

4.5.2 Function Prototypes

The next and last job to do prior to function definitions is to declare the prototypes of the following functions private for the level-2 library.

- The function `try_primary2()` performs the particle transfer in primary mode after calling its level-1 couterpart `try_primary1()` to check if we will be in primary mode in the next step.
- The function `try_stable2()` performs the particle transfer in secondary mode after calling its level-1 couterpart `try_stable1()` to check if we can keep the helpand-helper configuration.
- The function `rebalance2()` performs the particle transfer in secondary mode after calling its level-1 counterpart `rebalance1()` to establish a new helpand-helper configuration.
- The function `move_to_sendbuf_secondary()` moves particles to be transferred from `Particles[]` to `SendBuf[]` and packs those remaining in `Particles[]` in secondary mode.
- The functions `move_to_sendbuf_uw()` and `move_to_sendbuf_dw()` move particles to be transferred from `Particles[]` to `SendBuf[]` and packs those remaining in `Particles[]` for a block $pbuf(p, s)$ whose region is shifted upward and downward, i.e. to the direction of smaller/greater addresses, respectively.
- The function `move_injected_to_sendbuf()` moves injected particles from the bottom of `Particles[]` to `SendBuf[]`.
- The function `move_injected_from_sendbuf()` moves particles injected into the primary/secondary subdomain of the local node from `SendBuf[]` back to `Particles[]`.
- The function `receive_particles()` performs receive (and send in special cases) communication of particle transfer.
- The funciton `send_particles()` performs send communication of particle transfer.

```
/* Prototypes for private functions. */
static int   try_primary2(int currmode, int level, int stats);
static int   try_stable2(int currmode, int level, int stats);
static void  rebalance2(int currmode, int level, int stats);
static void  move_to_sendbuf_secondary(int secondary, int stats);
static void  move_to_sendbuf_uw(int ps, int me, int *putmes, int cbase,
                                int *ctp, int nbase, int *ntp,
                                struct S_particle **rbb);
static void  move_to_sendbuf_dw(int ps, int me, int *putmes, int ctail,
                                int *ctp, int ntail, int *ntp);
static void  move_injected_to_sendbuf();
static void  move_injected_from_sendbuf(int *injected, int mysd,
                                        struct S_particle **rbb);
static void  receive_particles(struct S_commlist *rlist, int rlsize, int *req);
static void  send_particles(struct S_commlist *slist, int slsize, int myregion,
                            int parentregion, int *req);
```

4.5.3 `oh2_init()` and `init2()`

`oh2_init_()` `oh2_init()`

The API functions `oh2_init_()` for Fortran and `oh2_init()` for C receive a set of array/structure variables through which level-1 and level-2 library functions communicate with the simulator body, and a few integer parameters to specify the behavior of the library. The functions have the following arguments.

- `sdid`
  `nspec`
  `maxfrac`
  `nphgram`
  `totalp`
  The five arguments above are perfectly equivalent to those of the level-1 countgerparts `oh1_init[_]()`.

- The argument `pbuf` should be the (double) pointer to an arrray of $[P_{lim}]$ of `S_particle` type particles which is referred to as `Particles[]` in the library functions. The array itself is allocated by `init2()` if `pbuf` points `NULL`.

- The argument `pbase` should be the (double) pointer to an array of three elements having displacements of the regions in `Particles[]`. The first element `pbase[0]` is the displacement of the head of the first part of `Particles[]` for primary particles and thus is always zero. The second element `pbase[1]` is the displacement of the head of its second part for secondary particles thus is equal to the number of primary particles Q_n^n for the local node n. The third element `pbase[]` is the displacement of the unused part of `Particles[]` and thus is equal to the total number of particles Q_n the local node n accommodates. Therefore, `pbase[1]` and `pbase[2]` is pointed by `secondaryBase` and `totalLocalParticles` so as to make these elements the shadows of `primaryParts` and `totalParts`.

- The argument `maxlocalp` should have the value of P_{lim} being the size of `Particles[]` and `SendBuf[]`.

- `mycomm`

nbor
pcoord
stats
repiter
verbose
The six arguments above are perfectly equivalent to those of the level-1 countgerparts oh1_init[_]().

Note that oh2_init[_]() does not have arguments rcounts and scounts which oh1_init[_]() has, because we do not need to report the transfer schedule.

The API functions almost simply call init2() passing all given arguments to it except for the followings.

- oh2_init_() passes the pointers to sdid, nphgram, totalp, pbuf, pbase and nbor rather than themselves.
- oh2_init_() passes mycomm to mycommf of init2() while NULL is passed through mycommc of init2() to keep it from allocation of MyCommC.
- oh2_init() passes mycomm to mycommc of init2() while NULL is passed through mycommf of init2() telling it that the body of MyCommF is not required. It also *casts* the argument as S_mycommc pointer type, because mycomm is declared as a void pointer to allow the simulator body to be completely unaware of the structure.
- Prior to calling init2(), oh2_init_() lets specBase be 1 to indicate the species in S_particle structures is represented by one-origin manner, while oh2_init() lets it be 0 to indicate zero-origin numbering.

```
void
oh2_init_(int *sdid, int *nspec, int *maxfrac, int *nphgram,
          int *totalp, struct S_particle *pbuf, int *pbase, int *maxlocalp,
          struct S_mycommf *mycomm, int *nbor, int *pcoord,
          int *stats, int *repiter, int *verbose) {
  specBase = 1;
  init2(&sdid, *nspec, *maxfrac, &nphgram, &totalp,
        &pbuf, &pbase, *maxlocalp, NULL, mycomm, &nbor, pcoord,
        *stats, *repiter, *verbose);
}
void
oh2_init(int **sdid, int nspec, int maxfrac, int **nphgram,
         int **totalp, struct S_particle **pbuf, int **pbase, int maxlocalp,
         void *mycomm, int **nbor, int *pcoord,
         int stats, int repiter, int verbose) {
  specBase = 0;
  init2(sdid, nspec, maxfrac, nphgram, totalp,
        pbuf, pbase, maxlocalp, (struct S_mycommc*)mycomm, NULL, nbor, pcoord,
        stats, repiter, verbose);
}
```

init2() The function init2() implements the initialization for its caller API functions, oh2_init_() and oh2_init(), or part of that for its higher level counterpart init3(). The arguments of this function are almost same as those of oh2_init() but its mycomm is split into two arguments mycommc and mycommf, which are NULL if called from oh2_init_() or oh2_init() respectively.

```
void
init2(int **sdid, int nspec, int maxfrac, int **nphgram,
      int **totalp, struct S_particle **pbuf, int **pbase, int maxlocalp,
      struct S_mycommc *mycommc, struct S_mycommf *mycommf,
      int **nbor, int *pcoord, int stats, int repiter, int verbose) {
  int ns, nn, nnns, s;
```

At first the function calls its level-1 counterpart `init1()` to initialize data structures specific to level-1 and common to level-2. The arguments passed to `init1()` are almost simply those given to `init2()` but `NULL` is passed to `rcounts` and `scounts` because level-2 library does not show the particle transfer schedule to the simulator body.

```
    init1(sdid, nspec, maxfrac, nphgram, totalp, NULL, NULL,
          mycommc, mycommf, nbor, pcoord, stats, repiter, verbose);
```

Then, after obtaining $S =$ `nOfSpecies` and $N =$ `nOfNodes`, we set `maxlocalp` into $P_{lim} =$ `nOfLocalPLimit` and allocate `Particles`$[P_{lim}]$ by `mem_alloc()` to set its base pointer into `*pbuf` if it was `NULL`, while the pointer to the array allocated by the simulator body is simply set into `Particles` otherwise. In addition, we initialize `totalParts` $= P_{lim}$ so that, prior to the first call of `oh2_transbound()` (or one of its higher-level counterparts) or `oh2_set_total_particles()`, any injections are judged causing particle buffer overflow, and level-4p functions such as `oh4p_map_particle_to_neighbor()` correctly judge their argument particles are not injected one.

```
    ns = nOfSpecies;  nn = nOfNodes;  nnns = nn * ns;

    nOfLocalPLimit = totalParts = maxlocalp;
    if (*pbuf)
      Particles = *pbuf;
    else
      Particles = *pbuf =
        (struct S_particle*)mem_alloc(sizeof(struct S_particle),
                                      maxlocalp, "Particles");
```

Next we define the MPI data-type for a `S_particle` structure, which is simply a `MPI_BYTE` sequence of `sizeof(struct␣S_particle)`, by `MPI_Type_contiguous()`[43], and commit its use by `MPI_Type_commit()`.

```
    MPI_Type_contiguous(sizeof(struct S_particle), MPI_BYTE, &T_Particle);
    MPI_Type_commit(&T_Particle);
```

We continue the conditional allocation of the interface array `*pbase` with `mem_alloc()`, and then clear all of its three elements with zero to assume that the local node has no particles at initial. We also let `secondaryBase` and `totalLocalParticles` point the elements `*pbase`[1] and `*pbase`[2] respecitively to let library functions know where these shadow variables are.

```
    if (!*pbase)  *pbase = (int*)mem_alloc(sizeof(int), 3, "ParticleBase");
```

[43] Because we ignore endian problem which could arise if an OhHelp'ed simulator were executed on a heterogeneous parallel system.

```
  (*pbase)[0] = (*pbase)[1] = (*pbase)[2] = 0;
  secondaryBase = *pbase + 1;  totalLocalParticles = *pbase + 2;
```

Finally we allocate the following library's own global variables by `mem_alloc()`; `SendBuf`$[P_{lim}]$ unless `OH_POS_AWARE` is defined to mean it is allocated by level-4p initializer `init4p()`, `RecvBufBases`$[2][S]$[44], `SendBufDisps`$[S][N]$, and `RecvBufDisps`$[N]$. We also allocate `Requests`$[4SN + 2 \cdot 3^D]$ and `Statuses`$[4SN + 2 \cdot 3^D]$. Note that the size is larger by $2 \cdot 3^D$ than $4SN$ discussed in §4.4.2, because that if we have position-aware particle management we could need have $4N + 2 \cdot 3^D$ entries for them as discussed in §4.9.4. We also initialize `nOfInjections` $= Q_n^{\text{inj}}$ clearling it with zero to indicate no particles are injected.

```
#ifndef OH_POS_AWARE
  SendBuf = (struct S_particle*)mem_alloc(sizeof(struct S_particle), maxlocalp,
                                          "SendBuf");
#endif
  RecvBufBases = (struct S_particle**)mem_alloc(sizeof(struct S_particle*),
                                                2*ns+1, "RecvBufBases");
  SendBufDisps = (int*)mem_alloc(sizeof(int),  nnns, "SendBufDisps");
  RecvBufDisps = (int*)mem_alloc(sizeof(int),  nn, "RecvBufDisps");
  nOfInjections = 0;

  Requests = (MPI_Request*)mem_alloc(sizeof(MPI_Request),
                                     nnns*4+OH_NEIGHBORS*2, "Requests");
  Statuses = (MPI_Status*) mem_alloc(sizeof(MPI_Status),
                                     nnns*4+OH_NEIGHBORS*2, "Statuses");
}
```

4.5.4 `oh2_transbound()` and `transbound2()`

`oh2_transbound_()` `oh2_transbound()`

The API functions `oh2_transbound_()` for Fortran and `oh2_transbound()` for C provide a simulator body calling them with the core mechanism of level-2 library. The meanings of their two arguments, `currmode` and `stats`, and return value in $\{-1, 0, 1\}$ are perfectly equivalent to those of the level-1 counterparts `oh1_transbound[_]()`. Also similarly to the counterparts, their bodies only have a simple call of `transbound2()` but the third argument `level` is 2 to indicate the function is called from level-2 API functions.

```
int
oh2_transbound_(int *currmode, int *stats) {
  return(transbound2(*currmode, *stats, 2));
}
int
oh2_transbound(int currmode, int stats) {
  return(transbound2(currmode, stats, 2));
}
```

`transbound2()`

The function `transbound2()`, called from `oh2_transbound_()`, `oh2_transbound()` or the level-3 counterpart `transbound3()`, first calls its level-1 counterpart `transbound1()` to calculate `NOfPrimaries`[][], `TotalPGlobal`[], `nOfParticles` and `nOfLocalPMax` from

[44] It has one extra element [2][0].

NOfPLocal[][][] of the local node and other nodes, and to have currmode which indicates not only the current execution mode but also the accommodation mode, i.e., normal or anywhere. The function also allocates and calculates TotalP[][] from NOfPLocal[][][] at the first call of it (and thus of transbound2()) and let primaryParts and totalParts have the number of particles the local node accommodates, i.e., the sum of TotalP[][].

```
int
transbound2(int currmode, int stats, int level) {
  int ret=MODE_NORM_SEC, nn=nOfNodes, ns=nOfSpecies, nnns2=2*nn*ns;
  int i, s, tp;

  stats = stats && statsMode;
  currmode = transbound1(currmode, stats, level);
```

The next part being the heart of balancing examination is very similar to that of transbound1() but the functions called in it are level-2's ones, try_primary2(), try_stable2() and rebalance2() which perform particle transfer in addition to the scheduling of it.

```
  if (try_primary2(currmode, level, stats))  ret = MODE_NORM_PRI;
  else if (!Mode_PS(currmode) || !try_stable2(currmode, level, stats)) {
    rebalance2(currmode, level, stats);  ret = MODE_REB_SEC;
  }
```

Finally, also as done in transbound1(), we clear NOfPLocal[][][] and copy TotalPNext[][] to its substance TotalP[][]. In addition, we also clear InjectedParticles[0][][] $= q^{\mathrm{inj}}(n)$ and nOfInjections $= Q_n^{\mathrm{inj}}$ to indicate we have no injected particles at the beginning of the next simulation step, and set totalParts and its shadow pointed by totalLocalParticles to the sum of TotalP[p][s] for all $p \in [0,1]$ and $s \in [0, S-1]$ to memorize the total number of particles the local node accommodates at the beginning of the next simulation step, i.e., before any injections and removals. Then we return to the simulator body with the return value defined in §4.3.10 for transbound1(), setting it to currMode also as done in transbound1().

```
  for (i=0; i<nnns2; i++) NOfPLocal[i] = 0;
  for (s=0,tp=0; s<ns*2; s++) {
    TotalP[s] = TotalPNext[s];  tp += TotalPNext[s];
  }
  for (s=0; s<ns*2; s++)  InjectedParticles[s] = 0;
  totalParts = *totalLocalParticles = tp;  nOfInjections = 0;
  return((currMode=ret));
}
```

4.5.5 try_primary2()

try_primary2() The function try_primary2(), called solely from transbound2(), examines if we can stay in or turn to primary mode. If so, the local node gathers all the particles in its primary subdomain from other nodes. The function has three arguments currmode, level and stats whose meanings are perfectly equivalent to those of its level-1 counterpart try_primary1().

First we call the level-1 counterpart try_primary1() to examine if the next execution mode is primary. If not, we simply return to its caller transbound2() with the return value

FALSE to indicate the mode will be secondary. Then we start particle transfer at first calling move_to_sendbuf_primary() which moves the particles outside of the primary subdomain of the local node and thus to be sent to other nodes, from the particle buffer Particles[] to the send buffer SendBuf[], while primary particles are packed in Particles[] to form $pbuf(0,s)$ for each species s together with the receive buffer $rbuf(0,s)$ for s located at the head or tail of $pbuf(0,s)$. It also sets SendBufDisps[s][m] to be the displacement of the block $sbuf(s,m)$ from the head of SendBuf[], and RecvBufBases[0][s] to point $rbuf(0,s)$.

Then we call exchange_primary_particles() to send particles in SendBuf[] to other nodes and to receive particles into $rbuf(0,s)$ in Particles[] from other nodes.

Finally we finish this function and return to transbound2() with TRUE to tell it we will be in primary mode in the next simulation step, after setting primaryParts and its shadow pointed by secondaryBase to the total number of particles the local node n accommodates, i.e., the number of particles in the primary subdomain, TotalPGlobal[n].

```
static int
try_primary2(int currmode, int level, int stats) {

  if (!try_primary1(currmode, level, stats))  return(FALSE);
  move_to_sendbuf_primary(Mode_PS(currmode), stats);
  exchange_primary_particles(currmode, stats);
  primaryParts = *secondaryBase = TotalPGlobal[myRank];
  return(TRUE);
}
```

4.5.6 exchange_primary_particles()

exchange_primary_particles()

The function exchange_primary_particles(), called from try_primary2() and some functions of level-4 or higher, sends and receives particles to/from other nodes when we will be in primary mode in the next simulation step. The particles of species s to be sent to the node m are in $sbuf(s,m)$ of SendBuf[] pointed by SendBufDisps[s][m], while those to be received are in $rbuf(0,s)$ of Particles[] pointed by RecvBufBases[0][s].

```
void
exchange_primary_particles(int currmode, int stats) {
  int i, s, nn=nOfNodes, ns=nOfSpecies, nnns=nn*ns, me=myRank;
  int *np, *rnp, *sbd;
```

We perform the particle transfer communications, after starting timing measurement of this process by oh1_stats_time() with the key STATS_TB_COMM. If we are already in primary mode and the accommodation mode is normal, all the particles the local node has to receive should be found in its neighborinhg nodes and those the local node has to send should be destined for the neighboring nodes too. Thus, for each $s \in [0, S-1]$, we pick neigboring nodes $n_d(k)$ = DstNeighbors[k] and their oppositional ones $n_s(k)$ = SrcNeighbors[k] for all $k \in [0, 3^D-1]$ so that the local node sends its accommodating particles of species s in the subdomain $n_d(k)$ to the node $n_d(k)$ simultaneously receiving its primary ones of species s accommodated by the node $n_s(k)$ from it by MPI_Sendrecv(). More accurately, the local node n performs neighboring communication exactly one send and receive for each neighbor node as follows.

1. If $n_d(k) = n$, it must be $n_s(k) = n$ by definition and symmetricity of the self neighboring. Thus we skip the k-th neighbor to avoid self communication.

2. If $n_d(k) \geq 0$ and $n_s(k) \geq 0$ meaning they are first appearence in the neighbor arrays, particles are transferred by `MPI_Sendrecv()`.

3. If either $n_d(k) < 0$ or $n_s(k) < 0$, particles are transferred by a one-way communication `MPI_Send()` or `MPI_Recv()` respectively, because we have already sent/received particles to/from the node correspoinding to $n_d(k) < 0$ or $n_s(k) < 0$. This situation may occur if the neighboring configuration is explicitly given by the simulator body through the argument `nbor` of `oh1_init()` (and thus `oh2_init()`).

4. Otherwise, i.e., $n_d(k) < 0$ and $n_s(k) < 0$, we skip the communication for k-th neighbors because we have already performed it.

Note that since it is assured that the local node n is $n_d(k)$ ($n_s(k)$) of the node $n_s(k)$ ($n_d(k)$), the blocking communications should be safely performed without deadlock.

As for the arguments of the communication functions, we give the followings for the local node n and a receiver `dest` $= n_d(k)$ and a sender `source` $= n_s(k)$.

- `sendbuf` is the pointer to $sbuf(s, n_d(k))$ and thus `SendBuf` + `SendBufDisps`$[s][n_d(k)]$.

- `sendcount` is the number of particles of species s in the subdomain $n_d(k)$ accommodated by the local node n as its primary ones and thus $q(n)[0][s][n_d(k)] =$ `NOfPLocal`$[0][s][n_d(k)]$.

- `recvbuf` is the pointer to the receive buffer $rbuf(0, s)/n_s(k)$ for the particles of species s receiving from the node $n_s(k)$ and thus;

$$\begin{aligned} &\texttt{RecvBufBases}[0][s] + \sum_{\substack{i<k \\ n_s(i)\geq 0}} q(n_s(i))[0][s][n] \\ &= \texttt{RecvBufBases}[0][s] + \sum_{\substack{i<k \\ n_s(i)\geq 0}} \texttt{NOfPrimaries}[0][s][n_s(i)] \end{aligned}$$

- `recvcount` is the number of particles of species s in the subdomain n accommodated by the node $n_s(k)$ and thus $q(n_s(k))[0][s][n] =$ `NOfPrimaries`$[0][s][n_s(k)]$. Therefore, the displacement of the receive buffer $rbuf(0, s)/n_s(k)$ from the top of $rbuf(0, s)$ pointed by `RecvBufBases`$[0][s]$ is obtained by summing up the values given to `recvcount` as shown above.

- `sendtype` and `recvtype` are commmonly `T_Particle`.

- `sendtag` and `recvtag` are commonly zero because we have no reason to distinguish the communications each other.

Note that we do not skip the communication even if `sendcount` or `recvcount` is zero to avoid coding complication, although skipping could give small but non-negligible performance improvement and should be safely implemented.

```
if (stats) oh1_stats_time(STATS_TB_COMM, 0);
np = NOfPLocal;                      /* &NOfPLocal[0][0][0] */
rnp = NOfPrimaries;                  /* &NOfPrimaries[0][0][0] */
```

```
sbd = SendBufDisps;                    /* SendBufDisps[0][0] */
if (currmode==MODE_NORM_PRI) {
  for (s=0; s<ns; s++,np+=nn,rnp+=nn,sbd+=nn) {
                                       /* np=&NOfPLocal[0][s][0] */
                                       /* rnp=&NOfPrimaries[0][s][0] */
                                       /* sbd=&SendBufDisps[s][0] */
    struct S_particle *rb;
    rb = RecvBufBases[s];              /* RecvBufBases[0][s] */
    for (i=0; i<OH_NEIGHBORS; i++) {
      int dst=DstNeighbors[i];
      int src=SrcNeighbors[i];
      int rc;
      MPI_Status st;
      if (dst==me) continue;
      if (src>=0) {
        rc = rnp[src];                 /* NOfPrimaries[0][s][src] */
        if (dst>=0)
          MPI_Sendrecv(SendBuf+sbd[dst], np[dst], T_Particle, dst, 0,
                       rb, rc, T_Particle, src, 0, MCW, &st);
        else
          MPI_Recv(rb, rc, T_Particle, src, 0, MCW, &st);
        rb += rc;
      } else if (dst>=0)
        MPI_Send(SendBuf+sbd[dst], np[dst], T_Particle, dst, 0, MCW);
    }
  }
```

On the other hand, if the mode has turned from secondary to primary or the accommodation mode is anywhere, some particles in the primary subdomain of the local node n can be found in any nodes. Therefore, for each species s, we perform an all-to-all communication by `MPI_Alltoallv()` with the following arguments.

- `sendbuf` is always `SendBuf` regardless of s because we specify each of $sbuf(s,m)$ for the node m by `sdispls` = `SendBufDisps`[].
- `sendcounts` is `NOfPLocal`[0][s][] but each element of it is incremented by the secondary counterpart `NOfPLocal`[1][s][] so that it has $q(n)[0][s][m] + q(n)[1][s][m]$ for all $m \in [0, N-1]$ in case of we are in secondary mode. Note that the local node's own `NOfPLocal`[0][s][n] is set to zero by `move_to_sendbuf_primary()` but `NOfPLocal`[1][s][n] is not to move particles which resides in the subdomain n and was secondary from `SendBuf`[] back to `Particles`[] by the self communication taken in `MPI_Alltoallv()`.
- `sdispls` is `SendBufDisps`[s][] to specify $sbuf(s,m)$ for each node m.
- `recvbuf` is `RecvBufBases`[0][s] which points the receive buffer $rbuf(0,s)$ in `Particles`[] for s.
- `recvcounts` is `TempArray`[] whose element [m] has the following to represent the number of particles of species s in the subdomain n currently accommodated by the node m, excepting n's own primary particles[45].

$$\texttt{TempArray}[m] = \begin{cases} q(m)[1][s][n] & m = n \\ q(m)[0][s][n] + q(m)[1][s][n] & m \neq n \end{cases}$$

[45] Instead of `TempArray`[], we may use `NOfPrimaries`[0][s][] destructively adding its secondary counterparts to it, but dare to use `TempArray`[] by some historical reason of the implementaion.

$$= \begin{cases} \texttt{NOfPrimaries}[1][s][m] & m = n \\ \texttt{NOfPrimaries}[0][s][m] + \texttt{NOfPrimaries}[1][s][m] & m \neq n \end{cases}$$

- `rdispls` is `RecvBufDisps`[] whose element m has $\sum_{i<m}$ `TempArray`$[m]$ = $\sum_{i<m}$ `rcounts`$[m]$ for the block $rbuf(0,s)/m$ in the receive buffer $rbuf(0,s)$ to which we receive the particles from the node m.
- `sendtype` and `recvtype` are commmonly `T_Particle`.

```
  } else {
    for (s=0; s<ns; s++,np+=nn,rnp+=nn,sbd+=nn) {
                                        /* np=&NOfPLocal[0][s][0] */
                                        /* sbd=&SendBufDisps[s][0] */
                                        /* rnp=&NOfPrimaries[0][s][0] */
      int rdisp=0;
      rnp[me] = 0;                      /* &NOfPrimaries[0][s][me] */
      for (i=0; i<nn; i++) {
        int rc = rnp[i] + rnp[i+nnns];
                        /* NOfPrimaries[0][s][i]+ NofPrimaries[1][s][i] */
        TempArray[i] = rc;
        RecvBufDisps[i] = rdisp;
        rdisp += rc;
        np[i] += np[i+nnns];            /* += NOfPLocal[1][s][i] */
      }
      MPI_Alltoallv(SendBuf, np, sbd, T_Particle,
                    RecvBufBases[s], TempArray, RecvBufDisps, T_Particle, MCW);
    }
  }
}
```

4.5.7 `try_stable2()`

`try_stable2()` The function `try_stable2()`, solely called from `transbound2()`, examines if the current helpand-helper configuration sustains the particle movements crossing subdomain boundaries which can bring intolerable load imbalance. The examination is done by calling its level-1 counterpart `try_stable1()` simply passing all the arguments of the function itself to the counterpart, because the meanings of them are perfectly equivalent to those of the counterpart. If the examination passes, we perform an all-to-all type particle transfer by calling `exchange_particles()` with the following arguments, before returning to `transbound2()` with the return value of `TRUE`.

- `secrlist` is the pointer to the secondary receiving block and thus `CommList` + `SLHeadTail`[1] because helpand-helper configuration has been kept. Similarly, the size of the block `secrlsize` is given by `SecSLHeadTail`[0].
- `oldparent` is the current helpand of the local node n and thus $parent(n)$ = `Nodes`$[n]$`.parentid`.
- `neighboring` is true if and only if the argument `currmode` of the function is equal to `MODE_NORM_SEC` indicating normal accommodation.
- `currmode` and `stats` are simply those passed to the function.

```
static int
try_stable2(int currmode, int level, int stats) {

  if (!try_stable1(currmode, level, stats)) return(FALSE);
  exchange_particles(CommList+SLHeadTail[1], SecSLHeadTail[0],
                     Nodes[myRank].parentid, currmode==MODE_NORM_SEC,
                     currmode, stats);
  return(TRUE);
}
```

4.5.8 rebalance2()

rebalance2()

The function rebalance2(), solely called from transbound2(), builds the new family tree to rebalance the load among nodes by calling its level-1 counterpart rebalance1() simply passing all the arguments of the function itself to the counterpart, because the meanings of them are perfectly equivalent to those of the counterpart. Then, before the particle transfer by exchange_particles(), it clears InjectedParticles[0][1][] $= q^{\mathrm{inj}}(n)[1][]$ if some particles are injected (nOfInjections $= Q_n^{\mathrm{inj}} > 0$), the local node n had a parent, and the old parent and new one are different, because particles injected into old secondary subdomain are simply slown away to the old parent or its new children. Note that the particles injected into the new secondary subdomain accidentally are regarded as primary particles and thus they are transferred from the primary subdomain to secondary one.

Then it performs an all-to-all type particle transfer by calling exchange_particles() with the following arguments.

- secrlist and secrlsize are SecRList and SecRLSize which are set to the head and size of secondary receiving or alternative secondary receiving block broadcasted from the new helpand by make_comm_count() called in rebalance1().
- oldparent is the helpand of the local node n in the old helpand-helper configuration if we are in secondary mode with normal accommodation. That is, if currmode argument of the function is MODE_NORM_SEC, oldparent is NodesNext[n].parentid because the old configuration is kept in NodesNext[] by rebalance1(). Otherwise, oldparent is -1 to indicate no information was given from the old helpand because we are in primary mode or with anywhere accommodataion.
- neighboring is true if and only if Mode_Is_Norm() for the argument currmode of the function is true indicating normal accommodation.
- currmode and stats are simply those passed to the function.

```
static void
rebalance2(int currmode, int level, int stats) {
  int me=myRank, ns=nOfSpecies, s, oldp, newp;

  rebalance1(currmode, level, stats);
  oldp = NodesNext[me].parentid;  newp=Nodes[me].parentid;
  if (nOfInjections && oldp>=0 && oldp!=newp)
    for (s=0; s<ns; s++)  InjectedParticles[ns+s] = 0;
  if (Mode_Is_Norm(currmode))
```

```
    exchange_particles(SecRList, SecRLSize,
                       Mode_PS(currmode) ? oldp : -1,
                       1, currmode, stats);
  else
    exchange_particles(SecRList, SecRLSize, -1, 0, currmode, stats);
}
```

4.5.9 move_to_sendbuf_primary()

move_to_sendbuf_primary()

The function move_to_sendbuf_primary(), called from try_primary2() and some library functions of level-4 or higher, moves particles not residing in the primary subdomain of the local node from the particle buffer Particles[] to the send buffer SendBuf[]. The particles residing in the primary subdomain are also moved in Particles[] so that they are contiguously packed in each $pbuf(0,s)$ for each species $s \in [0,S-1]$, each $pbuf(0,s)$ has a receive buffer $rbuf(0,s)$ of an appropriate size at its head or tail, and all $pbuf(0,s)$ are also contiguously aligned. The function also takes care of particles injected by oh2_inject_particle() or its higher level counterparts and located at the tail of Particles[], by at first moving them to SendBuf[] regardless of their residing subdomains and then moving primary ones from SendBuf[] back to Particles[]. The function is also responsible to let SendBufDisps$[s][m]$, RecvBufBases$[0][s]$ and TotalPNext$[0][s]$ have appropriate values for all $s \in [0,S-1]$ and $m \in [0,N-1]$.

The function is given two arguments, secondary being true (1) if and only if we were in the secondary mode in the last step, and stats being true (non-zero) if and only if the execution time spent in the process to move particles has to be measured.

```
void
move_to_sendbuf_primary(int secondary, int stats) {
  int me=myRank, ns=nOfSpecies, nn=nOfNodes, nnns=nn*ns;
  int s, i, j, *pp;
```

First we call oh1_stats_time() with the key STATS_TB_MOVE to measure the time spent in the function if the argument stats is true. Then, for each $s \in [0,S-1]$, we clear NOfPLocal$[0][s][n]$ for the local node n to indicate primary particles of the node are not moved to SendBuf[] but stay in Particles[]. Note that we keep NOfPLocal$[1][s][n]$ unchanged so that secondary particles in the primary subdomain, which are accommodated by the local node and incidentally moved from the subdomain which the local node was responsible for as its secondary subdomain in the last step, are moved to SendBuf[] and then sent to the local node itself.

We also set TotalPNext$[0][s]$ as follows;

$$\texttt{TotalPNext}[0][s] = \sum_{p\in\{0,1\}} \sum_{m=0}^{N-1} \texttt{NOfPrimaries}[p][s][m] = \sum_{p\in\{0,1\}} \sum_{m=0}^{N-1} q(m)[p][s][n]$$

to indicate all the particles of species s in the primary subdomain of the local node is accommodated by the node, while TotalPNext$[1][s]$ is cleared with 0 because we will have no secondary particles.

```
  if (stats) oh1_stats_time(STATS_TB_MOVE, 0);
  for (s=0,i=me,pp=NOfPrimaries; s<ns; s++,i+=nn,pp+=nn) {
    int t = 0;
```

```
  NOfPLocal[i] = 0;                          /* NOfPLocal[0][s][me] */
  for (j=0; j<nn; j++) t += pp[j] + pp[j+nnns];
                          /* NOfPrimaries[0][s][j] + NOfPrimaries[1][s][j] */
  TotalPNext[s] = t;                         /* TotalPNext[0][s] */
  TotalPNext[ns+s] = 0;                      /* TotalPNext[1][s] */
}
```

Next, we call `set_sendbuf_disps()`, with the first argument `secondary` and second -1 meaning the local node does not have parent, to calculate values of `SendBufDisps`$[S][N]$. We then call `move_injected_to_sendbuf()` to move injected particles to `SendBuf`[] if we have any of them, i.e., `nOfInjections` $= Q_n^{\rm inj} > 0$.

```
set_sendbuf_disps(secondary, -1);
if (nOfInjections)  move_injected_to_sendbuf();
```

Now we move the primary particles to `SendBuf`[] or pack them in `Particles`[] by calling `move_to_sendbuf_uw()` for the species whose block $pbuf(0, s)$ will not have head and tail addresses larger than their current ones and thus have some particles moving *upward*, or toward smaller addresses. That is, these blocks can be scanned in ascending order and be packed safely without any hazard to destroy the contents of other (following) blocks. On the other hand, the blocks which will have head and tail addresses larger than their current ones will be moved *downward* and thus are skipped by the function because they should be scanned in descending order *after* we process all of the *upward* blocks.

The arguments to be given to the function are as follows.

- `ps` is 0 to scan primary particles.
- `me` is the rank of the local node n to identify the primary particles in the next step.
- `putmes` is the slice `NOfPLocal`$[0][S][n]$ whose elements have been zero-cleared to indicates no particles in the primary subdomain are sent.
- `cbase` is 0 to tell the function to start the scan from the head of `Particles`[].
- `ctp` is `TotalP`$[0][S]$ to show the size of the current $pbuf(0, s)$ is `TotalP`$[0][s]$.
- `nbase` is 0 to tell the function to pack particles staying in the local node from the head of `Particles`[].
- `ntp` is `TotalPNext`$[0][S]$ to show the size of the next $pbuf(0, s)$ is `TotalPNext`$[0][s]$.
- `rbb` is `RecvBufBases`$[0][S]$ to tell the function that the head of $rbuf(0, s)$ should be set into `RecvBufBases`$[0][s]$.

```
move_to_sendbuf_uw(0, me, NOfPLocal+me, 0, TotalP, 0, TotalPNext,
                   RecvBufBases);
```

If we were in secondary mode in the last step, i.e. the argument `secondary` is 1, we move all secondary particles stored in `Particles`[`primaryParts`] and below to `SendBuf`[] regardless the subdomain where they reside by calling `move_to_sendbuf_uw()` again but with the following arguments.

- `ps` is 1 to scan secondary particles.

- `me` is -1 to force no particles to be judged to stay in `Particles[]`.
- `putmes` is `NULL` to indicate no subdomains are specially treated as the primary subdomain.
- `cbase` is `primaryParts` to tell the function to start the scan from the head of $pbuf(1,0)$ for secondary particles.
- `ctp` is `TotalP`$[1][S]$ to show the size of the current $pbuf(1,s)$ is `TotalP`$[1][s]$.
- `nbase` is 0 to make it sure that all the blocks $pbuf(1,s)$ is judged to move *upward*, while no particles are moved in `Particles[]` actually.
- `ntp` is `TotalPNext`$[1][S]$ to show the size of the next $pbuf(0,s)$ is `TotalPNext`$[1][s] = 0$ to make it sure that all the blocks $pbuf(1,s)$ is judged to move *upward*.
- `rbb` is `RecvBufBases`$[1][S]$ so that the function safely set the base of $rbuf(1,s) =$ `Particles` into `RecvBufBases`$[1][s]$, which will not be referred to though.

```
if (secondary) move_to_sendbuf_uw(1, -1, NULL, primaryParts, TotalP+ns,
                                  0, TotalPNext+ns, RecvBufBases+ns);
```

Now we revisit the primary particle blocks $pbuf(0,s)$ skipped by `move_to_sendbuf_uw()` due to their *downward* moving direction. We move particles in these blocks by calling `move_to_sendbuf_dw()` with arguments similar to its upward counterparts but different from it as follows.

- The third argument is named `ctail` instead of `cbase` and we give `primaryParts` to show the tail of the current primary particle buffer from which the function starts the scan.
- The fifth argument is named `ntail` instead of `nbase` and we give the number of primary particles of the local node n in the next step, `TotalPGlobal`$[n]$, to show the tail of the next primary particle buffer.
- The function does not have `rbb` argument because setting `RecvBufBases[][]` is solely done by `move_to_sendbuf_uw()`.

Note that it is unnecessary to call `move_to_sendbuf_dw()` for secondary particles even if we were in secondary mode in the last step, because all of them have been moved to `SendBuf[]` by `move_to_sendbuf_uw()` for them.

```
move_to_sendbuf_dw(0, me, NOfPLocal+me, primaryParts, TotalP,
                   TotalPGlobal[me], TotalPNext);
```

Finally, we call `set_sendbuf_disps()` again to regain the values of `SendBufDisps[][]` which have been set by the first call of it, because `move_to_sendbuf_uw()` and `move_to_sendbuf_dw()` have modified them for moving particles from `Particles[]` to `SendBuf[]`. Then we call `move_injected_from_sendbuf()` giving `InjectedParticles`$[0][0][] = q^{\mathrm{inj}}(n)[0][]$, n and `RecvBufBases`$[0][S]$ to its arguments to move injected primary particles, which have been moved to `SendBuf[]` by `move_injected_to_sendbuf()`, from $sbuf(s,n)$ in `SendBuf[]` for the local node n back to $pbuf(0,s)$ in `Particles[]`, if we have injected particles, i.e., `nOfInjections` $= Q_n^{\mathrm{inj}} > 0$.

```
    set_sendbuf_disps(secondary, -1);
    if (nOfInjections)
      move_injected_from_sendbuf(InjectedParticles, me, RecvBufBases);
  }
```

4.5.10 move_to_sendbuf_secondary()

move_to_sendbuf_secondary()

The function move_to_sendbuf_secondary(), called from exchange_particles() and some functions of level-4 or higher, moves particles to be sent to other nodes from the particle buffer Particles[] to the send buffer SendBuf[], in a similar manner move_to_sendbuf_primary() does. However, the particles to be sent are not only those residing in the subdmains other than primary or secondary ones of the local node, but also some of them resinding these responsible subdomains but being overflown from the node. The particles staying in the local node as its primary ($p = 0$) or secondary ($p = 1$) ones are also moved in Particles[] so that they are contiguously packed in each $pbuf(p, s)$ for each $p \in \{0, 1\}$ and species $s \in [0, S-1]$, each $pbuf(p, s)$ has a receive buffer $rbuf(p, s)$ of an appropriate size at its head or tail, and all $pbuf(p, s)$ are also contiguously aligned. The function also takes care of particles injected by oh2_inject_particle() and located at the tail of Particles[], by at first moving them to SendBuf[] regardless of their residing subdomains and then moving some of primary ones from SendBuf[] back to Particles[]. The function is also responsible to let SendBufDisps$[s][m]$, RecvBufBases$[p][s]$ and TotalPNext$[p][s]$ have appropriate values for all $p \in \{0, 1\}$, $s \in [0, S-1]$ and $m \in [0, N-1]$.

The function is given two arguments, secondary being true (1) if and only if we have already been in the secondary mode in the last step, and stats being true (non-zero) if and only if the execution time spent in the process to move particles has to be measured.

```
static void
move_to_sendbuf_secondary(int secondary, int stats) {
  int me=myRank, ns=nOfSpecies, ns2=ns<<1, nn=nOfNodes;
  struct S_node *node = Nodes+me;
  int put[2]={-node->get.prime, -node->get.sec}, pnext[2];
  int sec=node->parentid;
  int nnns=nn*ns;
  int *mynp[2]={NOfPLocal+me,           /* &NOfPLocal[0][0][me] */
                sec<0 ? NULL : NOfPLocal+nnns+sec};
                                        /* &NOfPLocal[1][0][sec] */
  int *mynps;
  int ps, s, i;
```

First we call oh1_stats_time() with the key STATS_TB_MOVE to measure the time spent in the function if the argument stats is true. Then, for each $p \in \{0, 1\}$ and $s \in [0, S-1]$, we modify NOfPLocal$[p][s][n]$ so that it has the number of particles in primary and secondary subdomains sent from the local node n due to overflow. We also calculate each element of TotalPNext$[p][s]$ by modifying its *base* value given by count_next_particles() or make_recv_count(), and the sum for all s to have Q_n^n and $Q_n^{parent(n)}$ (or pnext$[p]$) in the next step.

More specifically, the operations above are performed as follows. Let $m = n$ for $p = 0$ and $m = parent(n)$ for $p = 1$, and $put(p) =$ putme be the number of particles resinding its

primary ($p = 0$) or secondary ($p = 1$) subdomain, accommodated by the local node and being sent to other nodes due to overflow. That is;

$$put(p) = \begin{cases} -R_n^{\text{get}} = -\texttt{Nodes}[n].\texttt{get.prime} & p = 0 \\ -Q_n^{\text{get}} = -\texttt{Nodes}[n].\texttt{get.sec} & p = 1 \end{cases}$$

If $put(p) \leq 0$ indicating that the node have to get some particles as its primary/secondary particles, $\texttt{NOfPLocal}[p][s][m]$ are cleared with zero for all s to mean that no particles in the responsible subdomains are sent to other nodes. In this case, since $\texttt{TotalPNext}[p][s]$ was set to the total number of particles of species s to be recieved from other nodes, it is incremented by the original (before clearing) value of $\texttt{NOfPLocal}[p][s][m]$ to have the number of particles to be accommodated by the local node in the next step.

Otherwise, i.e., $put(p) > 0$, $\texttt{TotalPNext}[p][s]$ was set to zero for all s because no particles are received from other nodes, but some particles have to be sent out. We choose particles to be sent from leading species by emptying first t species such that $\Sigma(t) = \sum_{s<t} \texttt{NOfPLocal}[p][s][m] \leq put(p)$ and thus both $\texttt{NOfPLocal}[p][s][m]$ and $\texttt{TotalPNext}[p][s]$ $(= 0)$ remain unchanged for all $s < t$. As for the species t, it has some number of particles to be sent namely $put(p) - \Sigma(t)$, and remainders to stay namely $\Sigma(t+1) - put(p)$. Therefore, $\texttt{NOfPLocal}[p][t][m]$ is set to the former number while $\texttt{TotalPNext}[p][t]$ to the latter by adding it to the original value 0. The particles of remainder species of $s > t$ simply stay in the local node and thus $\texttt{NOfPLocal}[p][s][m]$ is cleared with zero moving its original value to $\texttt{TotalPNext}[p][s]$ (by adding it to the original value 0) as done in the case of $put(p) \leq 0$.

After that, if we have particles injected into the primary/secondary subdomain of the local node, we have to take care of them by further modifying the value to be set into $\texttt{NOfPLocal}[p][s][m]$ discussed above, because it is the number of particles in the primary/secondary subdomain to be sent including those of injected, but it should have the number of ordinary non-injected particles to be moved from `Particles[]` to `SendBuf[]`. Since all injected particles are moved to `SendBuf[]` regardless of their residing subdomains, we have decrease $\texttt{NOfPLocal}[p][s][m]$ by the amount of the particles injeted into the primary/secondary subdomain, namely $\texttt{InjectedParticles}[0][p][s] = q^{\text{inj}}(n)[p][s]$. More specifically, we perform the following.

$$\texttt{NOfPLocal}[p][s][m] \leftarrow \max(0,\ \texttt{NOfPLocal}[p][s][m] - q^{\text{inj}}(n)[p][m])$$

We also let $\texttt{InjectedParticles}[1][p][s]$ be the following to tell `move_injected_from_sendbuf()` how many particles should be moved from `SendBuf[]` back to `Particles[]`.

$$\texttt{InjectedParticles}[1][p][s] = \max(0,\ q^{\text{inj}}(n)[p][s] - \texttt{NOfPLocal}[p][s][m])$$

That is, if $\texttt{NOfPLocal}[p][s][m] > q^{\text{inj}}(n)[p][s]$, we send all injected particles and make non-injected ones of the same amount stay. Otherwise, we send some of injected particles while all non-injected are made stay.

In any cases, Q_n^n and $Q_n^{parent(n)}$ are obtained by summing up the updated values of $\texttt{TotalPNext}[p][s]$ for all s. Note that, however, $parent(n)$ can be -1 to indicate the local node is the root of the helpand-helper tree. In this case, we skip the update of $\texttt{NOfPLocal}[1][][]$ and $\texttt{TotalPNext}[1][]$ $(= 0)$ to make them remain unchanged because all the particles which *was* secondary in the last step, if any, have to be sent to other nodes, and the local node will not have any secondary particles in the next step. In addtion, the sum of secondary particles, conceptually Q_n^{-1}, is set to zero.

```
if (stats) oh1_stats_time(STATS_TB_MOVE, 1);
for (ps=0,i=0; ps<2; ps++) {
  int putme=put[ps], npnext=0;
  mynps=mynp[ps];
  if (mynps==NULL) {
    pnext[ps] = 0;  break;
  }
  if (putme<0) putme = 0;
  for (s=0; s<ns; s++,i++,mynps+=nn) {        /* i=ps*ns+s */
    int stay=*mynps;                          /* NofPLocal[ps][s][me/sec] */
    int tpni=TotalPNext[i];                   /* TotalPNext[ps][s] */
    int inj=InjectedParticles[i];             /* InjectedParticles[ps][s] */
    if (putme<stay) {
      TotalPNext[i] = tpni = tpni + stay - putme;
      stay = putme;
      putme = 0;
    } else
      putme -= stay;
    if (stay>inj) {
      InjectedParticles[ns2+i] = 0;  *mynps = stay - inj;
    } else {
      InjectedParticles[ns2+i] = inj - stay;  *mynps = 0;
    }
    npnext += tpni;
  }
  pnext[ps] = npnext;
}
```

Next, we call `set_sendbuf_disps()` with the first argument `secondary` and second $parent(n)$ of the local node n meaning we will be in secondary mode with the parent (or without if negative), to calculate values of `SendBufDisps`$[S][N]$. We then call `move_injected_to_sendbuf()` to move injected particles to `SendBuf`[] if we have any of them, i.e., `nOfInjections` $= Q_n^{\rm inj} > 0$.

```
set_sendbuf_disps(secondary, sec);
if (nOfInjections)  move_injected_to_sendbuf();
```

Now we move the primary particles to `SendBuf`[] or pack them in `Particles`[] by calling `move_to_sendbuf_uw()` for the species whose block $pbuf(0,s)$ will not have head and tail addresses larger than their current ones, as done in `move_to_sendbuf_primary()` with the following arguements.

- `ps` is 0 to scan primary particles.
- `me` is the rank of the local node n to identify the primary particles in the next step.
- `putmes` is the slice `NOfPLocal`$[0][S][n]$ whose elements have been set to the number of particles residing in the primary subdomain but being moved to `SendBuf`[] and sent to other nodes.
- `cbase` is 0 to tell the function to start the scan from the head of `Particles`[].
- `ctp` is `TotalP`$[0][S]$ to show the size of the current $pbuf(0,s)$ is `TotalP`$[0][s]$.

- `nbase` is 0 to tell the function to pack particles staying in the local node from the head of `Particles`[].
- `ntp` is `TotalPNext`[0][S] to show the size of the next $pbuf(0, s)$ is `TotalPNext`[0][s].
- `rbb` is `RecvBufBases`[0][S] to tell the function that the head of $rbuf(0, s)$ should be set into `RecvBufBases`[0][s].

```
move_to_sendbuf_uw(0, me, mynp[0],            /* &NOfPLocal[0][0][me] */
                   0, TotalP, 0, TotalPNext, RecvBufBases);
```

If we have already been in secondary mode, i.e. `secondary` = 1, we move secondary particles stored in `Particles`[`primaryParts`] and below to `SendBuf`[] or pack them in the region from `Particles`[Q_n^n] to `Particles`[$Q_n^n + Q_n^{parent(n)} - 1$]. This is done by calling `move_to_sendbuf_uw()` and `move_to_sendbuf_dw()` giving them the following arguments.

- `ps` is 1 to scan secondary particles.
- `me` is $parent(n)$ = `Nodes`[n].`parentid` for the rank of the local node's helpand or -1 if the local node is the root of the helpand-helper tree.
- `putmes` is the slice `NOfPLocal`[1][S][n] whose elements have been set to the number of particles residing in the secondary subdomain but being moved to `SendBuf`[] and sent to other nodes, if $parent(n) \geq 0$. Otherwise, i.e., if $parent(n) = -1$ for the local node rooting the tree, this argument is `NULL`.
- `cbase` for `move_to_sendbuf_uw()` is `primaryParts` to tell the function to start the forward scan from the head of $pbuf(1, 0)$, while `ctail` for `move_to_sendbuf_dw()` is `totalParts` for the tail (plus one) of $pbuf(1, S-1)$ from which it scans reversely.
- `ctp` is `TotalP`[1][S] to show the size of the current $pbuf(1, s)$ is `TotalP`[1][s].
- `nbase` for `move_to_sendbuf_uw()` is Q_n^n to tell the function to pack secondary particles staying in the local node following those of primary, while `ntail` for `move_to_sendbuf_dw()` is $Q_n^n + Q_n^{parent(n)}$ to show the tail (plus one) of the particle buffer in the next step.
- `ntp` is `TotalPNext`[1][S] to show the size of the next $pbuf(1, s)$ is `TotalPNext`[1][s].
- `rbb`, only for `move_to_sendbuf_uw()`, is `RecvBufBases`[1][S] to tell the function that the head of $rbuf(1, s)$ should be set into `RecvBufBases`[1][s].

```
if (secondary) {
  move_to_sendbuf_uw(1, sec, mynp[1],         /* &NOfPLocal[1][0][sec] */
                     primaryParts, TotalP+ns, pnext[0], TotalPNext+ns,
                     RecvBufBases+ns);
  move_to_sendbuf_dw(1, sec, mynp[1], totalParts, TotalP+ns,
                     pnext[0]+pnext[1], TotalPNext+ns);
```

Otherwise, i.e., if we were in primary mode in the last step, we have no secondary particles to be sent but will get them from other nodes. Since the number of particles to be received for each species s has been set in `TotalPNext`[1][s], we set `RecvBufBases`[1][s] as follows so that

$rbuf(1,0)$ follows $pbuf(0,S-1)$ of the next step and $rbuf(1,s)$ is as large as `TotalPNext`$[1][s]$ for each s.

$$\texttt{RecvBufBases}[1][s] = Q_n^n + \sum_{t=0}^{s-1} \texttt{TotalPNext}[1][t]$$

```
    } else {
      struct S_particle *rbb=Particles+pnext[0];
      for (s=0; s<ns; s++) {
        RecvBufBases[ns+s] = rbb;                /* RecvBufBases[1][s] */
        rbb += TotalPNext[ns+s];                 /* TotalPNext[1][s] */
      }
    }
```

Now we revisit the primary particle blocks $pbuf(0,s)$ skipped by `move_to_sendbuf_uw()` due to their *downward* moving direction. We move particles in these blocks by calling `move_to_sendbuf_dw()` as done in `move_to_sendbuf_primary()`, giving its `ctail` and `ntail` arguements the indices of the tail (plus one) of $pbuf(0,S-1)$ in the current and next step, namely `primaryParts` and Q_n^n respectively.

```
    move_to_sendbuf_dw(0, me, mynp[0], primaryParts, TotalP, pnext[0],
                       TotalPNext);
```

Then, we call `set_sendbuf_disps()` again to regain the values of `SendBufDisps`[][]. Then we call `move_injected_from_sendbuf()` giving `InjectedParticles`$[1][0][S]$, n and `RecvBufBases`$[0][S]$ to its arguments to move (some of) injected primary particles of the local node n from `SendBuf`[] back to `Particles`[] if `nOfInjections` $= Q_n^{\rm inj} > 0$. We also call it again with `InjectedParticles`$[1][1][[S]$, $parent(n)$ and `RecvBufBases`$[1][S]$ for injected secondary particles if the local node n is not the root. Finally, we set `primaryParts` and its shadow pointed by `secondaryBase` to Q_n^n.

```
    set_sendbuf_disps(secondary, sec);
    if (nOfInjections) {
      move_injected_from_sendbuf(InjectedParticles+ns2, me, RecvBufBases);
      if (sec>=0)
        move_injected_from_sendbuf(InjectedParticles+ns2+ns, sec,
                                   RecvBufBases+ns);
    }
    primaryParts = *secondaryBase = pnext[0];
  }
```

4.5.11 `set_sendbuf_disps()`

`set_sendbuf_disps()`

The function `set_sendbuf_disps()`, called from `move_to_sendbuf_primary()`, `move_to_sendbuf_secondary()` and higher level library functions such as those in level-4p, calculates values of `SendBufDisps`$[S][N]$ so that its element $[s][m]$ has the displacement to the head of $sbuf(s,m)$ in `SendBuf`[]. Since the number of primary particles of the species s to be sent to the node m is in `NOfPLocal`$[0][s][m]$ and, if we are in secondary mode and thus the argument of the function `secondary` is true, that of secondary particles is in `NOfPLocal`$[1][s][m]$, `SendBufDisps`$[s][m]$ is fundamentally defined as follows.

$$\texttt{SendBufDisps}[s][m] = \sum_{i=0}^{m-1} \texttt{NOfPLocal}[0][s][i] + \begin{cases} 0 & \neg\texttt{secondary} \\ \displaystyle\sum_{i=0}^{m-1} \texttt{NOfPLocal}[1][s][i] & \texttt{secondary} \end{cases}$$

However, we have to take care of an additional factor, injected particles which have to be moved to $sbuf(s,m)$ for the local node $m = n$ or its parent $m = parent(n)$ and whose count for a species s is recorded in `InjectedParticles`$[0][p][s] = q^{\mathrm{inj}}(n)[p][s]$ where $p = 0$ for primary and $p = 1$ for secondary particles respectively. Therefore, the equation above is modified as follows, where $parent(n)$ is given through the argument `parent`.

$$
\begin{aligned}
n' &= \{n, parent(n)\}[p] \\
q(p,s,m) &= \texttt{NOfPLocal}[p][s][m] + \begin{cases} 0 & m \neq n' \\ q^{\mathrm{inj}}(n)[p][s] & m = n' \end{cases} \\
q(s,m) &= q(0,s,m) + \begin{cases} 0 & \neg\texttt{secondary} \\ q(1,s,m) & \texttt{secondary} \end{cases} \\
\texttt{SendBufDisps}[s][m] &= \sum_{i=0}^{m-1} q(s,m)
\end{aligned}
$$

The equation above implies that `NOfPLocal`$[0][s][n]$ has the number of primary particles to be sent to helpers (thus should be 0 if we will be in primary mode) while `NOfPLocal`$[1][s][parent(n)]$ has that to be sent to the helpand and/or sibling helpers. It also means that `NOfPLocal`$[1][s][n]$ and `NOfPLocal`$[0][s][parent(n)]$ are not special but simply represents number of particles visiting to primary/secondary domains including those will be accommodated by the local node by self communication.

```
void
set_sendbuf_disps(int secondary, int parent) {
  int nn=nOfNodes, ns=nOfSpecies, me=myRank;
  int i, j, k, s, disp;

  for (s=0,i=0,disp=0; s<ns; s++) {
    for (k=0; k<nn; k++,i++) {
      SendBufDisps[i] = disp;                    /* SendBufDisps[s][k] */
      disp += NOfPLocal[i];                      /* NOfPLocal[0][s][k] */
      if (k==me)  disp += InjectedParticles[s]; /* InjectedParticles[0][s] */
    }
  }
  if (secondary) {
    for (s=0,j=0,disp=0; s<ns; s++) {
      for (k=0; k<nn; k++,i++,j++) {
        SendBufDisps[j] += disp;                 /* SendBufDisps[s][k] */
        disp += NOfPLocal[i];                    /* NOfPLocal[1][s][k] */
        if (k==parent)  disp += InjectedParticles[ns+s];
      }                                          /* InjectedParticles[1][s] */
    }
  }
}
```

4.5.12 `exchange_particles()`

`exchange_particles()` The function `exchange_particles()`, called from `try_stable2()`, `rebalance2()` and higher level library functions such as those in level-4p, performs an all-to-all type particle transfer when we will be in secondary mode in the next simulation step. The function is given the following arguments.

- `secrlist` points the secondary receiving or alternative secondary receiving block in `CommList[]`, which is given from the helpand of the local node in the next simulation step, and `secrlsize` is its size.
- `oldparent` is the helpand of the local node in the last simulation step. More specifically, it has the followings.
  - If the function is called from `try_stable2()` and thus the last simulation step is in secondary mode and the helpand-helper configuration is kept, it has the rank of the helpand of the current configuration regardless of the accommodation mode[46].
  - In the case that the function is called from `rebalance2()` by which helpand-helper configuration is rebuild, it has the rank of the helpand in the old configuration if the last simulation step is in secondary mode. Otherwise, i.e., if we were in primary mode, the argument has -1 to indicate no meaningful information is given from the configuration before rebuilding[47].
- `neighboring` is true (non-zero) if the accommodation mode is normal. Otherwise, i.e., if anywhere accommodation mode, it is false (zero).
- `currmode` is referred to only for examining the execution mode in the last simulation step is primary or secondary[48].
- `stats` is true (non-zero) if and only if timing measurements for the process of particle movement and transfer are required.

```
void
exchange_particles(struct S_commlist *secrlist, int secrlsize, int oldparent,
                   int neighboring, int currmode, int stats) {
  int me=myRank, nn=nOfNodes, ns=nOfSpecies, nnns=nn*ns;
  int newparent=Nodes[me].parentid;
  int s, i, req;
```

The first job of the function is to call `move_to_sendbuf_secondary()` to move particles to be sent to other nodes from the particle buffer `Particles[]` to the send buffer `SendBuf[]`, and to pack primary and secondary particles which are kept accommodated in the local node's `Particles[]` in which it reserves receive buffers for particle reception. It also sets `SendBufDisps`$[s][m]$ to be the displacement of the block $sbuf(s,m)$ from the head of `SendBuf[]`, and `RecvBufBases`$[p][s]$ to point $rbuf(p,s)$.

Then we call `oh1_stats_time()` with the key `STATS_TB_COMM`, if the argument `stats` is true, to measure the time spent for the particle transfer communication.

```
  move_to_sendbuf_secondary(Mode_PS(currmode), stats);
  if (stats) oh1_stats_time(STATS_TB_COMM, 1);
```

If the argument `neighboring` is true to indicate that the accommodation mode is normal, we perform particle transfer scanning the transfer schedules stored in `CommList[]` by `receive_particles()` and `send_particles()` as follows.

[46] If the accommodataion mode is anywhere, this argument is not referred to by the function.

[47] If the accommodation mode is anywhere, the argument also has -1 but is not referred to by the function.

[48] Since `currmode` has the information of `neighboring`, it is simply redundant to have two arguments separately, but we have both of them due to some historical reason of the implementaion.

- The primary receiving block, which is located at the top of `CommList`[] and whose size is `SLHeadTail`[0], is scanned by `receive_particles()` to receive primary particles and/or to send them to the helpers of the local node if it is required to *push down* the primary particles to them in order to make room to accommodate secondary particles. This scan and transfer are performed always.

- If the local node n is not the root of the helpand-helper tree, i.e., $parent(n) =$ `Nodes`$[n]$`.parentid` ≥ 0, the secondary receiving or alternative secondary receiving block specified by the arguments `secrlist` and `secrlsize` is scanned by `receive_particles()` to receive secondary particles and/or to send them to the members of the family to which the local node belongs to as a helper due to secondary particle overflow.

- If the helpand-helper configuration is rebuild by `rebalance2()` in secondary mode and the local node has new helpand different from the old one, the secondary receiving block given by the old helpand, which starts from `CommList` + `SLHeadTail`[1] and whose size is `SecSLHeadTail`[0], is scanned by `receive_particles()` to send (not to receive) particles in the subdomain which were secondary one of the local node.

- The primary sending block, which starts from `CommList` + `SLHeadTail`[0] and whose size is `SLHeadTail`[1] − `SLHeadTail`[0], is scanned by `send_particles()` to send particles to the nodes responsible for the subdomain neigboring to the primary of the local node. We have to be aware of the possibility that the primary sending block is given from the node which is the helpand of the local node in the current (new) or old configuration and its primary subdomain is neighboring to that of the local node simultaneously. In this case, the primary sending block may have an intersection with the secondary receiving and alternative secondary receiving blocks which have already been scannded by `receive_particles()` and thus simply scaninng the primary sending blcok could cause duplicated transmissions. Therefore, the ranks of new and old helpands are given to `send_particles()` to avoid the duplication so that it skips `S_commlist` records having `region` elements maching to the ranks. This scan and transfer are performed always.

- If we were in secondary mode in the last simulation step and the local node had the helpand in the configuration in the step, the secondary sending block, which starts from `CommList` + `SLHeadTail`[1] + `SecSLHeadTail`[0] and whose size is `SecSLHeadTail`[1] − `SecSLHeadTail`[0] is scanned by `send_particles()` to send particles to the node responsible for the subdomain neighboring to the secondary subdomain of the local node in the configuration. We have to be aware of the possibility that the secondary sending block is given from the node which was the helpand of the local node and its primary subdomain is neighboring to that of the local node itself or of its helpand in the new configuration. In this case, the secondary sending block may have an intersection with the primary receiving or alternative secondary receiving block which have already been scanned by `receive_particles()` and thus simply scaninng the secondary sending blcok could cause duplicated transmissions. Therefore, the rank of the local node itself and that of new helpand are given to `send_particles()` to avoid the duplication so that it skips `S_commlist` records having `region` elements maching to the ranks.

The functions `receive_particles()` and `send_particles()` receive and send particles updating `RecvBufBases`$[p][s]$ to let it point to the recieve buffer $rbuf(p, s)/m$ for primary or secondary particles of species s to be received from the node m, and `SendBufDisps`$[s][k]$

to let it has the displacement of $sbuf(s,k)/m$ for particles of species s in the subdomain k to be sent to the node m. They also increment their last argument `req` to count the number of calls of `MPI_Irecv()` and `MPI_Isend()`, with which we confirm thier completions by `MPI_Waitall()` giving it the arrays of non-blocking transfer requests `Requests[]` and their completion stauses `Statuses[]`.

```
if (neighboring) {
  req = 0;
  receive_particles(CommList, SLHeadTail[0], &req);
  if (newparent>=0)
    receive_particles(secrlist, secrlsize, &req);
  if (oldparent!=newparent && oldparent>=0)
    receive_particles(CommList+SLHeadTail[1], SecSLHeadTail[0], &req);
  send_particles(CommList+SLHeadTail[0], SLHeadTail[1]-SLHeadTail[0],
                 newparent, oldparent, &req);
  if (oldparent>=0)
    send_particles(CommList+SLHeadTail[1]+SecSLHeadTail[0],
                   SecSLHeadTail[1]-SecSLHeadTail[0], me, newparent, &req);
  MPI_Waitall(req, Requests, Statuses);
}
```

On the other hand, if the argument `neighboring` is false to indicate that the accommodation mode is anywhere, we perform `MPI_Alltoallv()` for each $p \in \{0,1\}$ and $s \in [0, S-1]$ giving it the following arguments.

- `sendbuf` is always `SendBuf` regardless of p and s because we specify each of $sbuf(s,k)/m$ for the node m responsible for the subdomain k by `sdispls`$[m]$.
- `sendcounts` is `NOfSend`$[p][s][]$ which is set by `make_comm_count()` called from `try_stable2()` via `try_stable1()` or `rebalance2()` via `rebalance1()`.
- `sdispls` is `SendBufDisps`$[s][]$ for $sbuf(s,m)$ for each node m if $p = 0$. Otherwise, for the node m whose secondary subdomain is $k = parent(m) \geq 0$, `sdispls`$[m]$ = `TempArray`$[m]$ should have the following.

$$\texttt{TempArray}[m] = \texttt{SendBufDisps}[s][k] + \texttt{NOfSend}[0][s][k] + \sum_{\substack{i<m \\ parent(i)=k}} \texttt{NOfSend}[1][s][i]$$

  This means that $sbuf(s,k)$ for a subdomain k which has $h = |H(k)|$ helpers is split into $h+1$ consecutive sub-blocks $sbuf(s,k)/k$, $sbuf(s,k)/m_1, \ldots, sbuf(s,k)/m_h$ where $sbuf(s,k)/k$ has `NOfSend`$[0][s][k]$ particles while $sbuf(s,k)/m$ has `NOfSend`$[1][s][m]$ particles.
- `recvbuf` is `RecvBufBases`$[p][s]$ to point $rbuf(p,s)$.
- `recvcounts` is `NOfRecv`$[p][s][]$ which is set by `make_comm_count()` called from `try_stable2()` via `try_stable1()` or `rebalance2()` via `rebalance1()`.
- `rdispls` is `RecvBufDisps[]` whose element $[m]$ is defined as follows.

$$\texttt{RecvBufDisps}[m] = \sum_{i=0}^{m-1} \texttt{NOfRecv}[p][s][i]$$

- `sendtype` and `recvtype` are commmonly `T_Particle`.

```
    else {
      int ps;
      int *rcount=NOfRecv;
      int *scount=NOfSend;
      struct S_particle **rbb=RecvBufBases;
      for (ps=0; ps<2; ps++,rbb+=ns) {            /* rbb=&RecvBufBases[p][0] */
        int *sbd0=SendBufDisps, *sbd;
        for (s=0; s<ns; s++,rcount+=nn,scount+=nn,sbd0+=nn) {
                                                  /* rcount=&NOfRecv[ps][s][0] */
                                                  /* sbd0=&SendBufDisps[s][0] */
          int rdisp=0;
          for (i=0; i<nn; i++) {
            RecvBufDisps[i] = rdisp;  rdisp += rcount[i];
          }
          if (ps==0) sbd = sbd0;                  /* &SendBufDisps[s][0] */
          else {
            sbd = TempArray;
            for (i=0; i<nn; i++) {
              int r=Nodes[i].parentid;
              if (r>=0) {
                sbd[i] = sbd0[r];
                sbd0[r] += scount[i];
              }
              else sbd[i] = 0;            /* not necessary becasuse scount[i]=0
                                             but ... */
            }
          }
          MPI_Alltoallv(SendBuf, scount, sbd, T_Particle,
                        rbb[s], rcount, RecvBufDisps, T_Particle, MCW);
          if (ps==0)
            for (i=0; i<nn; i++) sbd0[i] += scount[i];
        }
      }
    }
  }
```

4.5.13 `move_to_sendbuf_uw()`

`move_to_sendbuf_uw()` The function `move_to_sendbuf_uw()`, called from `move_to_sendbuf_primary()` and `move_to_sendbuf_secondary()`, scans primary ($p = 0$) or secondary ($p = 1$) particles in `Particles[]` to move its contents to `SendBuf` or to pack them in `Particles[]` itself. The function is given the following arguments according to the caller's context defined by p.

- `ps` $= p$ is used as the argument of `Subdomain_Id()` to extract the subdomain identifier of a particle when it is in a neighbor of the local node's primary/secondary subdomain.
- `me` $= n'$ is the rank of the local node n if $p = 0$, while it is $parent(n)$ otherwise. It is used to identify particles which will be reside in the local node and thus is to be moved in `Particles[]` by packing operation. Note that $n' = parent(n)$ can be -1 meaning that all secondary particles should be moved to `SendBuf[]` because the local node will not have its helpand in the next step.

- `putmes` is the slice `NOfPLocal`$[p][S][n']$ whose element $[p][s][n']$ has the number of primary ($p = 0$) or secondary ($p = 1$) particles residing in the primary/secondary subdomain but being moved to `SendBuf`[] and sent to other nodes in the primary/ secondary family of the local node.
- `cbase` is 0 if $p = 0$ or `primaryParts` otherwise, to specify the starting point of the scan, i.e., $pbuf(p, 0)$.
- `ctp` is `TotalP`$[p][S]$ to show the size of the current $pbuf(p, s)$ is `ctp`$[s]$.
- `nbase` is 0 if $p = 0$ or Q_n^n otherwise, to specify the head of $pbuf(p, 0)$ for the next step. That is, particles staying in the local node is packed to the region from `nbase`.
- `ntp` is `TotalPNext`$[p][S]$ to show the size of the next $pbuf(0, s)$ is `ntp`$[s]$.
- `rbb` is `RecvBufBases`$[p][S]$ to specify that the head of $rbuf(p, s)$ should be set into `rbb`$[s]$.

```
static void
move_to_sendbuf_uw(int ps, int me, int *putmes, int cbase, int *ctp,
                   int nbase, int *ntp, struct S_particle **rbb) {
  int i, in, j, jn, k, s;
  int ns=nOfSpecies, nn=nOfNodes, *sbd=SendBufDisps;
  Decl_Grid_Info();
```

The function moves particles in $pbuf(p, s)$, by scanning them in the ascending order of s, as follows. Let i and j be the followings being the head of the current and next $pbuf(p, s)$ respectively.

$$i = \texttt{cbase} + \sum_{t=0}^{s-1} \texttt{ctp}[t] = \texttt{cbase} + \sum_{t=0}^{s-1} \texttt{TotalP}[p][t]$$

$$j = \texttt{nbase} + \sum_{t=0}^{s-1} \texttt{ntp}[t] = \texttt{nbase} + \sum_{t=0}^{s-1} \texttt{TotalPNext}[p][t]$$

1. If $j \leq i$, all the particles staying in $pbuf(p, s)$ can be moved *upward*, i.e., toward smaller locations from their current locations, because the number of particles to be moved is at most `ctp`$[s]$. In this case, we move particles in `Particles`$[i+k]$ for all $k \in [0, \texttt{ctp}[s])$ as follows.
   (a) If the subdomain identifier m of `Particles`$[i+k]$ obtained by `Subdomain_Id()` is not equal to n', i.e., the particle is not in the subdomain specified by n' but in m, it is moved to `SendBuf`[`SendBufDisps`$[s][m]$] post-incrementing `SendBufDisps`$[s][m]$. Note that m can be -1 to mean the particle disappears from the simulation domain and thus we simply discard it. Also note that n' can be -1 to mean all particles in the subdomain $m \geq 0$ are unconditionally moved to `SendBuf`[].
   (b) Otherwise, if `putmes` $\neq$ `NULL` and $k <$ `putmes`$[s][0]$ to mean the particle `Particles`$[i+k]$ is in the leading region to be sent to a family member, the particle is moved to `SendBuf`[] as done in (a).

(c) Otherwise, i.e., after the movement of the particles done by (b), the reminders in the subdomain n' are moved upward to the next $pbuf(p,s)$ starting from `Particles`$[j]$.

Finally, we set `rbb`$[s]$ = `RecvBufBases`$[p][s]$ to point `Particles`$[j+l]$ where l is the number of particles staying in this block $pbuf(p,s)$ so that particles from other nodes are received to the bottom of the block.

```
for (s=0,i=cbase,j=nbase,k=0; s<ns; s++,i=in,j=jn,sbd+=nn,k+=nn) {
  int putme = putmes ? putmes[k] : 0; /* NOfPLocal[0/1][s][me/sec] */
  in = i + ctp[s];  jn = j + ntp[s];
  if (j<=i) {                         /* upward move only */
    for (; putme>0; i++) {            /* throw my particles to send buf */
      int dst=Subdomain_Id(Particles[i].nid, ps);
      if (dst<0) continue;
      SendBuf[sbd[dst]++] = Particles[i];
      if (dst==me) putme--;
    }
    for (; i<in; i++) {               /* move upward */
      int dst=Subdomain_Id(Particles[i].nid, ps);
      if (dst<0) continue;
      if (dst==me) Particles[j++] = Particles[i];
      else         SendBuf[sbd[dst]++] = Particles[i];
    }
    rbb[s] = Particles + j;           /* receive to bottom */
```

2. If $j > i$ and $j + $ `ntp`$[s] > i + $ `ctp`$[s]$, all the particles to stay in $pbuf(p,s)$ can be moved *downward* by a *bottom-up* scan of the block. Since this movement must be performed after the movement of the succeeding blocks, we leave it to the counterpart function `move_to_sendbuf_dw()`. Note that `rbb`$[s]$ = `RecvBufBases`$[p][s]$ is set to point `Particles`$[j]$ so that particles from other nodes are received to the top of the block because particles staying are packed to the bottom.

```
  } else if (jn>in) {                 /* downward only and thus skip */
    rbb[s] = Particles + j;           /* receive to top */
```

3. Otherwise, i.e., $j > i$ but $j + $ `ntp`$[s] \leq i + $ `ctp`$[s]$, the upper half of the block must be moved downward while the lower half have to be moved upward. Therefore, after we move particles as done in 1(a) and 1(b) and record the source location i_b of the first succeeding particle, we skip particles which should move downward (if any still), i.e., those staying in the subdomain $n' \geq 0$[49], recording the source and destination locations, namely i_m and j_m, of the last particle skipped. Then we move the remaining l particles upward in the way of 1(a) and 1(c), and set `rbb`$[s]$ = `RecvBufBases`$[p][s]$ to point `Particles`$[j_m + 1 + l]$ to receive particles from other nodes at the bottom of $pbuf(p,s)$. Finally, we move skipped particles `Particles`$[k]$ for all k such that $i_b \leq k \leq i_m$ (if any, i.e., $i_b \leq i_m$) downward scanning them descendingly from `Particles`$[i_m]$ in the way of 1(a) and 1(c) to the subblock whose tail is `Particles`$[j_m]$.

[49] We need to check the subdomain identifier obtained by `Subdomain_Id()` is non-negative because n' can be -1.

```
    } else {                            /* downward and upward */
      int ib, im, jm;
      for (; putme>0; i++) {            /* throw my particles to send buf */
        int dst=Subdomain_Id(Particles[i].nid, ps);
        if (dst<0) continue;
        SendBuf[sbd[dst]++] = Particles[i];
        if (dst==me) putme--;
      }
      ib = i;
      for (; i<j; i++) {                 /* skip downward movers if any */
        int dst=Subdomain_Id(Particles[i].nid, ps);
        if (dst==me && dst>=0)  j++;
      }
      im = i-1; jm = j-1;
      for (; i<in; i++) {               /* move remainders upward */
        int dst=Subdomain_Id(Particles[i].nid, ps);
        if (dst<0) continue;
        if (dst==me) Particles[j++] = Particles[i];
        else         SendBuf[sbd[dst]++] = Particles[i];
      }
      rbb[s] = Particles + j;           /* receive to bottom */
      for (i=im,j=jm; i>=ib; i--) {     /* move first half downward if any */
        int dst=Subdomain_Id(Particles[i].nid, ps);
        if (dst<0) continue;
        if (dst==me) Particles[j--] = Particles[i];
        else         SendBuf[sbd[dst]++] = Particles[i];
      }
    }
  }
}
```

4.5.14 `move_to_sendbuf_dw()`

`move_to_sendbuf_dw()` The function `move_to_sendbuf_dw()`, called from `move_to_sendbuf_primary()` and `move_to_sendbuf_secondary()`, scans primary ($p = 0$) or secondary ($p = 1$) particles in `Particles[]` to move its contents to `SendBuf` or to pack them in `Particles[]` itself. The scanning and moving are performed only on the blocks $pbuf(p, s)$ which are skipped by the upward counterpart `move_to_sendbuf_uw()` because they are shifted down as a whole. The function is given the following arguments according to the caller's context defined by p.

- `ps`, `me`, `putmes`, `ctp` and `ntp` are equivalent to those for `move_to_sendbuf_uw()`.
- `ctail` is `primaryParts` if $p = 0$ or `totalParts` otherwise, to specify the starting point of the scan, i.e., the tail (plus one) of $pbuf(p, S-1)$.
- `ntail` is Q_n^n if $p = 0$ or $Q_n^n + Q_n^{parent(n)}$ otherwise, to specify the tail (plus one) of $pbuf(p, S-1)$ in the next step.

```
static void
move_to_sendbuf_dw(int ps, int me, int *putmes, int ctail, int *ctp, int ntail,
```

```
                    int *ntp) {
  int i, in, j, jn, k, s, ns=nOfSpecies, nn=nOfNodes, nnnsm1=nn*(ns-1);
  int *sbd=SendBufDisps+nnnsm1;
  Decl_Grid_Info();
```

The function moves particles in $pbuf(p,s)$, by scanning them in the descending order of s, as follows. Let i and j be the followings being the tails of the current and next $pbuf(p,s)$ respectively.

$$i = \texttt{ctail} - 1 - \sum_{t=s}^{S-1} \texttt{ctp}[t] = \texttt{ctail} - 1 - \sum_{t=s}^{S-1} \texttt{TotalP}[p][t]$$

$$j = \texttt{ntail} - 1 - \sum_{t=s}^{S-1} \texttt{ntp}[t] = \texttt{ntail} - 1 - \sum_{t=s}^{S-1} \texttt{TotalPNext}[p][t]$$

If $i < j$ and $i - \texttt{ctp}[t] < j - \texttt{ntp[s]}$, the particles in the block $pbuf(p,s)$ are scanned and moved in the way shown in 1(a)–(c) in §4.5.13.

```
  in = ctail;  jn = ntail;
  for (s=ns-1,i=in-1,j=jn-1,k=nnnsm1; s>=0; s--,i=in-1,j=jn-1,sbd-=nn,k-=nn) {
    int putme = putmes ? putmes[k] : 0; /* NOfPLocal[0/1][s][me/sec] */
    in -= ctp[s];  jn -= ntp[s];
    if (i>=j || in>=jn) continue;       /* not downward only and thus skip */
    for (; putme>0; i--) {              /* throw my particles to send buf */
      int dst=Subdomain_Id(Particles[i].nid, ps);
      if (dst<0) continue;
      SendBuf[sbd[dst]++] = Particles[i];
      if (dst==me) putme--;
    }
    for (; i>=in; i--) {                /* move downward */
      int dst=Subdomain_Id(Particles[i].nid, ps);
      if (dst<0) continue;
      if (dst==me) Particles[j--] = Particles[i];
      else         SendBuf[sbd[dst]++] = Particles[i];
    }
  }
}
```

4.5.15 move_injected_to_sendbuf()

move_injected_to_sendbuf()

The function move_injected_to_sendbuf(), called from move_to_sendbuf_primary() and move_to_sendbuf_secondary(), moves particles injected by oh2_inject_particle() from the tail of Particles[] to SendBuf[]. It scans the block for the injected particles which starts from Particles[totalParts] and has nOfInjections $= Q_n^{\rm inj}$ particles. Each particle whose nid, or its part for subdmain identifier extracted by Subdomain_Id(), and spec are m and s_0 respectively is moved to $sbuf(s,m)$ where s is given by Particle_Spec() as follows, if nid is non-negative.

$$s = \begin{cases} s_0 - \texttt{specBase} & \texttt{OH\_HAS\_SPEC} \text{ is defined} \\ 0 & \text{otherwise} \end{cases}$$

That is, if S_particle has the element spec, s is the element offset by specBase which is 0 if the library is initialized by oh2_init() called from a C-coded simulator, while

1 if initialized by `oh2_init_()` called from a Fortran-coded simulator. The location in $sbuf(s,m)$ to which a particle is moved is given by `SendBufDisps`$[s][m]$ which was set to the head of $sbuf(s,m)$ by `set_sendbuf_disps()` prior to the call of this function. Then it is post-incremented at each move to make it point the head of the buffer for ordinary not-injected particles to be sent to the other nodes.

Note that this function moves particles injected into primary/secondary subdomain of the local node to `SendBuf[]`. These particles or part of them, however, will be moved back to $rbuf(p,s)$ in `Particles[]` by `move_injected_from_sendbuf()`. Also note that the subdomain identifier of a particle injected into a subdomain m can have $(N+3^D)+d$ or $(N+3^D)+m+3^D$ if the subdomain is secondary one of the local node or is its d-th neighbor, if `OH_POS_AWARE` is defined to mean we employ position-aware particle management. If so, we let the identifier be m by the macro `Primarize_Id()`.

```
static void
move_injected_to_sendbuf() {
  struct S_particle *pbuf=Particles+totalParts;
  int ninj=nOfInjections, nn=nOfNodes, sb=specBase;
  int i;
  Decl_Grid_Info();

  for (i=0; i<ninj; i++) {
    int dst = Subdomain_Id(pbuf[i].nid, 0);
    int s = Particle_Spec(pbuf[i].spec-sb);
    if (dst<0) continue;
#ifdef OH_POS_AWARE
    if (dst>=nn)  Primarize_Id(pbuf+i, dst);
#endif
    SendBuf[SendBufDisps[dst+s*nn]++] = pbuf[i];
  }
}
```

4.5.16 `move_injected_from_sendbuf()`

`move_injected_from_sendbuf()`

The function `move_injected_from_sendbuf()`, called from `move_to_sendbuf_primary()` and `move_to_sendbuf_secondary()`, moves particles injected by `oh2_inject_particle()` or its higher level counterparts and then temporarily moved to `SendBuf[]` by `move_injected_to_sendbuf()` back to `Particles[]`. The function has three arguments the first of which is an array `injected`$[S]$ = `InjectedParticles`$[\pi][p][S]$ whose element $[s]=q_s$ is the number of primary ($p=0$) or secondary ($p=1$) particles of species s to be moved back, where $\pi=0$ if called from `move_to_sendbuf_primary()` while $\pi=1$ if from `move_to_sendbuf_secondary()`, from the leading part of $sbuf(s,m)$ to that of $rbuf(p,s)$ where $m=n$ for the local node n if $p=0$ or $m=parent(n)$ if $p=1$ and m is given through its argument `mysd`. Therefore, for each $s\in[0,S)$, we move particles `SendBuf`$[$`SendBufDisps`$[s][m]+k]$ to the location pointed by `RecvBufBases`$[p][s]+k$ for all $k\in[0,q_s)$, where `RecvBufBases`$[p]$ is given through the argument `rbb`. Then we increment `SendBufDisps`$[s][m]$ and `RecvBufBases`$[p][s]$ by q_s so that they point the heads of send/receive buffers for particles to be sent/received.

```
static void
move_injected_from_sendbuf(int *injected, int mysd, struct S_particle **rbb) {
```

```
    int nn=nOfNodes, ns=nOfSpecies;
    int *sdisp=SendBufDisps+mysd;
    int s, i;

    for (s=0; s<ns; s++,sdisp+=nn) {
      struct S_particle *rbuf=rbb[s];
      struct S_particle *sbuf=SendBuf+*sdisp;
      int inj=injected[s];
      for (i=0; i<inj; i++)  rbuf[i] = sbuf[i];
      rbb[s] += inj;  *sdisp += inj;
    }
  }
```

4.5.17 receive_particles()

receive_particles()

The function receive_particles(), called solely from exchange_particles(), scans the S_commlist sequence, the primary or (alternative) secondary receiving block of CommList, whose head and size are given through the arguments rlist and rlsize. It posts MPI_Irecv() and MPI_Isend() each time a record for receiving/sending to/from the local node n is found. The arguments count, source, dest and tag for these MPI functions are simply given by the CommList record, and type and comm are obvious and invariant, but buf is a little bit complicated.

For MPI_Irecv(), buf for the k-th record with tag $= pS + s$ starts from $q_r(p, s, k)$-th particle in the buffer whose head is pointed by RecvBufBases$[p][s]$ when the function is called, where $q_r(p, s, k)$ is defined as follows.

$$\mathcal{C}_r(p, s, k) = \{i \mid i < k,\ \texttt{rlist}[i].(\texttt{rid}, \texttt{tag}) = (n,\ pS + s)\}$$
$$q_r(p, s, k) = \sum_{i \in \mathcal{C}_r(p,s,k)} \texttt{rlist}[i].\texttt{count}$$

Therefore each time we find a record with rid $= n$, we increment RecvBufBases$[p][s]$ = RecvBufBases[tag] by count of the record after letting buf be RecvBufBases$[p][s]$.

For MPI_Isend(), buf for the k-th record with tag $= pS + s$ and region $= m$ starts from $q_s(s, m, k)$-th particle in the buffer whose head's displacement of SendBuf[] is SendBufDisps$[s][m]$ = SendBufDisps$[sN+m]$ when the function is called, where $q_s(s, m, k)$ is defined as follows.

$$\mathcal{C}_s(s, m, k) = \{i \mid i < k,\ \texttt{rlist}[i].(\texttt{sid}, \texttt{tag}, \texttt{region}) = (n,\ pS + s,\ m),\ p \in \{0, 1\}\}$$
$$q_s(s, m, k) = \sum_{i \in \mathcal{C}_s(s,m,k)} \texttt{rlist}[i].\texttt{count}$$

Therefore each time we find a record with sid $= n$, we increment SendBufDisps$[s][m]$ by count of the record after letting buf be SendBuf + SendBufDisps$[s][m]$.

We also give the MPI functions the pointer to Requests$[r]$ to let them store an MPI_Request structure in it, where r's initial value is given through the argument req of this function, r is incremented each time the MPI functions are called, and then r's final value is returned to the caller through req to be used for the successive calls of this function and send_particles().

```
static void
```

```
receive_particles(struct S_commlist *rlist, int rlsize, int *req) {
  int me=myRank, i, r=*req, nn=nOfNodes, ns=nOfSpecies, sdisp;
  struct S_particle *rbuf;

  for (i=0; i<rlsize; i++) {
    if (rlist[i].rid==me) {
      int count=rlist[i].count, tag=rlist[i].tag;
      rbuf = RecvBufBases[tag];  RecvBufBases[tag] = rbuf + count;
      MPI_Irecv(rbuf, count, T_Particle, rlist[i].sid, tag, MCW,
                Requests+(r++));
    }
    if (rlist[i].sid==me) {
      int count=rlist[i].count, tag=rlist[i].tag, region=rlist[i].region;
      region += nn * (tag<ns ? tag : tag-ns);
      sdisp = SendBufDisps[region];  SendBufDisps[region] = sdisp + count;
                                                  /* SendBufDisps[s][region] */
      MPI_Isend(SendBuf+sdisp, count, T_Particle, rlist[i].rid, tag, MCW,
                Requests+(r++));
    }
  }
  *req = r;
}
```

4.5.18 send_particles()

send_particles()

The function send_particles(), called solely from exchange_particles(), scans the S_commlist sequence, the primary or secondary sending block of CommList, whose head and size are given through the arguments slist and slsize. It posts MPI_Isend() for records for sending from the local node, giving it arguments in the same way as receive_particles() does for sending records to send particles from buffers in SendBuf[] whose displacement is specified by SendBufDisps[][]. However, the records to be processed are not just those having sid $= n$ for the local node n, but those having region matching to myregion or parentregion argument are excluded from the processing. That is, as explained in §4.5.12, since such a record should have been already processed by receive_particles() because it should be in a receiving block, we have to exclude it to avoid duplicated transmission. The function also has an argument req to receive/return the entry number of Requests[] for MPI_Request structures also as done in receive_particles().

```
static void
send_particles(struct S_commlist *slist, int slsize, int myregion,
               int parentregion, int *req) {
  int me=myRank, i, r=*req, nn=nOfNodes, ns=nOfSpecies, sdisp, region;

  for (i=0; i<slsize; i++) {
    if (slist[i].sid==me && (region=slist[i].region)!=myregion &&
                            region != parentregion) {
      int count=slist[i].count, tag=slist[i].tag;
      region += nn * (tag<ns ? tag : tag-ns);
      sdisp = SendBufDisps[region];  SendBufDisps[region] = sdisp + count;
                                                  /* SendBufDisps[s][region] */
      MPI_Isend(SendBuf+sdisp, count, T_Particle, slist[i].rid, tag, MCW,
                Requests+(r++));
```

```
    }
  }
  *req = r;
}
```

4.5.19 oh2_inject_particle()

oh2_inject_particle_()
oh2_inject_particle()

The API function `oh2_inject_particle_()` for Fortran and `oh2_inject_particle()` for C provide a simulator body calling them with the way to inject a particle the pointer to which is given by the argument `part`. Before it moves the particle to the tail of the particle buffer, namely `Particles`$[i]$ where $i =$ `totalParts`$+$`nOfInjections` $= Q_n + Q_n^{\rm inj}$, it checks if `OH_HAS_SPEC` is defined or $S = 1$ to mean `Particle_Spec()` correctly gives the spec s of the particle from its `spec` element offset by `specBase` or $s = 0$ unconditionaly because of $S = 1$, and abort the execution by `local_errstop()` if not satisfied. It also checks if $i < P_{lim} =$ `nOfLocalPLimit` whose violation also causes abort by `local_errstop()` due to the overflow of `Particles`[].

Then the function moves the particle of `nid` $= m$ to `Particles`$[i]$ incrementing $Q_n^{\rm inj}$ to show the number of injections as well as the entry for the next injection. After that, if $m = parent(n)$ for local node n, it increments `NOfPLocal`$[1][s][m]$ to incorporate the injected particle to the secondary particle population histogram, and also increments `InjectedParticles`$[0][1][s] = q^{\rm inj}(n)[1][s]$ to count the number of particles injected into n's secondary subdomain. Otherwise, the injected particle is regarded as primary, and thus `NOfPLocal`$[0][s][m]$ is incremented if $m \geq 0$. Then, if $m = n$ for the local node n, it increments `InjectedParticles`$[0][0][s] = q^{\rm inj}(n)[0][s]$ to count the number of particles injected into n's primary subdomain[50].

```
void
oh2_inject_particle_(struct S_particle *part) {
  oh2_inject_particle(part);
}
void
oh2_inject_particle(struct S_particle *part) {
  const int ns=nOfSpecies, nn=nOfNodes;
  int inj = totalParts + nOfInjections++;
  int s = Particle_Spec(part->spec - specBase);
  int n=part->nid;

#ifndef OH_HAS_SPEC
  if (ns!=1)
    local_errstop("particles cannot be injected when S_particle does not "
                  "have 'spec' element and you have two or more species");
#endif
  if (inj>=nOfLocalPLimit)
    local_errstop("injection causes local particle buffer overflow");
  Particles[inj] = *part;
  if (n<0)  return;
  if (n==RegionId[1]) {
```

[50] Note that the particle residence subdomain is just `part->nid` instead of that with `Subdomain_Id()` because we have other function for particle injection with position-aware particle management. Also note that regarding particles injected into secondary subdomain as secondary should work almost perfectly well unless a particle is injected at the boundary of secondary subdomain.

```
    NOfPLocal[(ns+s)*nn+n]++;
    InjectedParticles[ns+s]++;
  } else {
    NOfPLocal[nn*s+n]++;
    if (n==myRank)  InjectedParticles[s]++;
  }
}
```

4.5.20 oh2_remap_injected_particle()

oh2_remap_injected_particle_()
oh2_remap_injected_particle()

The API function oh2_remap_injected_particle_() for Fortran and oh2_remap_injected_particle() for C maintain NOfPLocal$[p][s][m]$ and InjectedParticles$[0][p][s]$ of the particle π pointed by the sole argument part, which has been injected by oh2_inject_particle() with negative nid element or has been removed by oh2_remove_injected_particle(), where $m = \pi$.nid, s is the species of π obtained by Particle_Spec(), and $p \in \{0, 1\}$ is 1 iff $m = parent(n)$ for the local node n.

At first the function checks if

$$\texttt{Particles} + Q_n \le \&\pi < \texttt{Particles} + Q_n + Q_n^{\mathrm{inj}}$$

i.e., π in at a location for injected particles, and aborts the execution by local_errstop() if unsatisfied. Then we do the following as done in oh2_inject_particle(); check if OH_HAS_SPEC is defined or $S = 1$ and abort the execution if both are unsatisfied; increment NOfPLocal$[p][s][m]$ if $m \ge 0$; and increment InjectedParticles$[0][p][s]$ if $m \in \{n, parent(n)\}$.

```
void
oh2_remap_injected_particle_(struct S_particle *part) {
  oh2_remap_injected_particle(part);
}
void
oh2_remap_injected_particle(struct S_particle *part) {
  const int pidx = part - Particles, ns=nOfSpecies, nn=nOfNodes;
  int s, n;

  if (pidx<totalParts || pidx>=totalParts+nOfInjections)
    local_errstop("'part' argument pointing %c%d%c of the particle buffer is "\
                  "not for injected particles",
                  specBase?'(':'[', pidx+specBase, specBase?')':']');
#ifndef OH_HAS_SPEC
  if (ns!=1)
    local_errstop("particles cannot be injected when S_particle does not "
                  "have 'spec' element and you have two or more species");
#endif
  s = Particle_Spec(part->spec - specBase);
  n = part->nid;
  if (n<0)  return;
  if (n==RegionId[1]) {
    NOfPLocal[(ns+s)*nn+n]++;
    InjectedParticles[ns+s]++;
  } else {
    NOfPLocal[nn*s+n]++;
```

```
    if (n==myRank)  InjectedParticles[s]++;
  }
}
```

4.5.21 oh2_remove_injected_particle()

oh2_remove_injected_particle_()
oh2_remove_injected_particle()

The API function `oh2_remove_injected_particle_()` for Fortran and `oh2_remove_injected_particle()` for C remove the particle π pointed by the sole argument `part` which has been injected by `oh2_inject_particle()`, maintaining `NOfPLocal`$[p][s][m]$ and `InjectedParticles`$[0][p][s]$ of the particle pointed by `part` $= \pi$, where $m = \pi$.`nid`, s is the species of π obtained by `Particle_Spec()`, and $p \in \{0, 1\}$ is 1 iff $m = parent(n)$ for the local node n.

The functions performs the following similarly to `oh2_remap_injected_particle()` but opposite way in the maintaiance of `NOfPLocal`[][][] and `InjectedParticles`$[0][][]$; check if π is at a location for injected particles, and aborts the execution by `local_errstop()` if unsatisfied; check if `OH_HAS_SPEC` is defined or $S = 1$ and abort the execution if both are unsatisfied; decrement `NOfPLocal`$[p][s][m]$ if $m \geq 0$; and decrement `InjectedParticles`$[0][p][s]$ if $m \in \{n, parent(n)\}$. Finally, π.`nid` is let be -1 for removal.

```
void
oh2_remove_injected_particle_(struct S_particle *part) {
  oh2_remove_injected_particle(part);
}
void
oh2_remove_injected_particle(struct S_particle *part) {
  const int pidx = part - Particles, ns=nOfSpecies, nn=nOfNodes;
  int s, n;

  if (pidx<totalParts || pidx>=totalParts+nOfInjections)
    local_errstop("'part' argument pointing %c%d%c of the particle buffer is "\
                  "not for injected particles",
                  specBase?'(':'[', pidx+specBase, specBase?')':']');
#ifndef OH_HAS_SPEC
  if (ns!=1)
    local_errstop("particles cannot be injected when S_particle does not "
                  "have 'spec' element and you have two or more species");
#endif
  s = Particle_Spec(part->spec - specBase);
  n = part->nid;
  if (n<0)  return;
  if (n==RegionId[1]) {
    NOfPLocal[(ns+s)*nn+n]--;
    InjectedParticles[ns+s]--;
  } else {
    NOfPLocal[nn*s+n]--;
    if (n==myRank)  InjectedParticles[s]--;
  }
  part->nid = -1;
}
```

4.5.22 oh2_set_total_particles()

oh2_set_total_particles_()
oh2_set_total_particles()

The API function oh2_set_total_particles_() for Fortran and oh2_set_total_particles() for C tell the library that the simulator body has initialized Particles[] and NOfPLocal[][][] but it will modify them for particle injection/removal before the first call of oh2_transbound(). This means that the library should grasp the layout in Particles[] through NOfPLocal[][][] to initialize substance variables TotalP[][], primaryParts and totalParts. Since this initialization is usually done in the first call of transbound1() by set_total_particles(), the API functions simply call it.

```
void
oh2_set_total_particles_() {
  set_total_particles();
}
void
oh2_set_total_particles() {
  set_total_particles();
}
```

4.5.23 oh2_max_local_particles()

oh2_max_local_particles_()
oh2_max_local_particles()

The API function oh2_max_local_particles_() for Fortran and oh2_max_local_particles() for C calculates the maximum number of particles a local node can accommodate and returns it to a simulator body calling them. The function takes the following arguments.

- npmax = P^G_{lim} is the absolute maximum number of particles which the simulator is capable of as a whole.
- maxfrac = α is the tolerance factor percentage of load imbalance and should be same as the argument maxfrac of oh2_init().
- minmargin = Δ is the minimum margin by which the return value P_{lim} has to clear over the per node average of npmax.

Prior to calculating P_{lim} by;

$$\overline{P} = \lceil P^G_{lim}/N \rceil \qquad P_{lim} = \max(\lceil \overline{P}(100+\alpha)/100 \rceil\,,\ \overline{P} + \Delta)$$

the function obtains N by MPI_Comm_size() and confirms $P^G_{lim} > 0$ and $0 < \alpha \leq 100$ are satisfied or aborts the execution by errstop(). It also confirms $P_{lim} \leq$ INT_MAX or aborts the execution by mem_alloc_error().

```
int
oh2_max_local_particles_(dint *npmax, int *maxfrac, int *minmargin) {
  return(oh2_max_local_particles(*npmax, *maxfrac, *minmargin));
}
int
oh2_max_local_particles(dint npmax, int maxfrac, int minmargin) {
  int nn, nplint;
  dint npl, npmargin;

  MPI_Comm_size(MCW, &nn);
```

```
  if (npmax<=0) errstop("max # of particles should be greater than 0");
  if (maxfrac<=0 || maxfrac>100)
    errstop("load imbalance factor (%d) should be in the range [1..100]",
            maxfrac);
  npl = (npmax-1)/nn + 1; /* ceil(npmax/nn) */
  npmargin = (npl*maxfrac-1)/100 + 1;
  npl += (npmargin<minmargin) ? minmargin : npmargin;
  if (npl>INT_MAX) mem_alloc_error("Particles", 0);
  nplint = npl;
  return(nplint);
}
```

4.6 Header File ohhelp3.h

The header file of level-3 library, ohhelp3.h, declares global variables and their `structures` used in level-3 library codes to keep the configurations of the domain, subdomains and field-arrays, and gives prototypes of API functions and those called by higher level codes.

4.6.1 Control Variable

`excludeLevel2` At first we declare the integer variable `excludeLevel2` being true (1) if the level-3 library is initialized by `oh13_init()`, or false (0) otherwise, i.e., `oh3_init()` was called for initialization. It is set by `init3()` and is referred to by `transbound3()` to decide which of `transbound1()` (if true) or `transbound2()` (if false) is called.

```
EXTERN int excludeLevel2;
```

4.6.2 Domain and Subdomain Descriptors

`OH_LOWER` `OH_UPPER` Prior to the declaration of domain and subdomain variables, we declare two constants `OH_LOWER` = 0 and `OH_UPPER` = 1 to specify indices of an array dimension for lower and upper bounds of a subdomain.

```
#define OH_LOWER 0
#define OH_UPPER 1
```

Next we declare the following three variables to specify the domain and subdomains.

`SubDomains`

- The integer array `SubDomains`$[N][D][2]$ is the substance of the array pointed by `sdoms` argument of `oh3_init()`. Its element $[m][d][\beta] = \{\delta_d^l(m), \delta_d^u(m)\}$ has the d-th dimensional lower ($\beta = 0$) or upper ($\beta = 1$) *boundary coordinate*, which is the d-th dimensional coordinate of the $(D-1)$-dimensional *boundary plane* perpendicular to d-th axis, of the subdomain m. That is, a subdomain m is a cuboid defined as follows.

$$[\delta_0^l(m), \delta_0^u(m)) \times \cdots \times [\delta_{D-1}^l(m), \delta_{D-1}^u(m))$$

  The array is allocated and initialized by `init3()` with values given through its argument `sdoms` or given by `init_subdomain_actively()`. It (or its shadow pointed by `sdoms`) is referred to by `init_subdomain_passively()`, `init_fields()`, `set_field_descriptors()`, `set_border_exchange()` and `transbound3()`.

`SubDomainsFloat`

- The floating-point array `SubDomainsFloat`$[N][D][2]$ is the floating-point and *grid-size-aware* counterpart of `SubDomains`$[][][]$ to have $\delta_d^\beta(m) \cdot \gamma_d$ in its element $[m][d][\beta]$, where γ_d = `Grid`$[d]$`.gsize` is the d-th dimensional *grid size* given by `oh3_grid_size()` or 1 if the function is not invoked.

  The array is allocated and initialized by `init3()` with values given through its argument `sdoms` or given by `init_subdomain_actively()`, then may be updated by `oh3_grid_size()`, and is referred to by `oh3_map_particle_to_neighbor()` through the macro `Map_Particle_To_Neighbor()` and `Adjust_Subdomain()`.

S_grid
Grid

- The array `Grid`[3] of the `S_grid` structure with the following elements has the process/grid coordinate information of the d-th axis of the system domain in its element $[d] \in [0, 2]$.
  - `n` is the number of processes Π_d ranked along the d-th axis if the process coordinate is *regular*, i.e., the coordinate is generated by the library using `init_subdomain_actively()`. Otherwise, i.e. if *irregular* process coordinate is given by the simulator body through the `sdoms` argument of `oh3_init()`, it has 0 but is never referred to.
  - `coord`$[\beta] = \{\Delta_d^l, \Delta_d^u\}$ has the d-th dimensional lower ($\beta = 0$) or upper ($\beta = 1$) boundary coordinate of the system domain. More specifically, it has the minimum/maximum d-th dimensional boundary coordinate of subdomains, i.e.;

$$\texttt{Grid}[d]\texttt{.coord}[0] = \Delta_d^l = \min_{0 \le m < N}\{\delta_d^l(m)\}$$
$$\texttt{Grid}[d]\texttt{.coord}[1] = \Delta_d^u = \max_{0 \le m < N}\{\delta_d^u(m)\}$$

  - `fcoord`$[\beta]$ is the grid-size-aware counterpart of `coord`$[\beta]$ to have $\Delta_d^\beta \cdot \gamma_d$.
  - `size` is the maximum length (number of grid points) of subdomain edges parallel to the d-th axis, or d-th dimensional *subdomain size* in short, i.e.;

$$\texttt{Grid}[d]\texttt{.size} = \delta_d^{\max} = \max_{0 \le m < N}\{\delta_d^u(m) - \delta_d^l(m)\}$$

    Therefore, if the process coordinate is regular, it is $\lceil(\Delta_d^u - \Delta_d^l)/\Pi_d\rceil$.
  - `fsize` is the grid-size-aware counterpart of `size` to have $\delta_d^{\max} \cdot \gamma_d$.
  - `gsize` has the grid size γ_d.
  - `rgsize` has the reciprocal of grid size $1/\gamma_d$.
  - `light.size` is the minimum d-th dimensional subdomain size, i.e.;

$$\texttt{Grid}[d]\texttt{.light.size} = \delta_d^{\min} = \min_{0 \le m < N}\{\delta_d^u(m) - \delta_d^l(m)\}$$

    Therefore, if the process coordinate is regular, it is $\lfloor(\Delta_d^u - \Delta_d^l)/\Pi_d\rfloor$.
  - `light.rfsize` is the grid-size-aware counterpart of `light.size` but has its reciprocal, $1/(\delta_d^{\min} \cdot \gamma_d)$.
  - `light.rfsizeplus` is the grid-size-aware and reciprocal counterpart of `fsize` to have $1/((\delta_d^{\min}+1) \cdot \gamma_d)$ if regular process coordinate.
  - `light.n` is the number of processes along d-th axis whose d-th dimensional subdomain size is `light.size` if regular process coordinate, i.e.;

$$\texttt{Grid}[d]\texttt{.light.n} = \Pi_d^- = \Pi_d - ((\Delta_d^u - \Delta_d^l) \bmod \Pi_d)$$

    Otherwise if irregular process coordinate, it has 0 but is never referred to.
  - `light.thresh` is $\Delta_d^- = \Delta_d^l + \Pi_d^- \cdot \delta_d^{\min}$ to represent, if regular process coordinate, the threshold beyond which subdomains has d-th dimensional edges of $\delta_d^{\min} + 1$.

– `light.fthresh` is the grid-size-aware counterpart of `light.thresh` to have $\Delta_d^- \cdot \gamma_d$.

Note that if $D < 3$, `n` = `light.n` = 1, `gsize` = 1.0 and other elements are 0 for all `Grid`$[d]$ such that $d \geq D$, but they are never referred to. `Grid`[] is initialized by `init_subdomain_actively()` or `init_subdomain_passively()`, and its grid-size-aware elements may be updated by `oh3_grid_size()`. Then it is referred to by `oh3_map_particle_to_neighbor()` through the macro `Map_Particle_To_Neighbor()`, `oh3_map_particle_to_subdomain()` directly and through `Map_Particle_To_Subdomain()`, and `map_irregular()`.

`S_subdomdesc` `SubDomainDesc`

- The array `SubDomainDesc`$[N]$ of the `S_subdomdesc` structure with the following elements has the coordinate information of subdomains from which the subdomain including a particluar coordinate point will be found when we have irregular process coordinate. Its element $[m]$ is not for the subdomain m but for the m-th smallest subdomain in an ordering of boundary edges.
  - `coord`$[d]$ ($d \in [0, D)$) has the following d-th dimensional information for the subdomain.
    - `c`$[\beta] = \{\delta_d^l(m'), \delta_d^u(m')\}$ is the d-th dimensional lower ($\beta = 0$) or upper ($\beta = 1$) boundary coordinate where m' is the sumdomain identifier of $[m]$.
    - `fc`$[\beta]$ is the grid-size-aware couterpart of `c`$[\beta]$ to have $\delta_d^\beta(m') \cdot \gamma_d$.
    - `h` is the smallest index of the array element among those which shares `c`[0] and `c`[1], or as follows more specifically.

$$c(i, d, \beta) = \texttt{SubDomainDesc}[i]\texttt{.coord}[d]\texttt{.c}[\beta]$$
$$\mathcal{S}(m, d) = \{i \,|\, c(i, d, \beta) = c(m, d, \beta),\ \beta \in [0, 1]\}$$
$$\texttt{SubDomainDesc}[m]\texttt{.coord}[d]\texttt{.h} = \min(\mathcal{S}(m, d))$$

    - `n` is the number of array elements which shares `c`[0] and `c`[1], i.e., `SubDomainDesc`$[m]$`.coord`$[d]$`.n` $= |\mathcal{S}(m, d)|$
  - `id` is the subdomain identifier of the array element.

The array is set to `NULL` by `init_subdomain_actively()` to mean regular process coordinate, or allocated and initialized by `init_subdomain_passively()` if we have irregular process coordinate. Then it may be updated by `oh3_grid_size()`. It is referred to by `oh3_map_particle_to_subdomain()`, `map_irregular()` and `map_irregular_range()` to find the subdomain in which a particle resides.

`S_message` `Message`

- The constant `S_message` structure `Message` has the following items to put an error message from `init_subdomain_actively()` or `init_subdomain_passively()` for d-th dimensional coordinate and/or lower/upper boundary of the system domain or a subdomain.
  - `xyz` is `"xyz"` to print 0-th dimension as "x", etc.
  - `loup`[2] is `"lower"` and `"upper"` to print "lower" ($\beta = 0$) or "upper" ($\beta = 1$).

```
EXTERN int (*SubDomains)[OH_DIMENSION][2];      /* [N][D][l,u] */
EXTERN double (*SubDomainsFloat)[OH_DIMENSION][2];
struct S_grid {
```

```
  int n, coord[2], size;
  double fcoord[2], fsize, gsize, rgsize;
  struct {
    int size, n, thresh;
    double rfsize, rfsizeplus, fthresh;
  } light;
};
EXTERN struct S_grid Grid[3];

struct S_subdomdesc {
  struct {
    int c[2], h, n;
    double fc[2];
  } coord[OH_DIMENSION];
  int id;
};
EXTERN struct S_subdomdesc *SubDomainDesc;

static struct S_message {
  char xyz[4];
  char loup[2][6];
} Message = {
  "xyz",
  {"lower", "upper"}
};
```

4.6.3 Domain and Subdomain Boundaries

We have the following three variables related to domain/subdomain boundaries.

nOfBoundaries
- The integer variable `nOfBoundaries` has the number of boundary conditions B given by the `nbound` argument of `oh3_init()`. It is initialized by `init3()` and is referred to by `set_border_exchange()` to access `BoundaryCommTypes`$[C][B][2][3]$ where C is the number of boundary communication types stored in `nOfExc`.

Boundaries
- The integer array `Boundaries`$[N][D][2]$ is the substance of the array pointed by `bounds` argument of `oh3_init()`. Its element $[m][d][\beta]$ has the boundary condition $b \in [0, B)$ for the d-th dimensional lower ($\beta = 0$) or upper ($\beta = 1$) boundary plane of the subdomain m. The array is allocated and initialized by `init3()` with values given through its argument `bounds` or given by `init_subdomain_actively()`. It (or its shadow pointed by `bounds`) is referred to by `init_subdomain_passively()`, `set_border_exchange()` and `oh3_map_particle_to_neighbor()` through the macro `Map_Particle_To_Neighbor()`.

Adjacent
- The integer array `Adjacent`$[D][2]$ has the identifier of the subdomain adjacent by the lower ($\beta = 0$) or upper ($\beta = 1$) d-th dimensional boundary plane of the primary subdomain of the local node in its element $[d][\beta]$. The array is initialized by `init3()` and is referred to by `init_subdomain_passively()` and `oh3_exchange_borders()`.

```
EXTERN int nOfBoundaries;
EXTERN int (*Boundaries)[OH_DIMENSION][2];      /* [N][D][l,u] */
EXTERN int Adjacent[OH_DIMENSION][2];           /* [D][l,u] */
```

4.6.4 Field Array Descriptors

The next declarations are for the following three variables to represent field-arrays.

`nOfFields`

- The integer variable `nOfFields` has the number of elmenents F of the arrays `FieldTypes`[] and `FieldDesc`[]. Its value is set by `init_fields()` as the number of elements in the argument `ftypes` of `oh3_init()` preceding its terminator element.

`OH_FTYPE_ES` `OH_FTYPE_LO` `OH_FTYPE_UP` `OH_FTYPE_BL` `OH_FTYPE_BU` `OH_FTYPE_RL` `OH_FTYPE_RU` `OH_FTYPE_N`

- Prior to declare the array `FieldTypes`$[F][7]$, we define seven constant macros `OH_FTYPE_`$\{$`ES`, `LO`, `UP`, `BL`, `BU`, `RL`, `RU`$\}$ to specify one of the elements in its second dimension, namely $\varepsilon(f)$, $e_l(f)$, $e_u(f)$, $e^b_l(f)$, $e^b_u(f)$, $e^r_l(f)$ and $e^r_u(f)$ for `FieldTypes`$[f][]$. We also define `OH_FTYPE_N` as the number of elements 7.

`FieldTypes`

- The integer array `FieldTypes`$[F][7]$ is the substance of the `ftypes` argument of `oh3_init()`. Its array element $[f][]$ has the following seven elements for the field-array identified by f.
  - $[0] = \varepsilon(f)$ is the number of elements associated to a grid point of the field-array.
  - $[1{:}2] = \{e_l(f), e_u(f)\}$ define lower/upper extensions required for the field-array besides those for communication.
  - $[3{:}4] = \{e^b_l(f), e^b_u(f)\}$ define lower/upper extensions for the broadcast of the field-array.
  - $[5{:}6] = \{e^r_l(f), e^r_u(f)\}$ define lower/upper extensions for the reduction of the field-array.

  The array is allocated and initialized by `init_fields()` and referred to by `set_field_descriptors()` and `transbound3()`.

`S_flddesc` `FieldDesc`

- The array `FieldDesc`$[F]$ of the `S_flddesc` structure with the following elements has the size information of field-array f for primary/secondary subdomain of the local node in $[f]$.
  - `esize` $= \varepsilon(f)$
  - `ext`$[0{:}1] = \{e^{\min}_l, e^{\max}_u\}$ where;

$$e^{\min}_l = \min\{e^\gamma_l(f), e_l(f), e^b_l(f), e^r_l(f)\}$$
$$e^{\max}_u = \max\{e^\gamma_u(f), e_u(f), e^b_u(f), e^r_u(f)\}$$

    where $e^\gamma_l(f)$ and $e^\gamma_u(f)$ is the minimum/maximum extensions required for the boundary communication of the field-array f.
  - `size`$[d] = \Phi_d(f) = \delta^{\max}_d + (e^{\max}_u(f) - e^{\min}_l(f))$ to represent the required size of d-th dimension of the field-array f.
  - `bc` and `red` is `S_brdesc` structure with the following elements where x is b for `bc` or r for `red`.

`S_brdesc`

* `base` is the index of the element of the conceptual one-dimensional array corresponding to the base grid point $[e_l^x(f)]\cdots[e_l^x(f)][0]$ of the field-array f. More specifically it is defined as follows.

$$b_{D-1} = e_l^x(f) \qquad b_d = b_{d+1}\,\Phi_d(f) + e_l^x(f) \qquad \texttt{base} = b_0 \cdot \varepsilon(f)$$

* `size`$[p]$ is the number of elements of field-array f to be broadcasted/reduced for the primary subdomain $m = n$ $(p = 0)$ or secondary subdomain $m = \mathit{parent}(n)$ of the local node n. More specifically it is defined as follows.

$$\upsilon_d(m) = \delta_d^u(m) - \delta_d^l(m) + e_u^x(f) - 1$$
$$a_{D-1} = \upsilon_{D-1}(m) \qquad a_d = a_{d+1}\,\Phi_d(f) + \upsilon_d(m)$$
$$\texttt{size}[p] = (a_0 + 1) \cdot \varepsilon(f) - \texttt{base}$$

The array is allocated and partly initialized by `init_fields()` and its `size`[] elements of `bc` and `red` are set by `set_field_descriptors()`. The functions which refer to the array are `oh3_bcast_field()`, `oh3_reduce_field()`, `oh3_allreduce_field()`, and `set_border_comm()` through the macro `Field_Disp()` which also used in `init_fields()` and `set_field_descriptors()`.

```
EXTERN int nOfFields;

#define OH_FTYPE_ES 0
#define OH_FTYPE_LO 1
#define OH_FTYPE_UP 2
#define OH_FTYPE_BL 3
#define OH_FTYPE_BU 4
#define OH_FTYPE_RL 5
#define OH_FTYPE_RU 6
#define OH_FTYPE_N  7
EXTERN int (*FieldTypes)[OH_FTYPE_N];          /* [F][es,lo,up,bl,bu,rl,ru] */

struct S_brdesc {
  int base, size[2];
};
struct S_flddesc {
  int esize, ext[2], size[OH_DIMENSION];
  struct S_brdesc bc, red;
};
EXTERN struct S_flddesc *FieldDesc;            /* [F] */
```

4.6.5 Boundary Communication Descriptors

The next declarations are for the following five variables to represent boundary communications of field-arrays.

`nOfExc`

- The integer variable `nOfExc` has the number of elements C of the arrays `BoundaryCommFields`[], `BoundaryCommTypes`[][][][] and `BorderExc`[][][][]. Its value is set by `init_fields()` as the number of elements in the argument `cfields` of `oh3_init()` preceding its terminator element.

BoundaryCommFields

- The integer array `BoundaryCommFields`$[C]$ is the substance of the `cfields` argument of `oh3_init()`. Its array element $[c]$ has a field-array identifier f for which boundary communication identified by c is defined. The array is allocated and initialized by `init_fields()` and referred to by `set_border_exchange()`.

OH_CTYPE_FROM
OH_CTYPE_TO
OH_CTYPE_SIZE
OH_CTYPE_N

- Prior to declare the array `BoundaryCommTypes`$[C][B][2][3]$, we define three constant macros `OH_CTYPE_`$\{$`FROM`$, $`TO`$, $`SIZE`$\}$ to specify one of the elements in its fouth dimension, namely e_f, e_t and s. We also define `OH_CTYPE_N` as the number of elements 3.

BoundaryCommTypes

- The integer array `BoundaryCommTypes`$[C][B][2][3]$ is the substance of the `ctypes` argument of `oh3_init()`. Its array element $[c][b][w][]$ has the following three elements for the downward ($w = 0$) or upward ($w = 1$) communication of the type c through a boundary plane having boundary condition b.
  - $[0] = e_f$ defines the first (with smallest coordinate) *sending plane* of those sent through a subdomain boundary plane and parallel to it. It is the displacement of the first plane from the boundary plane.
  - $[1] = e_t$ defines the first (with smallest coordinate) *receiving plane* of those received through a subdomain boundary plane and parallel to it. It is the displacement of the first plane from the boundary plane.
  - $[2] = s$ defines the number of sending/receiving planes.

  The array is allocated and initialized by `init_fields()` and referred to by `set_border_exchange()`.

S_bcomm
S_borderexc
BorderExc

- The array `BorderExc`$[C][2][D][2]$ of the `S_borderexc` structure with `send` and `recv` elements, which are `S_bcomm` structures with the following elements, has the information for a set of sending/receiving planes. Its element $[c][p][d][w]$ is for downward ($w = 0$) or upward ($w = 1$) communication of type c for field-array `BoundaryCommFields`$[c]$ through the d-th dimensional boundary plane of the primary ($p = 0$) or secondary ($p = 1$) subdomain of the local node.
  - `buf` is the index of a conceptual one-dimensinal array corresponding to the base grid point of the first sending/receiving plane.
  - `count` is the number of data elements of the `type` to be sent/received.
  - `deriv` is true (1) iff `type` is a derivative data type and thus should be freed when the associated secondary subdomain is changed by rebalancing.
  - `type` is the `MPI_Datatype` of data elements to be sent/received.

  The array is allocated and initialized by `init_fields()` and then is set up by `set_border_exchange()` and `set_border_comm()` while `clear_border_exchange()` reinitializes it, for the refereces from `oh3_exchange_borders()`.

```
EXTERN int nOfExc;
EXTERN int *BoundaryCommFields;                /* [C] */

#define OH_CTYPE_FROM 0
#define OH_CTYPE_TO   1
#define OH_CTYPE_SIZE 2
#define OH_CTYPE_N    3
```

```
EXTERN int (*BoundaryCommTypes)[2][OH_CTYPE_N]; /* [C][B][d,u][from,to,size] */

struct S_bcomm {
  int buf, count, deriv;
  MPI_Datatype type;
};
struct S_borderexc {
  struct S_bcomm send, recv;
};
EXTERN struct S_borderexc (*BorderExc)[2][OH_DIMENSION][2];
                                                    /* [C][ps][D][l,u] */
```

4.6.6 Function Prototypes

The next and last block is to declare the prototypes of the API function pairs each of which consists of API for Fortran and C, as listed below.

- The function `oh3_init[_]()` initializes data strucutures of the level-3 and lower level libraries.
- The function `oh13_init[_]()` initializes data strucutures of the level-3 and level-1 library.
- The function `oh3_grid_size[_]()` is to set grid size γ_d.
- The function `oh3_transbound[_]()` at first performs what its level-2 or level-1 counterpart `oh2_transbound[_]()` or `oh1_transbound[_]()` does according to the library initialized by `oh3_init[_]()` or `oh13_init[_]()`, to transfer particles or to have particle transfer schedule, and then maintain `FieldDesc` and `BorderExc` if the secondary domain of the local node was changed.
- The function `oh3_map_particle_to_neighbor[_]()` finds the subdomain in which a given particle resides, providing that the subdomain is a neighbor of the primary/secondary subdomain of the local node[51].
- The function `oh3_map_particle_to_subdomain[_]()` finds the subdomain in which a given particle resides, allowing that the subdomain is not necessary to be a neighbor of the primary/secondary subdomain of the local node[52].
- The function `oh3_bcast_field[_]()` performs broadcast communication for a field-array in the primary/secondary family of the local node.
- The function `oh3_reduce_field[_]()` performs reduction communication for a field-array in the primary/secondary family of the local node.
- The function `oh3_allreduce_field[_]()` performs all-reduce communication for a field-array in the primary/secondary family of the local node.
- The function `oh3_exchange_borders[_]()` exchanges a set of sending/receiving planes of a field-array of the primary subdomain of the local node and then, if specified, performs broadcast communication for the planes in the primary/secondary family of the local node.

[51] For downward compatibility, we have an aliase `oh3_map_region_to_adjacent_node_()`.
[52] For downward compatibility, we have an aliase `oh3_map_region_to_node_()`.

As done in §4.2.11 and §4.4.5, prior to showing the function prototypes, we show the third part of the header files ohhelp_c.h for C-coded simulators and ohhelp_f.h for Fortran-coded ones, which define the aliases of level-3 API functions. In the `#else` part of `#if␣OH_LIB_LEVEL=2`, at first they `#define` the aliases of API functions which do not have higher level counterparts.

```
#else
#define oh_grid_size(A1)                  oh3_grid_size(A1)
#define oh_bcast_field(A1,A2,A3)          oh3_bcast_field(A1,A2,A3)
#define oh_reduce_field(A1,A2,A3)         oh3_reduce_field(A1,A2,A3)
#define oh_allreduce_field(A1,A2,A3)      oh3_allreduce_field(A1,A2,A3)
#define oh_exchange_borders(A1,A2,A3,A4) oh3_exchange_borders(A1,A2,A3,A4)
```

Then ohhelp_c.h gives the prototypes of the functions above, which are also given in ohhelp3.h, while their Fortran versions are given in oh_mod3.F90 as shown in §3.6.

```
void oh3_grid_size(double size[OH_DIMENSION]);
void oh3_bcast_field(void *pfld, void *sfld, int ftype);
void oh3_reduce_field(void *pfld, void *sfld, int ftype);
void oh3_allreduce_field(void *pfld, void *sfld, int ftype);
void oh3_exchange_borders(void *pfld, void *sfld, int ctype, int bcast);
```

Then ohhelp3.h continues prototype declaration for Fortran API functions.

```
void oh3_grid_size_(double size[OH_DIMENSION]);
void oh3_bcast_field_(void *pfld, void *sfld, int *ftype);
void oh3_reduce_field_(void *pfld, void *sfld, int *ftype);
void oh3_allreduce_field_(void *pfld, void *sfld, int *ftype);
void oh3_exchange_borders_(void *pfld, void *sfld, int *ctype, int *bcast);
```

We repeat the alias and prototype declarations above if `OH_LIB_LEVEL` = 3, because remaining functions have level-4p (or higher) counterparts. At first we do for `oh3_map_particle_to_neighbor()` and `oh3_map_particle_to_subdomain()` in one-dimensional simulations with `OH_DIMENSION` = 1, for ohhelp_c.h and ohhelp_f.h;

```
#if OH_LIB_LEVEL==3
#if OH_DIMENSION==1
#define oh_map_particle_to_neighbor(A1,A2) \
        oh3_map_particle_to_neighbor(A1,A2)
#define oh_map_particle_to_subdomain(A1) \
        oh3_map_particle_to_subdomain(A1)
```

then for ohhelp_c.h and ohhelp3.h;

```
int  oh3_map_particle_to_neighbor(double *x, int ps);
int  oh3_map_particle_to_subdomain(double x);
```

and finally for ohhelp3.h.

```
int  oh3_map_region_to_adjacent_node_(double *x, int *ps);
int  oh3_map_particle_to_neighbor_(double *x, int *ps);
int  oh3_map_region_to_node_(double *x);
int  oh3_map_particle_to_subdomain_(double *x);
```

Now we repeat the declarations above for three-dimensional simulations with OH_DIMENSION = 2, for ohhelp_c.h and ohhelp_f.h;

```
#elif OH_DIMENSION==2
#define oh_map_particle_to_neighbor(A1,A2,A3) \
        oh3_map_particle_to_neighbor(A1,A2,A3)
#define oh_map_particle_to_subdomain(A1,A2) \
        oh3_map_particle_to_subdomain(A1,A2)
```

then for ohhelp_c.h and ohhelp3.h;

```
int  oh3_map_particle_to_neighbor(double *x, double *y, int ps);
int  oh3_map_particle_to_subdomain(double x, double y);
```

and finally for ohhelp3.h.

```
int  oh3_map_region_to_adjacent_node_(double *x, double *y, int *ps);
int  oh3_map_particle_to_neighbor_(double *x, double *y, int *ps);
int  oh3_map_region_to_node_(double *x, double *y);
int  oh3_map_particle_to_subdomain_(double *x, double *y);
```

Then next iteration is with OH_DIMENSION = 3, for ohhelp_c.h and ohhelp_f.h;

```
#else
#define oh_map_particle_to_neighbor(A1,A2,A3,A4) \
        oh3_map_particle_to_neighbor(A1,A2,A3,A4)
#define oh_map_particle_to_subdomain(A1,A2,A3) \
        oh3_map_particle_to_subdomain(A1,A2,A3)
```

then for ohhelp_c.h and ohhelp3.h;

```
int  oh3_map_particle_to_neighbor(double *x, double *y, double *z, int ps);
int  oh3_map_particle_to_subdomain(double x, double y, double z);
```

and finally for ohhelp3.h.

```
int  oh3_map_region_to_adjacent_node_(double *x, double *y, double *z,
                                      int *ps);
int  oh3_map_particle_to_neighbor_(double *x, double *y, double *z,
                                   int *ps);
int  oh3_map_region_to_node_(double *x, double *y, double *z);
int  oh3_map_particle_to_subdomain_(double *x, double *y, double *z);
#endif
```

The final repetition is for the declarations of other functions for ohhelp_c.h and ohhelp_f.h;

```
#define \
oh_init(A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,A15,A16,A17,A18,
A19,A20,A21,A22,A23) \
oh3_init(A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,A15,A16,A17,A18,
A19,A20,A21,A22,A23)
#define oh_transbound(A1,A2)            oh3_transbound(A1,A2)
```

then for ohhelp_c.h and ohhelp3.h;

```
void oh3_init(int **sdid, int nspec, int maxfrac, int **nphgram, int **totalp,
              struct S_particle **pbuf, int **pbase, int maxlocalp,
              void *mycomm, int **nbor, int *pcoord,
              int **sdoms, int *scoord, int nbound, int *bcond, int **bounds,
              int *ftypes, int *cfields, int *ctypes, int **fsizes,
              int stats, int repiter, int verbose);
void oh13_init(int **sdid, int nspec, int maxfrac, int **nphgram,
               int **totalp, int **rcounts, int **scounts,
               void *mycomm, int **nbor, int *pcoord,
               int **sdoms, int *scoord, int nbound, int *bcond, int **bounds,
               int *ftypes, int *cfields, int *ctypes, int **fsizes,
               int stats, int repiter, int verbose);
int  oh3_transbound(int currmode, int stats);
```

and finally for ohhelp3.h.

```
void oh3_init_(int *sdid, int *nspec, int *maxfrac, int *nphgram,
               int *totalp, struct S_particle *pbuf, int *pbase,
               int *maxlocalp, struct S_mycommf *mycomm, int *nbor,
               int *pcoord, int *sdoms, int *scoord, int *nbound, int *bcond,
               int *bounds, int *ftypes, int *cfields, int *ctypes,
               int *fsizes, int *stats, int *repiter, int *verbose);
void oh13_init_(int *sdid, int *nspec, int *maxfrac, int *nphgram,
                int *totalp, int *rcounts, int *scounts,
                struct S_mycommf *mycomm, int *nbor, int *pcoord,
                int *sdoms, int *scoord, int *nbound, int *bcond, int *bounds,
                int *ftypes, int *cfields, int *ctypes, int *fsizes,
                int *stats, int *repiter, int *verbose);
int  oh3_transbound_(int *currmode, int *stats);
```

Next and finally, we declare the prototypes of the following functions so that they are called from level-4p (and other higher level) library.

- The function `init3()` is the body of `oh3_init()`.
- The function `set_field_descriptors()` sets `FieldDesc`$[f]$`.{bc,red}.size`$[p]$ for all $f \in [0, F)$ and given $p \in \{0, 1\}$.
- The function `clear_border_exchange()` initializes `BorderExc`$[c][1][d][\beta]$`.{send, recv}` for all $c \in [0, C)$, $d \in [0, D)$ and $\beta \in \{0, 1\}$, or reinitializes them for the subdomain which the local node has had as the secondary one but discarded by rebalancing.
- The function `map_irregular_subdomain()` finds the subdomain of irregular process coordinate in which a particle resides.

```
/* Prototype for the function called from higher-level library code */
void init3(int **sdid, int nspec, int maxfrac, int **nphgram, int **totalp,
           int **rcounts, int **scounts, struct S_particle **pbuf, int **pbase,
           int maxlocalp, struct S_mycommc *mycommc, struct S_mycommf *mycommf,
```

```
           int **nbor, int *pcoord, int **sdoms, int *scoord, int nbound,
           int *bcond, int **bounds, int *ftypes, int *cfields, int cfid,
           int *ctypes, int **fsizes, int stats, int repiter, int verbose,
           int skip2);
void set_field_descriptors(int (*ft)[OH_FTYPE_N], int sd[OH_DIMENSION][2],
                           int ps);
void clear_border_exchange();
int  map_irregular_subdomain(double x, double y, double z);
```

4.7 C Source File ohhelp3.c

4.7.1 Header File Inclusion

The first job done in ohhelp3.c is the inclusion of the header files ohhelp1.h, ohhelp2.h and ohhelp3.h. Before the inclusion of ohhelp1.h and ohhelp2.h, we `#define` the macro `EXTERN` as `extern` so as to make variables declared in the files external, but after that we make it `#undef`'iend and then `#define` it as empty so as to provide variables declared in ohhelp3.h with their homes, as discussed in §4.2.3.

```
#define EXTERN extern
#include "ohhelp1.h"
#include "ohhelp2.h"
#undef  EXTERN
#define EXTERN
#include "ohhelp3.h"
```

4.7.2 Function Prototypes

The next and last job to do prior to function definitions is to declare the prototypes of the following functions private for the level-3 library.

- The function `init_subdomain_actively()` initializes domain/subdomain and their boudnary descriptors `SubDomains`[][][], `Grid`[] and `Boundaries`[][][] when regular process coordinate is specified by `sdoms` argument of `oh3_init()`.
- The function `init_subdomain_passively()` initializes domain/subdomain and their boudnary descriptors `Grid`[], `SubDomainDesc`[] and `Boundaries`[][][] according to `SubDomains`[][][] = `sdoms` argument of `oh3_init()` which specifies irregular process coordinate.
- The function `comp_xyz()` is used to sort `SubDomainDesc`[] in `init_subdomain_passively()` through `qsort()`.
- The function `init_fields()` initializes field-array descriptors `FieldTypes`[][] and `FieldDesc`[]. It also initializes boundary communication descriptors `BoundaryCommFields`[], `BoundaryCommTypes`[][][][] and `BorderExc`[][][][].
- The function `set_border_exchange()` sets up the elements of `BorderExc`$[c][p][d][w]$ for given c and p and for all $d \in [0, D)$ and $w \in \{0, 1\}$.
- The function `set_border_comm()` sets up `BorderExc`$[c][p][d][w].\{$`send`$,$`recv`$\}$ for given $c \in [0, C)$, $p \in \{0, 1\}$ and $w \in \{0, 1\}$, and for all $d \in [0, D)$.
- The function `transbound3()` is the body of `oh3_transbound()`.
- The function `map_irregular()` is the body of `map_irregular_subdomain()`.
- The function `map_irregular_range()` is to find a set of candidate subdomains from which the subdomain is searched by `map_irregular()`.

```
static void init_subdomain_actively(int (*sd)[OH_DIMENSION][2],
                                    int sc[OH_DIMENSION][2],
                                    int *pcoord, int bc[OH_DIMENSION][2],
                                    int (*bd)[OH_DIMENSION][2], int nb,
                                    int bbase);
static void init_subdomain_passively(int (*sd)[OH_DIMENSION][2],
                                     int (*bd)[OH_DIMENSION][2], int nb,
                                     int bbase);
static int  comp_xyz(const void* aa, const void* bb);
static void init_fields(int (*ft)[OH_FTYPE_N], int *cf, int cfid,
                        int (*ct)[2][OH_CTYPE_N], int nb,
                        int sd[OH_DIMENSION][2], int **fsizes);
static void set_border_exchange(int e, int ps, MPI_Datatype type);
static void set_border_comm(int esize, int f, int *xyz, int *wdh,
                            int (*exti)[2], int (*exto)[2],
                            int (*off)[2], int (*size)[2],
                            int lu, int sr, MPI_Datatype basetype,
                            struct S_borderexc bx[OH_DIMENSION][2]);
static int  transbound3(int currmode, int stats, int level);
static int  map_irregular(double p0, double p1, double p2, int dim, int from,
                          int n);
static int  map_irregular_range(double p, int dim, int from, int to);
```

4.7.3 `oh3_init()` and `oh13_init()`

`oh3_init_()` `oh3_init()`

The API functions `oh3_init_()` for Fortran and `oh3_init()` for C receive a set of array/structure variables through which level-1 to level-3 library functions communicate with the simulator body, and a few integer parameters to specify the behavior of the library. The functions have the following arguments.

- `sdid`
  `nspec`
  `maxfrac`
  `nphgram`
  `totalp`
  The five arguments above are perfectly equivalent to those of the level-1 counterparts `oh1_init[_]()`.

- `pbuf`
  `pbase`
  `maxlocalp`
  The three arguments above are perfectly equivalent to those of the level-2 counterparts `oh2_init[_]()`.

- `mycomm`
  `nbor`
  `pcoord`
  The three arguments above are perfectly equivalent to those of the level-1 counterparts `oh1_init[_]()`.

- The argument `sdoms` should be the (double) pointer to an integer arrray of $[N][D][2]$ being the shadow of `SubDomains`[][][]. If `sdoms`[0][0][0] > `sdoms`[0][0][1], or `sdoms` points `NULL` to make `init3()` allocate the array, all the elements of the array $[m][d][\beta]$ are filled by `init_subdomain_actively()` to have boundary coordinates of subdomain m, $\delta_d^l(m)$ ($\beta = 0$) and $\delta_d^u(m)$ ($\beta = 1$), for regular process coordinate. Otherwise, the array should have $\delta_d^{\{l,u\}}(m)$ to be referred to by `init_subdomain_passively()` to create `SubDomainDesc`[] for irregular process coordinate.

- The argument `scoord` should be the pointer to an integer array of $[D][2]$ to specify boundary coordinates of the system domain, Δ_d^l ($\beta = 0$) and Δ_d^u ($\beta = 1$), in its element $[d][\beta]$, if regular process coordinate is specified. Otherwise, it can be `NULL` or can point anything.

- The integer argument `nbound` shoud have the number of boundary conditions $B =$ `nOfBoundaries`.

- The argument `bcond` should be the pointer to an integer array of $[D][2]$ to specify the boundary condition type $b \in [0, B)$ of the d-th dimensional lower ($\beta = 0$) or upper ($\beta = 1$) boundary plane of the system domain in its element $[d][\beta]$, if regular process coordinate is specified. Otherwise, it can be `NULL` or can point anything.

- The argument `bounds` should be the (double) pointer to an integer array of $[N][D][2]$ being the shadow of `Boundaries`[][][] or to `NULL`. If regular process coordinate is specified by `sdoms`, all the elements of the array $[m][d][\beta]$ are filled by `init_subdomain_actively()` to have the boundary conditions of d-th dimensional lower ($\beta = 0$) and upper ($\beta = 1$) boundary planes of subdomain m. Otherwise, the array should have the boundary conditions for each boundary plane of each subdomain to be referred to by `init_subdomain_passively()`.

- The argument `ftypes` should be the pointer to an integer array of $[F+1][7]$ being the shadow of `FieldTypes`[] to have $\varepsilon(f)$, $e_l(f)$, $e_u(f)$, $e_l^b(f)$, $e_u^b(f)$, $e_l^r(f)$ and $e_u^r(f)$ in $[f][0{:}6]$ for all $f \in [0, F)$ being field-array identifiers, while $[F][0] \leq 0$ as the terminator.

- The argument `cfields` should be the pointer to an integer arryay of $[C + 1]$ being the shadow of `BoundaryCommFields`[] to have an index $f \in [0, F)$ of `FieldTypes`[] in its element $[c]$ to specify that `ctypes`$[c][B][2][3]$ = `BoundaryCommTypes`$[c][B][2][3]$ is for field-array f, while $[C] < 1$ for `oh3_init_()` or $[C] < 0$ for `oh3_init()` as the terminator.

- The argument `ctypes` should be the pointer to an integer array of $[C][B][2][3]$ to specify three parameters e_f, e_t and s of sending/receiving boundary planes for the type c downward ($w = 0$) and upward ($w = 1$) boundary communication through a boundary plane of the boundary condition b in its element $[c][b][w][0{:}2]$.

- The argument `fsizes` should be the (double) pointer to an integer array of $[F][D][2]$ or to `NULL`. Its element $[f][d][\beta]$ is filled with $\phi_d^l(f)$ ($\beta = 0$) or $\phi_d^u(f)$ ($\beta = 1$) to specify the lower or upper terminal index of the field-array f.

- `stats`
  `repiter`

`verbose`
The three arguments above are perfectly equivalent to those of the level-1 countgerparts `oh1_init[_]()`.

`oh13_init_()` `oh13_init()` We also have two additional API functions for initialization, namely `oh13_init_()` for Fortran and `oh13_init()` for C to perform what `oh3_init[_]()` does but excluding what `oh2_init[_]()` does. Therefore, they have the following arguments equivalent to those of `oh1_init[_]()` and/or `oh3_init[_]()`.

- The following eleven are equivalent to those of `oh1_init[_]()` and `oh3_init[_]()`.

  `sdid`, `nspec`, `maxfrac`, `nphgram`, `totalp`, `mycomm`, `nbor`, `pcoord`, `stats`, `repiter`, `verbose`

- The following two are equivalent to those of `oh1_init[_]()` but `oh3_init[_]()` does not have them.

  `rcounts`, `scounts`

- The following nine are equivalent to those of `oh3_init[_]()`'s own.

  `sdoms`, `scoord`, `nbound`, `bcond`, `bounds`, `ftypes`, `cfields`, `ctypes`, `fsizes`

These four API functions almost simply call the initializer function `init3()` passing all given arguments to it except for the followings.

- `oh3_init_()` and `oh13_init_()` pass the pointers to `sdid`, `nphgram`, `totalp`, `nbor`, `sdoms`, `bounds` and `fsizes` rather than themselves. `oh3_init_()` also does so for `pbuf` and `pbase`, while `oh13_init_()` does so for `rcounts` and `scounts`.
- `oh3_init_()` and `oh13_init_()` pass `mycomm` to `mycommf` of `init3()` while `NULL` is passed through `mycommc` of `init3()` to keep it from allocation of `MyCommC`.
- `oh3_init()` and `oh13_init()` pass `mycomm` to `mycommc` of `init3()` while `NULL` is passed through `mycommf` of `init3()` telling it that the body of `MyCommF` is not required. It also *casts* the argument as `S_mycommc` pointer type, because `mycomm` is declared as a `void` pointer to allow the simulator body to be completely unaware of the structure.
- Prior to calling `init3()`, `oh3_init_()` lets `specBase` be 1 to indicate the species in `S_particle` structures is represented by one-origin manner, while `oh3_init()` lets it be 0 to indicate zero-origin numbering.
- `oh3_init_()` and `oh13_init_()` pass -1 to `cfid` argument while `oh3_init()` and `oh13_init()` pass 0 to it, in order to obtain the boundary condition and field-array identifiers b and f from those specified in `bounds` and `ftypes`, b' and f', by $b = b' +$ `cfid` and $f = f' +$ `cfid`.
- `oh3_init[_]()` passes `NULL` to `rcounts` and `scounts` of `init3()` because they are not required. Similarly, `oh13_init[_]()` passes `NULL` to `pbuf` and `pbase` and 0 to `maxlocalp` which are used only in the level-2 library.
- `oh3_init[_]()` passes 0 to the `skip2` argument of `init3()` while `oh13_init[_]()` passes 1 to it, to let `init3()` skip the initialization for the level-2 library iff the argument is 1.

```
void
oh3_init_(int *sdid, int *nspec, int *maxfrac, int *nphgram,
          int *totalp, struct S_particle *pbuf, int *pbase, int *maxlocalp,
          struct S_mycommf *mycomm, int *nbor, int *pcoord,
          int *sdoms, int *scoord, int *nbound, int *bcond, int *bounds,
          int *ftypes, int *cfields, int *ctypes, int *fsizes,
          int *stats, int *repiter, int *verbose) {
  specBase = 1;
  init3(&sdid, *nspec, *maxfrac, &nphgram, &totalp, NULL, NULL, &pbuf, &pbase,
        *maxlocalp, NULL, mycomm, &nbor, pcoord, &sdoms, scoord, *nbound,
        bcond, &bounds, ftypes, cfields, -1, ctypes, &fsizes,
        *stats, *repiter, *verbose, 0);
}
void
oh3_init(int **sdid, int nspec, int maxfrac, int **nphgram,
         int **totalp, struct S_particle **pbuf, int **pbase, int maxlocalp,
         void *mycomm, int **nbor, int *pcoord,
         int **sdoms, int *scoord, int nbound, int *bcond, int **bounds,
         int *ftypes, int *cfields, int *ctypes, int **fsizes,
         int stats, int repiter, int verbose) {
  specBase = 0;
  init3(sdid, nspec, maxfrac, nphgram, totalp, NULL, NULL, pbuf, pbase,
        maxlocalp, (struct S_mycommc*)mycomm, NULL, nbor, pcoord, sdoms,
        scoord, nbound, bcond, bounds, ftypes, cfields, 0, ctypes, fsizes,
        stats, repiter, verbose, 0);
}
void
oh13_init_(int *sdid, int *nspec, int *maxfrac, int *nphgram,
           int *totalp, int *rcounts, int *scounts,
           struct S_mycommf *mycomm, int *nbor, int *pcoord,
           int *sdoms, int *scoord, int *nbound, int *bcond, int *bounds,
           int *ftypes, int *cfields, int *ctypes, int *fsizes,
           int *stats, int *repiter, int *verbose) {
  init3(&sdid, *nspec, *maxfrac, &nphgram, &totalp, &rcounts, &scounts,
        NULL, NULL, 0, NULL, mycomm, &nbor, pcoord, &sdoms, scoord, *nbound,
        bcond, &bounds, ftypes, cfields, -1, ctypes, &fsizes,
        *stats, *repiter, *verbose, 1);
}
void
oh13_init(int **sdid, int nspec, int maxfrac, int **nphgram,
          int **totalp, int **rcounts, int **scounts,
          void *mycomm, int **nbor, int *pcoord,
          int **sdoms, int *scoord, int nbound, int *bcond, int **bounds,
          int *ftypes, int *cfields, int *ctypes, int **fsizes,
          int stats, int repiter, int verbose) {
  init3(sdid, nspec, maxfrac, nphgram, totalp, rcounts, scounts, NULL, NULL,
        0, (struct S_mycommc*)mycomm, NULL, nbor, pcoord, sdoms, scoord,
        nbound, bcond, bounds, ftypes, cfields, 0, ctypes, fsizes,
        stats, repiter, verbose, 1);
}
```

4.7.4 init3()

init3() The function init3(), called from oh3_init[_]() and oh13_init[_](), implements the initialization for those API functions. The arguments of the function are almost same as the union of those of oh3_init() and oh13_init(), but their mycomm is split into two arguments mycommc and mycommf and there are two additions cfid and skip2 as discussed in §4.7.3.

```
void
init3(int **sdid, int nspec, int maxfrac, int **nphgram,
      int **totalp, int **rcounts, int **scounts,
      struct S_particle **pbuf, int **pbase, int maxlocalp,
      struct S_mycommc *mycommc, struct S_mycommf *mycommf,
      int **nbor, int *pcoord, int **sdoms, int *scoord,
      int nbound, int *bcond, int **bounds, int *ftypes,
      int *cfields, int cfid, int *ctypes, int **fsizes,
      int stats, int repiter, int verbose, int skip2) {
  int nn;
  int (*sd)[OH_DIMENSION][2]=(int(*)[OH_DIMENSION][2])*sdoms;
  double (*sdf)[OH_DIMENSION][2];
  int (*sc)[2]=(int(*)[2])scoord;
  int (*bc)[2]=(int(*)[2])bcond;
  int (*bd)[OH_DIMENSION][2]=(int(*)[OH_DIMENSION][2])*bounds;
  int (*ft)[OH_FTYPE_N]=(int(*)[OH_FTYPE_N])ftypes;
  int (*ct)[2][OH_CTYPE_N]=(int(*)[2][OH_CTYPE_N])ctypes;
  int d, n, m;
```

First, the function calls its level-1 or level-2 counterpart init1() or init2() according to the specification given by skip2 and then set it into excludeLevel2 so that transbound3() refers to it to determine which of transbound1() or transbound2() should be called.

```
  if (skip2)
    init1(sdid, nspec, maxfrac, nphgram, totalp, rcounts, scounts,
          mycommc, mycommf, nbor, pcoord, stats, repiter, verbose);
  else
    init2(sdid, nspec, maxfrac, nphgram, totalp, pbuf, pbase, maxlocalp,
          mycommc, mycommf, nbor, pcoord, stats, repiter, verbose);
  excludeLevel2 = skip2;
  nn = nOfNodes;
```

Next, we allocate shadow arrays of SubDomains$[N][D][2]$ and/or Boundaries$[N][D][2]$ by mem_alloc() if the arguments sdoms and/or bounds point NULL. In addition, $[0][0][0]$ and $[0][0][1]$ of the shadow of SubDomains[][][] are set to 0 and -1 to specify regular process coordinate.

```
  if (!sd) {
    sd = (int(*)[OH_DIMENSION][2])
         (*sdoms = (int*)mem_alloc(sizeof(int), nn*OH_DIMENSION*2,
                                   "SubDomains"));
    sd[0][OH_DIM_X][OH_LOWER] = 0;  sd[0][OH_DIM_X][OH_UPPER] = -1;
  }
  if (!bd)
```

```
bd = (int(*)[OH_DIMENSION][2])
     (*bounds = (int*)mem_alloc(sizeof(int), nn*OH_DIMENSION*2,
                                "Boundaries"));
```

Next we initialize `Adjacent`$[d][\beta]$ for all $d \in [0, D)$ and $\beta \in \{0, 1\}$ referring to `DstNeighbors`[] whose element $[k]$ where $k = \sum_{i=0}^{D-1} \nu_i 3^i$ has $r_k = rank(\pi_0+\nu_0-1, \ldots, \pi_{D-1}+\nu_{D-1}-1)$ or $-(r_k+1)$ where $(\pi_0, \ldots, \pi_{D-1})$ is the coordinate of the local node in the D-dimensional process coordinate space. Since `Adjacent`$[d][\beta]$ should have $rank(\pi'_0, \ldots \pi'_{D-1})$ where $\pi'_d = \pi_d + 2\beta - 1$ and $\pi'_i = \pi_i$ for all $i \neq d$, i.e., $\nu_d = 2\beta$ and $\nu_i = 1$ for all $i \neq d$, it should be set to the following.

$$k(d,\beta) = \sum_{\substack{0 \le i < D \\ i \neq d}} 3^i + 2\beta \cdot 3^d = \sum_{i=0}^{D-1} 3^i - 3^d + 2\beta \cdot 3^d$$

$$= (3^D - 1)/2 + (2\beta - 1)3^d = \lfloor 3^D/2 \rfloor + \begin{cases} -3^d & \beta = 0 \\ 3^d & \beta = 1 \end{cases}$$

$$r'_k = \texttt{DstNeighbors}[k] \qquad r_k = \begin{cases} r'_k & r'_k \ge 0 \\ -(r'_k + 1) & r'_k < 0 \end{cases}$$

$$\texttt{Adjacent}[d][\beta] = r'_{k(d,\beta)}$$

Note that if the local node has a non-existent neighbor in `DstNeighbors`$[k]$, it was set to $-(N+1)$ by `init1()` and thus its correspondent in `Adjacent`[][] is set to N.

```
for (d=0,n=1,m=OH_NEIGHBORS>>1; d<OH_DIMENSION; d++,n*=3) {
  int nl=DstNeighbors[m-n], nu=DstNeighbors[m+n];
  Adjacent[d][OH_LOWER] = nl<0 ? -(nl+1) : nl;
  Adjacent[d][OH_UPPER] = nu<0 ? -(nu+1) : nu;
}
```

Next if `*sdoms`[0][0][0] > `*sdoms`[0][0][1] meaning regular process coordinate, we call `init_subdomain_actively()` to initialize `*sdoms`[][][] and `*bounds`[][][]. Otherwise, i.e., irregular process coordinate, we call `init_subdomain_passively()` to create `SubDomainDesc`[]. Both functions also initialize `Grid`[3].

```
if (sd[0][OH_DIM_X][OH_LOWER]>sd[0][OH_DIM_X][OH_UPPER])
  init_subdomain_actively(sd, sc, pcoord, bc, bd, nbound, -cfid);
else
  init_subdomain_passively(sd, bd, nbound, -cfid);
```

Then we allocate `SubDomains`$[N][D][2]$ and `Boundaries`$[N][D][2]$ by `malloc()` and then copy their shadows `*sdoms`[][][] and `*bounds`[][][] to them by `memcpy()`. We also allocate `SubDomainsFloat`$[N][D][2]$ being grid-size-aware counterpart of `SubDomains`[][][] and copy `*sdoms`[][][] into it with integer/floating-point conversion to make it has the default coordinate values with grid size $\gamma_d = 1$ for all $d \in [0, D)$. After that, if `cfid` $= -1$ meaning that `*bounds`[][][] $\in [1, B]$, all elements of `Boundaries`[][][] are decremented so that they have values in $[0, B)$.

```
SubDomains = (int(*)[OH_DIMENSION][2])
             mem_alloc(sizeof(int), nn*OH_DIMENSION*2, "SubDomains");
sdf = SubDomainsFloat =
```

```
        (double(*)[OH_DIMENSION][2])
        mem_alloc(sizeof(double), nn*OH_DIMENSION*2, "SubDomainsFloat");
  Boundaries = (int(*)[OH_DIMENSION][2])
               mem_alloc(sizeof(int), nn*OH_DIMENSION*2, "Boundaries");
  memcpy(SubDomains, sd, sizeof(int)*nn*OH_DIMENSION*2);
  for (n=0; n<nn; n++)  for (d=0; d<OH_DIMENSION; d++) {
    sdf[n][d][OH_LOWER] = sd[n][d][OH_LOWER];
    sdf[n][d][OH_UPPER] = sd[n][d][OH_UPPER];
  }
  memcpy(Boundaries, bd, sizeof(int)*nn*OH_DIMENSION*2);
  bd = Boundaries;
  if (cfid) {
    for (n=0; n<nn; n++)  for (d=0; d<OH_DIMENSION; d++) {
      bd[n][d][OH_LOWER]--;  bd[n][d][OH_UPPER]--;
    }
  }
```

Finally we call `init_fields()` to initialize field-array and boundary communication descriptors.

```
  init_fields(ft, cfields, cfid, ct, nbound, sd[myRank], fsizes);
}
```

4.7.5 `init_subdomain_actively()`

`init_subdomain_actively()`

The function `init_subdomain_actively()`, called solely from `init3()`, initializes `*sdoms`$[N][D][2]$ = `sd`$[N][D][2] = \{\delta_d^{\{l,u\}}(m)\}$ and `Grid`[3] referring to `scoord`$[D][2]$ = `sc`$[D][2] = \{\Delta_d^{\{l,u\}}\}$ and `pcoord`$[D] = \{\Pi_d\}$. It also initializes `*bounds`$[N][D][2]$ = `bd`$[N][D][2]$ to have a value in $[b, B + b)$, where B is given through `nbound` = `nb` and $b \in \{0, 1\}$ is given through `bbase`, referring to `bcond`$[D][2]$ = `bc`$[D][2]$, for regular process coordintate.

```
static void
init_subdomain_actively(int (*sd)[OH_DIMENSION][2], int sc[OH_DIMENSION][2],
                        int *pcoord, int bc[OH_DIMENSION][2],
                        int (*bd)[OH_DIMENSION][2], int nb, int bbase) {
  int nn=nOfNodes, pqr=1;
  int d, lu, i, j, k, x, y, z, n;
```

At first we set `SubDomainDesc` to `NULL` to indicate regular process coordinate, and then initialize `Grid`$[d]$ for all $d \in [0, D)$ as follows with $\gamma_d = 1$ for all d at initial.

- Define `Grid`$[d]$`.coord`$[\beta]$ = `Grid`$[d]$`.fcoord`$[\beta]$ = `scoord`$[d][\beta] = \Delta_d^\beta$ and `Grid`$[d].n$ = `pcoord`$[n] = \Pi_d$ simply.
- Let $\overline{\Delta_d} = (\Delta_d^u - \Delta_d^l)$, then define `Grid`$[d]$`.light.size` $= \delta_d^{\min} = \lfloor \overline{\Delta_d}/\Pi_d \rfloor$ also simply. Also define `Grid`$[d]$`.light.rfsize` $= 1/\delta_d^{\min}$ and `Grid`$[d]$`.light.rfsizeplus` $= 1/(\delta_d^{\min} + 1)$.
- Define `Grid`$[d]$`.light.thresh` = `Grid`$[d]$`.light.fthresh` $= \Delta_d^- = \Delta_d^l + \Pi_d^- \cdot \delta_d^{\min}$.

- Let $\mathit{\Pi}_d^- = \texttt{Grid}[d]\texttt{.light.n} = \mathit{\Pi}_d - (\overline{\mathit{\Delta}_d} \bmod \mathit{\Pi}_d)$, then $\texttt{Grid}[d]\texttt{.size} = \texttt{Grid}[d]\texttt{.fsize} = \delta_d^{\max} = \lceil \overline{\mathit{\Delta}_d}/\mathit{\Pi}_d \rceil$ is defined as;

$$\delta_d^{\max} = \begin{cases} \delta_d^{\min} & \mathit{\Pi}_d^- = \mathit{\Pi}_d \\ \delta_d^{\min} + 1 & \mathit{\Pi}_d^- \neq \mathit{\Pi}_d \end{cases}$$

- Finally, initialize $\texttt{Grid}[d]\texttt{.gsize} = \gamma_d = 1$ and $\texttt{Grid}[d]\texttt{.rgsize} = 1/\gamma_d = 1$ as default.

In addition, we check if $\mathit{\Pi}_d > 0$ and $\overline{\mathit{\Delta}_d} > 0$, or abort the execution by `errstop()`. Note that if $D < 3$, we set $\mathit{\Pi}_d = \mathit{\Pi}_d^- = 1$, $\gamma_d = 1/\gamma_d = 1$, $\mathit{\Delta}_d^l = \mathit{\Delta}_d^u = \mathit{\Delta}_d^- = 0$ and $\delta_d^{\max} = \delta_d^{\min} = 0$ with their reciprocals for all $d \geq D$.

```
SubDomainDesc = NULL;
for (d=0; d<OH_DIMENSION; d++) {
  int lo = Grid[d].coord[OH_LOWER] = sc[d][OH_LOWER];
  int up = Grid[d].coord[OH_UPPER] = sc[d][OH_UPPER];
  int size = up - lo;
  int ave, nl;
  Grid[d].fcoord[OH_LOWER] = lo;  Grid[d].fcoord[OH_UPPER] = up;
  n = Grid[d].n = pcoord[d];
  if (n<=0)
    errstop("# of %c-nodes (%d) should be positive", Message.xyz[d], n);
  if (size<=0)
    errstop("upper edge of %c-coordinate (%d) should be greater than "
            "lower edge (%d)", Message.xyz[d], up, lo);
  ave = Grid[d].light.size = size/n;
  Grid[d].light.rfsize = 1.0/(double)ave;
  Grid[d].light.rfsizeplus = 1.0/(double)(ave+1);
  nl =  Grid[d].light.n = n - size%n;
  Grid[d].light.fthresh = (Grid[d].light.thresh = lo + nl * ave);
  Grid[d].fsize = (Grid[d].size = n==nl ? ave : ave+1);
  Grid[d].gsize = Grid[d].rgsize = 1.0;
  pqr *= n;
}
for (; d<3; d++) {
  Grid[d].n = Grid[d].light.n = 1;
  Grid[d].coord[OH_LOWER] = Grid[d].coord[OH_UPPER] = 0;
  Grid[d].fcoord[OH_LOWER] = Grid[d].fcoord[OH_UPPER] = 0.0;
  Grid[d].size = Grid[d].light.size = Grid[d].light.thresh = 0;
  Grid[d].fsize = Grid[d].light.rfsize
                = Grid[d].light.rfsizeplus = Grid[d].light.fthresh = 0.0;
  Grid[d].gsize = Grid[d].rgsize = 1.0;
}
```

We also check if $N = \prod_{d=0}^{D-1} \mathit{\Pi}_d$, or abort the execution by `errstop()` with a message appropriate to D. The other check is to confirm that $\texttt{bcond}[d][\beta] = \texttt{bc}[d][\beta] \in [b, B+b)$ for all $d \in [0, D)$ and $\beta \in \{0, 1\}$ where $b =$ `bbase` argument of the function which has 0 or 1 for C or Fortran simulator body respectively. If the condition is not satisfied we abort the execution by `errstop()` giving $\texttt{Message.xyz}[d]$ and $\texttt{Message.loup}[\beta]$ to produce an appropriate error message.

```
if (pqr!=nn) {
  if (OH_DIMENSION==1)
    errstop("<# of x-nodes>(%d) should be eqal to <# of nodes>(%d)",
```

```
            pcoord[0], nn);
    else if (OH_DIMENSION==2)
      errstop("<# of x-nodes>(%d) * <# of y-nodes>(%d) "
              "should be eqal to <# of nodes>(%d)",
              pcoord[0], pcoord[1], nn);
    else
      errstop("<# of x-nodes>(%d) * <# of y-nodes>(%d) * <# of z-nodes>(%d) "
              "should be eqal to <# of nodes>(%d)",
              pcoord[0], pcoord[1], pcoord[2], nn);
  }
  for (d=0; d<OH_DIMENSION; d++) {
    for (lu=OH_LOWER; lu<=OH_UPPER; lu++) {
      if (bc[d][lu]<bbase || bc[d][lu]>=nb+bbase)
        errstop("system's %s boundary condition for %c-coordinate %d is "
                "invalid",
                Message.loup[lu], Message.xyz[d], bc[d][lu]);
    }
  }
```

The last operation is to fill $\texttt{*sdoms}[N][D][\beta] = \texttt{sd}[N][D][\beta] = \{\delta_d^{\{l,u\}}(m))\}$ and $\texttt{*bounds}[N][D][\beta] = \texttt{bd}[N][D][\beta]$ as follows.

$$
\begin{aligned}
m &= rank(\pi_0(m), \ldots, \pi_{D-1}(m)) \\
m_d^- &= rank(\pi_0(m), \ldots, \pi_d(m){-}1, \ldots, \pi_{D-1}(m)) \\
\delta_d^l(m) &= \begin{cases} \Delta_d^l & \pi_d(m) = 0 \\ \delta_d^u(m_d^-) & \pi_d(m) > 0 \end{cases} \\
\delta_d^u(m) &= \delta_d^l(m) + \begin{cases} \delta_d^{\min} & \pi_d(m) < \Pi_d^- \\ \delta_d^{\min} + 1 & \pi_d(m) \geq \Pi_d^- \end{cases} \\
\texttt{bd}[m][d][0] &= \begin{cases} \texttt{bc}[d][0] & \pi_d(m) = 0 \\ b & \pi_d(m) > 0 \end{cases} \\
\texttt{bd}[m][d][1] &= \begin{cases} \texttt{bc}[d][1] & \pi_d(m) = \Pi_d - 1 \\ b & \pi_d(m) < \Pi_d - 1 \end{cases}
\end{aligned}
$$

The definitions of $\delta_d^l(m)$ and $\delta_d^u(m)$ above are corresponding to the implementation but are different from those shown in §3.6.1. Their equivalence is, however, proved as follows.

$$
\begin{aligned}
\overline{\delta_d}(\pi) &= \begin{cases} \delta_d^{\min} & \pi < \Pi_d^- \\ \delta_d^{\min} + 1 & \pi \geq \Pi_d^- \end{cases} \\
\delta_d^u(m) &= \overline{\delta_d}(\pi_d(m)) + \delta_d^l(m) = \overline{\delta_d}(\pi_d(m)) + \begin{cases} \Delta_d^l & \pi_d(m) = 0 \\ \delta_d^u(m_d^-) & \pi_d(m) > 0 \end{cases} \\
&= \Delta_d^l + \sum_{\pi=0}^{\pi_d(m)} \overline{\delta_d}(\pi) = \Delta_d^l + \sum_{\pi=0}^{\pi_d(m)} \delta_d^{\min} + \begin{cases} 0 & \pi_d(m) < \Pi_d^- \\ \pi_d(m) + 1 - \Pi_d^- & \pi_d(m) \geq \Pi_d^- \end{cases} \\
&= \Delta_d^l + (\pi_d(m) + 1)\delta_d^{\min} + \max(0,\ \pi_d(m) + 1 - \Pi_d^-) \\
\delta_d^l(m) &= \Delta_d^l + \pi_d(m)\delta_d^{\min} + \max(0,\ \pi_d(m) - \Pi_d^-) \\
&= \Delta_d^l + \begin{cases} \pi_d(m)\delta_d^{\min} & \pi_d(m) \leq \Pi_d^- \\ \pi_d(m)\delta_d^{\min} + (\pi_d(m) - \Pi_d^-) & \pi_d(m) > \Pi_d^- \end{cases} \\
m_d^+ &= rank(\pi_0(m), \ldots, \pi_d(m){+}1, \ldots, \pi_{D-1}(m)) \\
\delta_d^u(m) &= \delta_d^l(m_d^+) \qquad (\pi_d(m) < \Pi_d - 1)
\end{aligned}
$$

$$
\begin{aligned}
\delta_d^u(m) &= \Delta_d^l + \Pi_d \delta_d^{\min} + \max(0,\ \Pi_d - \Pi_d^-) \\
&= \Delta_d^l + \Pi_d \lfloor (\Delta_d^u - \Delta_d^l)/\Pi_d \rfloor + \Pi_d - (\Pi_d - ((\Delta_d^u - \Delta_d^l) \bmod \Pi_d)) \\
&= \Delta_d^l + (\Delta_d^u - \Delta_d^l) - ((\Delta_d^u - \Delta_d^l) \bmod \Pi_d) + ((\Delta_d^u - \Delta_d^l) \bmod \Pi_d) \\
&= \Delta_d^u \qquad (\pi_d(m) = \Pi_d - 1)
\end{aligned}
$$

$$
\delta_d^u(m) = \begin{cases} \delta_d^l(m_d^+) & \pi_d(m) < \Pi_d - 1 \\ \Delta_d^u & \pi_d(m) = \Pi_d - 1 \end{cases}
$$

```
  for (i=0,z=Grid[OH_DIM_Z].coord[OH_LOWER],n=0; i<Grid[OH_DIM_Z].n; i++) {
    int bot=z, top=z+Grid[OH_DIM_Z].light.size;
    int bzlo = i==0 && OH_DIMENSION>OH_DIM_Z ?
                bc[OH_DIM_Z][OH_LOWER] : bbase;
    int bzup = i==Grid[OH_DIM_Z].n-1  && OH_DIMENSION>OH_DIM_Z ?
                bc[OH_DIM_Z][OH_UPPER] : bbase;
    if (i>=Grid[OH_DIM_Z].light.n)  top++;
    z = top;
    for (j=0,y=Grid[OH_DIM_Y].coord[OH_LOWER]; j<Grid[OH_DIM_Y].n; j++) {
      int south=y, north=y+Grid[OH_DIM_Y].light.size;
      int bylo = j==0 && OH_DIMENSION>OH_DIM_Y ?
                  bc[OH_DIM_Y][OH_LOWER] : bbase;
      int byup = j==Grid[OH_DIM_Y].n-1  && OH_DIMENSION>OH_DIM_Y ?
                  bc[OH_DIM_Y][OH_UPPER] : bbase;
      if (j>=Grid[OH_DIM_Y].light.n)  north++;
      y = north;
      for (k=0,x=Grid[OH_DIM_X].coord[OH_LOWER]; k<Grid[OH_DIM_X].n; k++,n++) {
        int west=x, east=x+Grid[OH_DIM_X].light.size;
        if (k>=Grid[OH_DIM_X].light.n)  east++;
        x = east;
        sd[n][OH_DIM_X][OH_LOWER] = west;
        sd[n][OH_DIM_X][OH_UPPER] = east;
        bd[n][OH_DIM_X][OH_LOWER] = bd[n][OH_DIM_X][OH_UPPER] = bbase;
        if (OH_DIMENSION>OH_DIM_Y) {
          sd[n][OH_DIM_Y][OH_LOWER] = south;
          sd[n][OH_DIM_Y][OH_UPPER] = north;
          bd[n][OH_DIM_Y][OH_LOWER] = bylo;
          bd[n][OH_DIM_Y][OH_UPPER] = byup;
        }
        if (OH_DIMENSION>OH_DIM_Z) {
          sd[n][OH_DIM_Z][OH_LOWER] = bot;
          sd[n][OH_DIM_Z][OH_UPPER] = top;
          bd[n][OH_DIM_Z][OH_LOWER] = bzlo;
          bd[n][OH_DIM_Z][OH_UPPER] = bzup;
        }
      }
      bd[n-Grid[OH_DIM_X].n][OH_DIM_X][OH_LOWER] = bc[OH_DIM_X][OH_LOWER];
      bd[n-1][OH_DIM_X][OH_UPPER] = bc[OH_DIM_X][OH_UPPER];
    }
  }
}
```

4.7.6 init_subdomain_passively()

init_subdomain_passively()

The function init_subdomain_passively(), called solely from init3(), initialize Grid[3] and SubDomainDesc[N] after allocating it by mem_alloc(), referring to *sdoms[N][D][2] = sd[N][D][2] = $\{\delta_d^{\{l,u\}}(m)\}$. It also checks the consistency of sd[][][], *bounds[N][D][2] = bd[N][D][2], nb = B and bbase = $b \in \{0,1\}$.

```
static void
init_subdomain_passively(int (*sd)[OH_DIMENSION][2],
                         int (*bd)[OH_DIMENSION][2], int nb, int bbase) {
  int nn=nOfNodes;
  struct S_subdomdesc *sdd = SubDomainDesc =
    (struct S_subdomdesc*)mem_alloc(sizeof(struct S_subdomdesc), nn,
                                    "SubDomainDesc");
  int min[OH_DIMENSION], max[OH_DIMENSION];
  int smin[OH_DIMENSION], smax[OH_DIMENSION];
  int me=myRank;
  int i, d, dd, lu, l;
  int lo[OH_DIMENSION-1], up[OH_DIMENSION-1], h[OH_DIMENSION-1];
```

First we copy sd[m][d][β] = $\delta_d^\beta(m)$ to SubDomainDesc[m].coord[d].c[β] and SubDomainDesc[m].coord[d].fc[β], and then calculate $\Delta_d^l = \min_m\{\delta_d^l(m)\}$, $\Delta_d^u = \max_m\{\delta_d^u(m)\}$, $\delta_d^{\min} = \min_m\{\delta_d^u(m) - \delta_d^l(m)\}$ and $\delta_d^{\max} = \max_m\{\delta_d^u(m) - \delta_d^l(m)\}$. We also initialize SubDomainDesc[m].coord[d].n = 0 for all $m \in [0,N)$ and $d \in [0,D)$ to make them have a some specific value (but not being referred to) because the set-up opreation discussed later will leave them unchanged for $m \neq$ SubDomainDesc[m].h[d]. The other but necessary initialization is SubDomainDesc[m].id = m for all $m \in [0,N)$ to keep the subdomain identity after the sorting also discussed later. In addition we check if $\delta_d^l(m) < \delta_d^u(m)$, and if bd[$m$][$d$][$\beta$] $\in [b, B+b)$ where $b = 0$ for C or $b = 1$ for Fortran simulator body, or abort the execution by errstop() giving Message.xyz[d] and Message.loup[β] to produce appropriate error messages.

```
  for (d=0; d<OH_DIMENSION; d++) {
    min[d] = sd[0][d][OH_LOWER];  max[d] = sd[0][d][OH_UPPER];
    smin[d] = smax[d] = max[d] - min[d];
  }
  for (i=0; i<nn; i++) {
    for (d=0; d<OH_DIMENSION; d++) {
      int lo=sd[i][d][OH_LOWER], up=sd[i][d][OH_UPPER], n=up-lo;
      sdd[i].coord[d].fc[OH_LOWER] = (sdd[i].coord[d].c[OH_LOWER] = lo);
      sdd[i].coord[d].fc[OH_UPPER] = (sdd[i].coord[d].c[OH_UPPER] = up);
      sdd[i].coord[d].n = 0;
      if (n<smin[d])  smin[d] = n;
      if (n>smax[d])  smax[d] = n;
      if (lo<min[d])  min[d] = lo;
      if (up>max[d])  max[d] = up;
      if (n<=0)
        errstop("subdomain %d has %c-coordinate lower boundary %d "
                "not less than upper boundary %d", i, Message.xyz[d], lo, up);
      for (lu=OH_LOWER; lu<=OH_UPPER; lu++) {
        if (bd[i][d][lu]<bbase || bd[i][d][lu]>=nb+bbase)
          errstop("rank-%d's %s boundary condition for %c-coordinate %d is "
```

```
                "invalid",
                i, Message.loup[lu], Message.xyz[d], bd[i][d][lu]);
      }
    }
    sdd[i].id = i;
  }
```

Next we set elements of `Grid`$[d]$ for all $d \in [0, D)$ as `size` = `fsize` = $\delta_d^{\max}$, `light.size` = $\delta_d^{\min}$, `coord`$[0]$ = `fcoord`$[0]$ = Δ_d^l, `coord`$[1]$ = `fcoord`$[1]$ = Δ_d^u, `gsize` = $\gamma_d = 1$ and `rgsize` = $1/\gamma_d = 1$. The other elements `n` = Π_d, `light.n` = Π_d^-, `light.thresh` = `light.fthresh` = Δ_d^-, `light.rfsize` = $1/\delta_d^{\min}$ and `light.rfsize` = $1/(\delta_d^{\min} + 1)$ are set to 0 but they are never referred to. Note that if $D < 3$, we set $\Pi_d = \Pi_d^- = 1$, $\Delta_d^l = \Delta_d^u = \Delta_d^- = 0$, $\gamma_d = 1/\gamma_d = 1$, and $\delta_d^{\max} = \delta_d^{\min} = 0$ with their reciprocals for all $d \geq D$, after the loop for $d \in [0, D)$.

Then we check if the boundary coordinates of local node are consistent with those of its neighbors. Let m_d^β be `Adjacent`$[d][\beta]$ the d-th dimensional lower ($\beta = 0$) or upper ($\beta = 1$) neighbor of the local node n. Unless $m_d^\beta = N$ meaning the neighbor does not exist or `bd`$[n][d][\beta] \neq b$ meaning the boundary between the local node and it is special, $\delta_e^\beta(n)$, $\delta_e^\beta(m_d^\beta)$ and $\delta_e^{1-\beta}(m_d^\beta)$ should satisfy the following where $\delta_d^\beta(m) = \delta_d^{\{l,u\}[\beta]}$.

$$\begin{aligned} \delta_d^{1-\beta}(m_d^\beta) = \delta_d^\beta(n) \ &\vee \ \delta_d^{1-\beta}(m_d^\beta) = \delta_d^\beta(n) + (\Delta_d^u - \Delta_d^l) \\ &\vee \ \delta_d^{1-\beta}(m_d^\beta) = \delta_d^\beta(n) - (\Delta_d^u - \Delta_d^l) \\ \delta_e^\gamma(m_d^\beta) = \delta_e^\gamma(n) \qquad & (e \in [0, D) - \{d\},\ \gamma \in \{0, 1\}) \end{aligned}$$

If a condition above is not satisfied, we abort the execution by `local_errstop()` giving it elements of `Message` to produce appropriated error messages.

```
  for (d=0; d<OH_DIMENSION; d++) {
    Grid[d].fsize = (Grid[d].size = smax[i]);
    Grid[d].light.size = smin[d];
    Grid[d].light.rfsize = Grid[d].light.rfsizeplus = 0.0;
    Grid[d].fcoord[OH_LOWER] = (Grid[d].coord[OH_LOWER] = min[d]);
    Grid[d].fcoord[OH_UPPER] = (Grid[d].coord[OH_UPPER] = max[d]);
    Grid[d].n = Grid[d].light.n = 0;    /* never referred but ... */
    Grid[d].light.thresh = 0;  Grid[d].light.fthresh = 0.0;
    Grid[d].gsize = Grid[d].rgsize = 1.0;
    for (lu=OH_LOWER; lu<=OH_UPPER; lu++) {
      int n=Adjacent[d][lu];
      if (n==nn || bd[me][d][lu]!=bbase) continue;
      for (dd=0; dd<OH_DIMENSION; dd++) {
        if (d==dd) {
          int diff = sd[n][dd][OH_UPPER-lu] - sd[me][dd][lu];
          int dsize = max[dd] - min[dd];
          if (diff!=0 && diff!=dsize && diff!=-dsize)
            local_errstop("rank-%d and its %c-%s neighbor rank-%d have "
                          "incompatible %s/%s boundaries of %c-coordinate "
                          "%d and %d",
                          me, Message.xyz[d], Message.loup[lu], n,
                          Message.loup[lu], Message.loup[OH_UPPER-lu],
                          Message.xyz[dd],
                          sd[me][dd][lu], sd[n][dd][OH_UPPER-lu]);
        } else {
```

```
        for (l=OH_LOWER; l<=OH_UPPER; l++) {
          if (sd[n][dd][l]!=sd[me][dd][l])
            local_errstop("rank-%d and its %c-%s neighbor rank-%d have "
                          "incompatible %s boundary of %c-coordinate "
                          "%d and %d",
                          me, Message.xyz[d], Message.loup[lu], n,
                          Message.loup[l], Message.xyz[dd],
                          sd[me][dd][l], sd[n][dd][l]);
        }
      }
    }
  }
}
for (; d<3; d++) {
  Grid[d].n = Grid[d].light.n = 1;
  Grid[d].coord[OH_LOWER] = Grid[d].coord[OH_UPPER] = 0;
  Grid[d].fcoord[OH_LOWER] = Grid[d].fcoord[OH_UPPER] = 0.0;
  Grid[d].size = Grid[d].light.size = Grid[d].light.thresh = 0;
  Grid[d].fsize = Grid[d].light.rfsize
                = Grid[d].light.rfsizeplus = Grid[d].light.fthresh = 0.0;
  Grid[d].gsize = Grid[d].rgsize = 1.0;
}
```

Finally, we sort `SubDomainDesc`$[N]$ and set its elements so that `oh3_map_particle_to_subdomain()` find the subdomain in which a particle resides. The sorting is performed by `qsort()` which compares two elements by `comp_xyz()` which defines the total ordering of nodes with the irreflective relation $m_1 \prec m_2$ for $m_1 \neq m_2$ as follws.

$$
\begin{aligned}
\delta^m_d(m) &= \delta^l_d(m) + \delta^u_d(m) \\
m_1 \overset{d}{=} m_2 &\Leftrightarrow \delta^l_d(m_1) = \delta^l_d(m_2) \ \wedge \ \delta^u_d(m_1) = \delta^u_d(m_2) \\
m_1 \overset{d}{\prec} m_2 &\Leftrightarrow \delta^m_d(m_1) < \delta^m_d(m_2) \ \vee \ (\delta^m_d(m_1) = \delta^m_d(m_2) \ \wedge \ \delta^l_d(m_1) < \delta^l_d(m_2)) \ \vee \\
&\qquad (\delta^m_d(m_1) = \delta^m_d(m_2) \ \wedge \ \delta^l_d(m_1) = \delta^l_d(m_2) \ \wedge \ \delta^u_d(m_1) < \delta^u_d(m_2)) \qquad (d < D) \\
m_1 \overset{D}{\prec} m_2 &\Leftrightarrow m_1 < m_2 \\
m_1 \prec m_2 &\Leftrightarrow \exists d \in [0, D] : \bigwedge_{e=0}^{d-1} (m_1 \overset{e}{=} m_2) \ \wedge \ m_1 \overset{d}{\prec} m_2
\end{aligned}
$$

The definition above assures that, if $D = 3$, for a *wall* (or a set of them) of subdomains which share $\delta^l_0(m)$ and $\delta^u_0(m)$, the members in it constitutes a sequence in the sorted `SubDomainDesc[]`. It is also assured that for a *pillar* (or a set of them) of subdomains in a wall (set) which also share $\delta^l_1(m)$ and $\delta^u_1(m)$, the members in it constitues a sequence. Therefore, we let `SubDomainDesc`$[m]$`.coord`$[d]$`.h` have the *head* of the wall ($d = 0$) and pillar ($d = 1$) which are the first members of the wall/pillar to which the m-th subdomain belongs to. We also let `SubDomainDesc`$[h]$`.coord`$[d]$`.n` have the number of members in a wall/pillar whose head is h. More specifically, we let `h` and `n` of `SubDomainDesc`$[m]$`.coord`$[d]$ for all $m \in [0, N)$ and $d \in [0, D-1)$ be the followings.

$$
\begin{aligned}
& i(m) = \texttt{SubDomainDesc}[m]\texttt{.id} \\
& M_d(m) = \{k \,|\, \forall e \leq d \,:\, \delta^l_e(i(k)) = \delta^l_e(i(m)),\ \delta^u_e(i(k)) = \delta^u_e(i(m))\} \\
& \texttt{SubDomainDesc}[m]\texttt{.coord}[d]\texttt{.h} = \min(M_d(m))
\end{aligned}
$$

$$\texttt{SubDomainDesc}[m]\texttt{.coord}[d]\texttt{.n} = \begin{cases} |M_d(m)| & m = \min(M_d(m)) \\ 0 & m \neq \min(M_d(m)) \end{cases}$$

For the setting above, we scan `SubDomainDesc[]` keeping track $h_d = \min(M_d(m))$ in $\texttt{h}[d]$ and $\delta_d^{\{l,u\}}(i(h_d))$ in $\{\texttt{lo},\texttt{up}\}[d]$ with $h_d = 0$ at initial. Then each time we find m such that $\delta_d^l(i(m)) \neq \texttt{lo}[d]$ or $\delta_d^u(i(m)) \neq \texttt{up}[d]$ and thus the head of a new wall/pillar, for all e such that $e \in [d, D-1)$ we let `n` of the current d-th dimensional head h_e be $|M_d(h_e)| = m - h_e$, and then $h_e \leftarrow m$, $\texttt{lo}[e] \leftarrow \delta_e^l(i(m))$ and $\texttt{up}[e] \leftarrow \delta_e^u(i(m))$. On the other hand, if $\delta_e^l(i(m)) = \texttt{lo}[e]$ and $\delta_e^u(i(m)) = \texttt{up}[e]$ hold for all $e \in [0, d]$, we let `h` of m be h_e.

On the other hand, we let $\texttt{h} = m$ and $\texttt{n} = 1$ for $\texttt{SubDomainDesc}[m]\texttt{.coord}[D-1]$ for all $m \in [0, N)$ because a subdomain in a pillar will not (and should not) share its $(D-1)$-th boundary coordinates with other subdomains in the pillar.

```
  qsort(sdd, nn, sizeof(struct S_subdomdesc), comp_xyz);
  for (d=0; d<OH_DIMENSION-1; d++) {
    sdd[0].coord[d].h = h[d] = 0;
    lo[d] = sdd[0].coord[d].c[OH_LOWER];
    up[d] = sdd[0].coord[d].c[OH_UPPER];
  }
  for (i=1; i<nn; i++) {
    for (d=0; d<OH_DIMENSION-1; d++) {
      if (lo[d]!=sdd[i].coord[d].c[OH_LOWER] ||
          up[d]!=sdd[i].coord[d].c[OH_UPPER]) {
        for (dd=d; dd<OH_DIMENSION-1; dd++) {
          sdd[h[dd]].coord[dd].n = i - h[dd];
          sdd[i].coord[dd].h = h[dd] = i;
          lo[dd] = sdd[i].coord[dd].c[OH_LOWER];
          up[dd] = sdd[i].coord[dd].c[OH_UPPER];
        }
        break;
      } else {
        sdd[i].coord[d].h = h[d];
      }
    }
    sdd[i].coord[OH_DIMENSION-1].n = 1;  sdd[i].coord[OH_DIMENSION-1].h = i;
  }
  for (d=0; d<OH_DIMENSION-1; d++)  sdd[h[d]].coord[d].n = nn - h[d];
}
```

4.7.7 `comp_xyz()`

`comp_xyz()` The function `comp_xyz()`, called solely from `qsort()` called in `init_subdomain_passively()`, compares two elements of $\texttt{SubDomainDesc}[N]$ pointed by its arguments `aa` and `bb` to return -1 if $m_a \prec m_b$ or 1 otherwise where $\texttt{*aa} = \texttt{SubDomainDesc}[m_a]$, $\texttt{*bb} = \texttt{SubDomainDesc}[m_b]$ and $\prec$ is the irreflective relation defined in §4.7.6.

```
static int
comp_xyz(const void* aa, const void* bb) {
  struct S_subdomdesc *a=(struct S_subdomdesc*)aa, *b=(struct S_subdomdesc*)bb;
  int d;

  for (d=0; d<OH_DIMENSION; d++) {
```

```
    if (a->coord[d].c[OH_LOWER]+a->coord[d].c[OH_UPPER]<
        b->coord[d].c[OH_LOWER]+b->coord[d].c[OH_UPPER])  return(-1);
    if (a->coord[d].c[OH_LOWER]+a->coord[d].c[OH_UPPER]>
        b->coord[d].c[OH_LOWER]+b->coord[d].c[OH_UPPER])  return(1);
    if (a->coord[d].c[OH_LOWER]<b->coord[d].c[OH_LOWER])  return(-1);
    if (a->coord[d].c[OH_LOWER]>b->coord[d].c[OH_LOWER])  return(1);
    if (a->coord[d].c[OH_UPPER]<b->coord[d].c[OH_UPPER])  return(-1);
    if (a->coord[d].c[OH_UPPER]>b->coord[d].c[OH_UPPER])  return(1);
  }
  return(a->id<b->id ? -1 : 1);
}
```

4.7.8 Macro `Field_Disp()`

`Field_Disp()` The macro `Field_Disp`(f, i_0, i_1, i_2), used in `init_fields()`, `set_field_descriptors()` and `set_border_comm()`, is replaced with the one-dimensional index $a = fdisp(f, i_0, i_1, i_2)$ of $[i_2][i_1][i_0][0]$ in an array of $[\Phi_2(f)][\Phi_1(f)][\Phi_0(f)][\varepsilon(f)]$ where $\varepsilon(f) =$ `FieldDesc`$[f]$`.esize` and $\Phi_d(f) =$ `FieldDesc`$[f]$`.size`$[d]$, providing $i_d = 0$ and $\Phi_d(f) = 1$ for all $d \geq D$, as follows.

$$a_{D-1} = i_{D-1} \qquad a_d = a_{d+1} \cdot \Phi_d(f) + i_d \qquad a = a_0 \cdot \varepsilon(f)$$

```
#if OH_DIMENSION==1
#define Field_Disp(F,X,Y,Z) (FieldDesc[F].esize * (X))
#elif OH_DIMENSION==2
#define Field_Disp(F,X,Y,Z)\
  (FieldDesc[F].esize *\
   ((X) + FieldDesc[F].size[OH_DIM_X] * (Y)))
#else
#define Field_Disp(F,X,Y,Z)\
  (FieldDesc[F].esize *\
   ((X) + FieldDesc[F].size[OH_DIM_X] *\
    ((Y) + FieldDesc[F].size[OH_DIM_Y] * (Z))))
#endif
```

4.7.9 `init_fields()`

`init_fields()` The function `init_fields()`, called solely from `init3()`, makes the substances of `FieldTypes`$[F][7]$, `BoundaryCommFields`$[C]$ and `BoundaryCommFields`$[C][B][2][3]$, by copying the contents of their shadows given through its and `oh3_init()`'s arguments `ft`$[][] =$ `ftypes`$[][]$, `cf`$[] =$ `cfields`$[]$ and `ct`$[][][] =$ `ctypes`$[][][]$ to them, referring to its and `init3()`'s argument `cfid` being 0 for C or -1 for Fortan simulator body. Then the funcion creates and initializes `FieldDesc`$[F]$, `BorderExc`$[C][2][D][2]$ and the array pointed by `*fsizes` argument of the function and `oh3_init()`, referring to its and `oh3_init()`'s argument `nb` = `nbound` $= B$.

```
static void
init_fields(int (*ft)[OH_FTYPE_N], int *cf, int cfid, int (*ct)[2][OH_CTYPE_N],
            int nb, int sd[OH_DIMENSION][2], int **fsizes) {
  struct S_flddesc *fd;
  struct S_borderexc (*bx)[2][OH_DIMENSION][2];
```

```
  int (*fs)[OH_DIMENSION][2]=(int(*)[OH_DIMENSION][2])*fsizes;
  int nf, ne;
  int f, e, b, d, lu, i, *tmp;
```

First, we set the array size variable `nOfBoundaries` $= B$, as well as `nOfFields` $= F$ being the number of leading elements of `ft`[][] having positive $\varepsilon(f)$, and `nOfExc` $= C$ being non-negative (`cfid` $= 0$) or positive (`cfid` $= -1$) leading elements of `cf`[]. Note that the terminator of `cf`[] is negative (-1) regardless of `cfid` if `OH_POS_AWARE` is defined to mean `cf`[] is `BoundaryCommFields`[] allocated and initialized by level-4p initializer `init4p()` and thus have zero-origin indices of `ft`[][] = `FieldTypes`[].

```
  nOfBoundaries = nb;
  for (nf=0; ft[nf][OH_FTYPE_ES]>0; nf++);
  nOfFields = nf;
#ifdef OH_POS_AWARE
  for (ne=0; cf[ne]>=0; ne++);
#else
  for (ne=0; cf[ne]+cfid>=0; ne++);
#endif
  nOfExc = ne;
```

Next, we allocate `FieldDesc`[F] by `mem_alloc()`. We also allocate substances of `FieldTypes`[F][7], `BoundaryCommTypes`[C][B][2][3] and `BoundaryCommFields`[C], whose elements are then copied from their shadows `ft`[][], `ct`[][][][] and `cf`[] by `memcpy()` for first two and by an explict for-loop for the last to make its elements zero-origined, unless `OH_POS_AWARE` is defined to mean they are allocated and initialized by level-4p initializer `init4p()` with one additional element for each. The other allocation takes place for `*fsizes`[F][D][2] to have $\phi_d^{\{l,u\}}(f)$ if the double pointer points `NULL`.

```
  FieldDesc = fd = (struct S_flddesc*)mem_alloc(sizeof(struct S_flddesc), nf,
                                                "FieldDesc");
#ifndef OH_POS_AWARE
  FieldTypes = (int(*)[OH_FTYPE_N])
               mem_alloc(sizeof(int), nf*OH_FTYPE_N, "FieldTypes");
  BoundaryCommTypes  = (int(*)[2][OH_CTYPE_N])
                       mem_alloc(sizeof(int), ne*nb*2*OH_CTYPE_N,
                                 "BoundaryCommTypes");
  memcpy(FieldTypes, ft, sizeof(int)*nf*OH_FTYPE_N);
  memcpy(BoundaryCommTypes, ct, sizeof(int)*ne*nb*2*OH_CTYPE_N);
  ft = FieldTypes;  ct = BoundaryCommTypes;

  tmp = (int*)mem_alloc(sizeof(int), ne, "BoundaryCommFields");
  for (e=0; e<nOfExc; e++)  tmp[e] = cf[e] + cfid;
  BoundaryCommFields = cf = tmp;
#endif

  if (!fs)
    fs = (int(*)[OH_DIMENSION][2])
         (*fsizes = (int*)mem_alloc(sizeof(int), nf*OH_DIMENSION*2,
                                    "FieldSizes"));
```

Next we scan `FieldTypes`$[F][7]$ to set `FieldDesc`$[f]$`.esize` $= \varepsilon(f)$, and temporalily `FieldDesc`$[f]$`.ext`$[0]$ $= e_l^{\min}(f) = \min(e_l(f), e_l^b(f), e_l^r(f))$ and `FieldDesc`$[f]$`.ext`$[1]$ $= e_u^{\max}(f) = \max(e_u(f), e_u^b(f), e_u^r(f))$.

```
for (f=0; f<nf; f++) {
  int lo=ft[f][OH_FTYPE_LO], up=ft[f][OH_FTYPE_UP];
  fd[f].esize = ft[f][OH_FTYPE_ES];
  for (lu=OH_FTYPE_BL; lu<OH_FTYPE_RU; lu+=2) {
    int lot=ft[f][lu], upt=ft[f][lu+1];
    if (lot<lo)  lo = lot;
    if (upt>up)  up = upt;
  }
  fd[f].ext[OH_LOWER] = lo;  fd[f].ext[OH_UPPER] = up;
}
```

Next we scan `BoundaryCommTypes`$[C][B][2][3]$ to calculate $e^{\gamma}_{\{l,u\}}(f)$ as follows and then let $e_l^{\min}(f) \leftarrow \min(e_l^{\min}(f), e_l^{\gamma}(f))$ and $e_u^{\max}(f) \leftarrow \max(e_u^{\max}(f), e_u^{\gamma}(f))$.

$$
\begin{aligned}
\Gamma(f) &= \{c \,|\, \texttt{BoundaryCommFields}[c] = f\} \\
\lambda(e,s) &= \begin{cases} e & s \neq 0 \\ 0 & s = 0 \end{cases} \\
s^{\downarrow}(b,c) &= \texttt{BoundaryCommTypes}[c][b][0][2] \\
s^{\uparrow}(b,c) &= \texttt{BoundaryCommTypes}[c][b][1][2] \\
e_f^{\downarrow}(b,c) &= \lambda(\texttt{BoundaryCommTypes}[c][b][0][0], s^{\downarrow}(b,c)) \\
e_t^{\downarrow}(b,c) &= \lambda(\texttt{BoundaryCommTypes}[c][b][0][1], s^{\downarrow}(b,c)) \\
e_f^{\uparrow}(b,c) &= \lambda(\texttt{BoundaryCommTypes}[c][b][1][0], s^{\uparrow}(b,c)) \\
e_t^{\uparrow}(b,c) &= \lambda(\texttt{BoundaryCommTypes}[c][b][1][1], s^{\uparrow}(b,c)) \\
e_l^{\gamma}(f) &= \min_{b\in[0,B),c\in\Gamma(f)} (\{e_f^{\downarrow}(b,c)\} \cup \{e_t^{\uparrow}(b,c)\}) \\
e_u^{\gamma}(f) &= \max_{b\in 0B,c\in\Gamma(f)} (\{e_t^{\downarrow}(b,c) + s^{\downarrow}(b,c)\} \cup \{e_f^{\uparrow}(b,c) + s^{\uparrow}(b,c)\})
\end{aligned}
$$

That is, $e_l^{\gamma}(f)$ is the minimum local coordinate among the bottom of non-empty sending boundary plane sets for downward communication and that of receiving counterparts for upward, for all boundary conditions and for all communication types for the field-array f. Similarly, $e_u^{\gamma}(f) - 1$ is the maximum local coordinate among the top of non-empty sending boundary plane sets for upward communication and that of receiving counterparts for downward, for all boundary condtions and for all communication types for the field-array f.

In the scan of `BoundaryCommTypes`$[c]$ for all $c \in [0, C)$, we check if `BoundaryCommFields`$[c] < F$ or abort the execution by `errstop()`.

```
for (e=0,i=0; e<ne; e++) {
  int f=cf[e];
  int lo, up;
  if (f>=nf)
    errstop("boundary communication #%d cannot be defined for "
            "undefined field #%d", e-cfid, f-cfid);
  lo = fd[f].ext[OH_LOWER];  up = fd[f].ext[OH_UPPER];
  for (b=0; b<nb; b++,i++) {
```

```
    int sl=ct[i][OH_LOWER][OH_CTYPE_SIZE];
    int su=ct[i][OH_UPPER][OH_CTYPE_SIZE];
    int lo1=ct[i][OH_LOWER][OH_CTYPE_FROM];
    int lo2=ct[i][OH_UPPER][OH_CTYPE_TO];
    int up1=ct[i][OH_LOWER][OH_CTYPE_TO]   + sl;
    int up2=ct[i][OH_UPPER][OH_CTYPE_FROM] + su;
    if (sl && lo1<lo)  lo = lo1;
    if (su && lo2<lo)  lo = lo2;
    if (sl && up1>up)  up = up1;
    if (su && up2>up)  up = up2;
  }
  fd[f].ext[OH_LOWER] = lo;  fd[f].ext[OH_UPPER] = up;
}
```

Next, we do the followings to define the required size of field-array $f \in [0, F)$ for all its dimensions $d \in [0, D)$.

$$\begin{aligned} \texttt{*fsizes}[f][d][0] &= \phi_d^l(f) = e_l^{\min}(f) \\ \texttt{*fsizes}[f][d][1] &= \phi_d^u(f) = \delta_d^{\max} + e_u^{\max}(f) + \texttt{cfid} \\ \texttt{FieldDesc}[F]\texttt{.size[d]} &= \Phi_d(f) = \delta_d^{\max} + (e_u^{\max}(f) - e_l^{\min}(f)) \end{aligned}$$

Note that we add $\texttt{cfid} \in \{0, -1\}$ to give the upper limit of the field-array to a Fortan simulator body while the positive extent is given to a C simulator body.

```
for (f=0; f<nf; f++) {
  int lo=fd[f].ext[OH_LOWER], up=fd[f].ext[OH_UPPER];
  for (d=0; d<OH_DIMENSION; d++) {
    fs[f][d][OH_LOWER] = lo;
    fs[f][d][OH_UPPER] = (Grid[d].size+cfid) + up;
    fd[f].size[d] = Grid[d].size + (up - lo);
  }
}
```

Next, we let $\texttt{FieldDesc}[f]\texttt{.}\{\texttt{bc},\texttt{red}\}\texttt{.base}$ be $fdisp(f, e_l^x(f), e_l^x(f), e_l^x(f))$ where $x \in \{b, r\}$ for all $f \in [0, F)$. We also set $\texttt{FieldDesc}[f]\texttt{.}\{\texttt{bc},\texttt{red}\}\texttt{.size}[0]$ for the primary domain by `set_field_descriptors()` giving it the argument of this function $\texttt{ft}[][] = \texttt{*ftypes}[][]$ and $\texttt{sd}[][] = \texttt{SubDomains}[n][][]$ for the local node n, letting its argument $\texttt{ps} = 0$ to indicate that the target subdomain is primary.

```
for (f=0; f<nf; f++) {
  int bl = ft[f][OH_FTYPE_BL];
  int rl = ft[f][OH_FTYPE_RL];
  fd[f].bc.base  = Field_Disp(f, bl, bl, bl);
  fd[f].red.base = Field_Disp(f, rl, rl, rl);
}
set_field_descriptors(ft, sd, 0);
```

Finally, we allocate $\texttt{BorderExc}[C][2][D][2]$ and set its elements $[c][0][d][\beta]$ of the primary subdomain for all $c \in [0, C)$, $d \in [0, D)$ and $\beta \in \{0, 1\}$ by `set_border_exchange()` giving it c and letting its argument $\texttt{ps} = 0$ to indicate that the target subdomain is primary. Note that the last argument `type` is usually `MPI_DOUBLE` but can be `MPI_LONG_LONG_INT` if `OH_POS_AWARE` is defined to mean position-aware particle management is in effect and $c = C - 1$ for the per-grid histogramrather than user-defined field-array. We also initialize

`BorderExc`$[c][1][d][\beta]$ for the secondary subdomain by `clear_border_exchange()` setting $\{$`send`, `recv`$\}$`.deriv` $= 0$ before calling it to keep it from freeing the undefined data-type in $\{$`send`, `recv`$\}$`.type`.

```
  BorderExc = bx =
    (struct S_borderexc(*)[2][OH_DIMENSION][2])
    mem_alloc(sizeof(struct S_borderexc), ne*2*OH_DIMENSION*2, "BorderExc");

  for (e=0; e<ne; e++) {
    for (d=0; d<OH_DIMENSION; d++) {
      for (lu=0; lu<2; lu++)
        bx[e][1][d][lu].send.deriv = bx[e][1][d][lu].recv.deriv = 0;
    }
#ifdef OH_POS_AWARE
    set_border_exchange(e, 0, e<ne-1 ? MPI_DOUBLE : MPI_LONG_LONG_INT);
#else
    set_border_exchange(e, 0, MPI_DOUBLE);
#endif
  }
  clear_border_exchange();
}
```

4.7.10 `set_field_descriptors()`

`set_field_descriptors()` The function `set_field_descriptors()`, called from `init_fields()` and `transbound3()`, sets `FieldDesc`$[f]$`.{bc,red}.size`$[p] = \sigma(f, \{b, r\}, m)$ for all $f \in [0, F)$ where $p =$ `ps` argument of the function to indicate primary subdomain $m = n$ if $p = 0$, or secondary subdomain $m = \mathit{parent}(n)$ otherwise, for the local node n. The function refers $\varepsilon(f)$, $e^b_u(f)$ and $e^r_u(f)$ through its argument `ft`$[F][7] =$ `FieldTypes`$[F][7]$, and $\delta^l_d(m)$ and $\delta^l_d(m)$ through the argument `sd`$[D][2] =$ `SubDomains`$[m][D][2]$. The value of $\sigma(f, \{b, r\}, m)$ is calculated by `Field_Disp()` as follows.

$$\upsilon_d(m) = \begin{cases} \delta^u_d(m) - \delta^l_d(m) + e^{\{b,r\}}_u(f) - 1 & d < D \\ 0 & d \geq D \end{cases}$$

$$\sigma(f, \{b, r\}, m) = (\mathit{fdisp}(f, \upsilon_0(m), \upsilon_1(m), \upsilon_2(m)) + \varepsilon(f) - \texttt{FieldDesc}[f].\{\texttt{bc}, \texttt{red}\}.\texttt{base}$$

Note that we use `size`$[3]$ for $\upsilon_d(m)$ above.

```
void
set_field_descriptors(int (*ft)[OH_FTYPE_N], int sd[OH_DIMENSION][2], int ps) {

  int nf=nOfFields;
  struct S_flddesc *fd = FieldDesc;
  int size[3] = {0,0,0};
  int d, f;

  for (d=0; d<OH_DIMENSION; d++)  size[d] = sd[d][OH_UPPER] - sd[d][OH_LOWER];
  for (f=0; f<nf; f++) {
    int bu = ft[f][OH_FTYPE_BU] - 1;
    int ru = ft[f][OH_FTYPE_RU] - 1;
    int es = ft[f][OH_FTYPE_ES];
    fd[f].bc.size[ps] =
```

```
      Field_Disp(f, size[OH_DIM_X]+bu, size[OH_DIM_Y]+bu, size[OH_DIM_Z]+bu) -
      fd[f].bc.base + es;
    fd[f].red.size[ps] =
      Field_Disp(f, size[OH_DIM_X]+ru, size[OH_DIM_Y]+ru, size[OH_DIM_Z]+ru) -
      fd[f].red.base + es;
  }
}
```

4.7.11 set_border_exchange()

set_border_exchange()

The function set_border_exchange(), called from init_fields() and oh3_exchange_borders(), fill the elements in BorderExc[c][p][2], where c and p are given through its arguments e and ps, for a communication type $c \in [0, C)$ of the field-array $f =$ BoundaryCommFields[c] of the primary ($p = 0$) or secondary subdomain m of the local node, i.e., $m =$ RegionId[p]. The MPI data types to be recorded in BorderExc[c][p][][] is given by type argument or that derived from it, which is usually MPI_DOUBLE but can be MPI_LONG_LONG_INT for per-grid histogram used in level-4p library.

```
static void
set_border_exchange(int e, int ps, MPI_Datatype type) {
  struct S_borderexc (*bx)[2] = BorderExc[e][ps];
  int f = BoundaryCommFields[e];
  int nb = nOfBoundaries;
  int (*bt)[2][OH_CTYPE_N] = &BoundaryCommTypes[e*nb];
  int (*bd)[2] = Boundaries[RegionId[ps]];
  int (*sd)[2] = SubDomains[RegionId[ps]];
  struct S_flddesc *fd = &FieldDesc[f];
  int esize = fd->esize;
  int fext = fd->ext[OH_UPPER] - fd->ext[OH_LOWER];
  int xyz[3] = {
    sd[OH_DIM_X][OH_UPPER]-sd[OH_DIM_X][OH_LOWER],
    OH_DIMENSION>OH_DIM_Y ? sd[OH_DIM_Y][OH_UPPER]-sd[OH_DIM_Y][OH_LOWER] : 0,
    OH_DIMENSION>OH_DIM_Z ? sd[OH_DIM_Z][OH_UPPER]-sd[OH_DIM_Z][OH_LOWER] : 0
  };
  int *wdh = fd->size;
  int exti[OH_DIMENSION][2], exto[OH_DIMENSION][2];
  int soff[OH_DIMENSION][2], roff[OH_DIMENSION][2];
  int ssize[OH_DIMENSION][2], rsize[OH_DIMENSION][2];
  int d, lu;
```

First, it fills the following argument arrays for set_border_comm(), for all $d \in [0, D)$, $\beta \in \{0, 1\}$ and $w \in \{0, 1\}$.

- exti[d][β] is the d-th dimensional *inner extension* of the field-array f being the bottom ($\beta = 0$) or top-plus-one ($\beta = 1$) coordinate of the sending planes relative to the bottom or top boundary plane.
- exto[d][β] is the d-th dimensional *outer extension* of the field-array f being the bottom ($\beta = 0$) or top-plus-one ($\beta = 1$) coordinate of the receiving planes relative to the bottom or top boundary plane.

- `soff`$[d][w]$ and `roff`$[d][w]$ are the bottoms of the d-th dimensional sending/receiving planes of the field-array f to be sent/received in downward ($w = 0$) or upward ($w = 1$) communication.
- `ssize`$[d][w]$ and `rsize`$[d][w]$ are the number of the d-th dimensional sending/receiving planes of the field-array f to be sent/received in downward ($w = 0$) or upward ($w = 1$) communication.

Therefore, they are defined as follows, where $\{e_f, e_t, s\}(b_d, \beta) =$ `BoundaryCommTypes`$[c][b_d][w][0{:}2]$ and $b_d \in \{b_d^l, b_d^u\} =$ `Boundaries`$[m][d][0{:}1]$.

$$\begin{aligned}
\texttt{exti}[d][0] &= e_f(b_d^l, 0) & \texttt{exti}[d][1] &= e_f(b_d^u, 1) + s(b_d^u, 1) \\
\texttt{exto}[d][0] &= e_t(b_d^l, 1) & \texttt{exto}[d][1] &= e_t(b_d^u, 0) + s(b_d^u, 0) \\
\texttt{soff}[d][w] &= e_f(b_d^w, w) & \texttt{roff}[d][w] &= e_t(b_d^{1-w}, w) \\
\texttt{ssize}[d][w] &= s(b_d^w, w) & \texttt{rsize}[d][w] &= s(b_d^{1-w}, w)
\end{aligned}$$

Note that a downward ($w = 0$) or upward ($w = 1$) communication is a sending one through lower ($\beta = 0$) or upper ($\beta = 1$) boundary plane respectively (i.e., $\beta = w$), while it is a receiving one through upper ($\beta = 1$) or ($\beta = 0$) lower boundary plane respectively (i.e., $\beta = 1 - w$).

```
for (d=0; d<OH_DIMENSION; d++) {
  int blo=bd[d][OH_LOWER], bup=bd[d][OH_UPPER];
  exti[d][OH_LOWER] = bt[blo][OH_LOWER][OH_CTYPE_FROM];
  exti[d][OH_UPPER] =
    bt[bup][OH_UPPER][OH_CTYPE_FROM] + bt[bup][OH_UPPER][OH_CTYPE_SIZE];
  exto[d][OH_LOWER] = bt[blo][OH_UPPER][OH_CTYPE_TO];
  exto[d][OH_UPPER] =
    bt[bup][OH_LOWER][OH_CTYPE_TO] + bt[bup][OH_LOWER][OH_CTYPE_SIZE];
  for (lu=OH_LOWER; lu<=OH_UPPER; lu++) {
    int sb=bd[d][lu], rb=bd[d][1-lu];
    soff[d][lu] = bt[sb][lu][OH_CTYPE_FROM];
    roff[d][lu] = bt[rb][lu][OH_CTYPE_TO];
    ssize[d][lu] = bt[sb][lu][OH_CTYPE_SIZE];
    rsize[d][lu] = bt[rb][lu][OH_CTYPE_SIZE];
  }
}
```

Then, we call `set_border_comm()` four times for all combination of its argument `lu` $= \{0, 1\}$ for downward/upward communication and `sr` $= \{0, 1\}$ for sending/receiving, with the following other arguments.

- `esize` $=$ `FieldDesc`$[f]$`.esize` $= \varepsilon(f)$ and `f` $= f$.
- `xyz`$[0{:}D{-}1] =$ {`SubDomains`$[m][d][1] -$ `SubDomains`$[m][d][0]$} $= \{\delta_d^u(m) - \delta_d^l(m)\}$ followed by 0's.
- `wdh`$[D] =$ `FieldDesc`$[f]$`.size`$[D] = \{\Phi_d(f)\}$
- `exti`$[D][2]$ and `exto`$[D][2]$.
- `off`$[D][2] \in \{$`soff`$[D][2],$ `roff`$[D][2]\}$ and `size`$[D][2] \in \{$`ssize`$[D][2],$ `rsize`$[D][2]\}$.
- `type` is the argument of this function itself and is `MPI_DOUBLE` usually but can be `MPI_LONG_LONG_INT`.

- bx$[D][2]$ = BorderExc$[c][p][D][2]$.

```
  for (lu=OH_LOWER; lu<=OH_UPPER; lu++) {
    set_border_comm(esize, f, xyz, wdh, exti, exto, soff, ssize, lu, 0, type,
                    bx);
    set_border_comm(esize, f, xyz, wdh, exti, exto, roff, rsize, lu, 1, type,
                    bx);
  }
}
```

4.7.12 set_border_comm()

set_border_comm()

The function set_border_comm(), called solely from set_border_exchange(), fills elements buf, count, deriv and type of BorderExc$[c][p][d][w]\{$send, recv$\}$ for the downward ($w = 0$) or upward ($w = 1$) boundary communication of type c through d-dimensional boundary plane of a field-array f of primary ($p = 0$) or secondary ($p = 1$) subdomain m of the local node for all $d \in [0, D)$, where BorderExc$[c][p]$ is given through its argument bx, w is through the argument lu, and the argument sr determines send (sr $= 0$) or recv (sr $= 1$). The other arguments, esize $= \varepsilon(f)$, f $= f$, xyz$[D] = \{\delta^u_d(m) - \delta^l_d(m)\}$, wdh$[D] = \Phi_{[0,D)}(f)$, exti$[D][2] = e^{\{l,u\}}_i([0, D)) = e^{\{l,u\}}_i(\{x, y, z\})$, exto$[D][2] = e^{\{l,u\}}_o([0, D)) = e^{\{l,u\}}_o(\{x, y, z\})$, off$[D][2]$, size$[D][2] = \sigma^{\{0,1\}}_{[0,D)}$ and basetype $\in \{$MPI_DOUBLE, MPI_LONG_LONG_INT$\}$ have been discussed in §4.7.11.

Since;

- the boundary communications of the field-array f take place through d-th dimensional boundary plane in the ascending order of d;
- the MPI data-type for each communication depends on D; and
- we optimize the MPI data-type if the boundaries of sending/receiving planes are also those of field-array;

the code has many if-then-else's.

```
static void
set_border_comm(int esize, int f, int *xyz, int *wdh,
                int exti[OH_DIMENSION][2], int exto[OH_DIMENSION][2],
                int off[OH_DIMENSION][2], int size[OH_DIMENSION][2],
                int lu, int sr, MPI_Datatype basetype,
                struct S_borderexc bx[OH_DIMENSION][2]) {
  int bl[2]={1,1};
  MPI_Datatype tmptype[2]={MPI_DATATYPE_NULL,MPI_UB};
  int w=wdh[OH_DIM_X], wd=w*esize;
  int dp=OH_DIMENSION==1 ? 1 : wdh[OH_DIM_Y];
  MPI_Aint dispz[2]={0, wd*dp*sizeof(double)};
  struct S_bcomm *bcx, *bcy, *bcz;
  int xexto, yexti, yexto, zexti;
  int s;
  int lower = sr ? lu==OH_UPPER : lu==OH_LOWER;
```

First, we set `BorderExc`$[c][p][d][w]$`.`t`.deriv` $= 0$, where t is `send` (`sr` $= 0$) or `recv` (`sr` $= 1$), as their default values indicating `type` has a basic MPI data-type, for all $d \in [0, D)$. Then we calculate following *inner* and *outer extents* of the field-array f as;

$$\chi_d^i = (\delta_d^u + e_i^u(d)) - (\delta_d^l + e_i^l(d))$$
$$\chi_d^o = (\delta_d^u + e_o^u(d)) - (\delta_d^l + e_o^u(d))$$

to have $\chi_0^o \times \cdots \chi_{d-1}^o \times \chi_{d+1}^i \times \cdots \chi_{D-1}^i$ as the shape of a d-th dimensional sending/receiving plane[53]. By this shape definition and the ascending order of communications through each of d-th dimensional boundary planes from $d = 0$ to $d = D-1$, the d-th dimensional sending plane should have a part of $(d-1)$-th and lower dimensional receiving planes to *relay* boundary data of a subdomain to its neighbor contacted with a edge (if $D = 3$) or vertex (if $D \geq 2$).

The base (lowest) local coordinate of the d-th dimensional sending/receiving plane $(\lambda_0, \ldots, \lambda_{D-1})$ is defined as follows.

$$b_d = \begin{cases} \texttt{off}[d][w] & t = w \\ (\delta_d^u(m) - \delta_d^l(m)) + \texttt{off}[d][w] & t \neq w \end{cases}$$
$$\lambda_k = \begin{cases} e_o^l(k) & k < d \\ b_d & k = d \\ e_i^l(k) & k > d \end{cases}$$

Note that $t = w$ above means the downward sending or upward receiving planes and thus lower planes, while $t \neq w$ means the upward sending or downward receiving ones being upper planes.

```
bcx = (sr==0) ? &bx[OH_DIM_X][lu].send : &bx[OH_DIM_X][lu].recv;
bcx->deriv = 0;
xexto = xyz[OH_DIM_X] + exto[OH_DIM_X][OH_UPPER] - exto[OH_DIM_X][OH_LOWER];
if (OH_DIMENSION>OH_DIM_Y) {
  bcy = (sr==0) ? &bx[OH_DIM_Y][lu].send : &bx[OH_DIM_Y][lu].recv;
  bcy->deriv = 0;
  yexti = xyz[OH_DIM_Y] +
          exti[OH_DIM_Y][OH_UPPER] - exti[OH_DIM_Y][OH_LOWER];
  yexto = xyz[OH_DIM_Y] +
          exto[OH_DIM_Y][OH_UPPER] - exto[OH_DIM_Y][OH_LOWER];
}
if (OH_DIMENSION>OH_DIM_Z) {
  bcz = (sr==0) ? &bx[OH_DIM_Z][lu].send : &bx[OH_DIM_Z][lu].recv;
  bcz->deriv = 0;
  zexti = xyz[OH_DIM_Z] +
          exti[OH_DIM_Z][OH_UPPER] - exti[OH_DIM_Z][OH_LOWER];
}
```

Next, we fill `BorderExc`$[c][p][d][w]$`.`t`.{buf, count, deriv, type}` according to D and the shape of sending/receiving planes. In general, we let `buf` $=$ `count` $= 0$ and `type` $=$ `MPI_DATATYPE_NULL` and keep `deriv` $= 0$ for d if $\sigma_d^w = 0$ to mean no downward ($w = 0$) or upward ($w = 1$) communications take place through d-th dimensional boundary plane.

[53] Since $1 \leq D \leq 3$, at most we have two $\chi_d^{\{i,o\}}$ in the shape definition but it is conceptually as shown.

If $D = 1$ and $\sigma_x^w > 0$, we simply fill the elements only for $d = 0$ to transfer contiguous `size` $= \sigma_x^w \cdot \varepsilon(f)$ elements of `basetype` from `buf` $= fdisp(f, b_x, 0, 0)$ given by `Field_Disp()` for the element $[b_x][0]$ of the field-array f.

```
if (OH_DIMENSION==1) {
  if ((s=size[OH_DIM_X][lu])==0) {
    bcx->buf = bcx->count = 0;  bcx->type = MPI_DATATYPE_NULL;
  } else {
    bcx->type = basetype;
    bcx->count = s * esize;
    bcx->buf =
      Field_Disp(f,
                 lower ? off[OH_DIM_X][lu] : xyz[OH_DIM_X]+off[OH_DIM_X][lu],
                 0, 0);
  }
```

If $D = 2$, the elements for $d = 0$ are to transfer a set of σ_x^w line segments of χ_y^i grid points perpendicular to x-axis unless $\sigma_x^w = 0$. Therefore, we create a MPI data-type by `MPI_Type_vector()` with `count` $= \chi_y^i$ being the inner extent along y-axis, `blocklength` $= \sigma_x^w \cdot \varepsilon(f)$ and `stride` $= \Phi_x(f) \cdot \varepsilon(f)$ to store it in `type` and then commit it by `MPI_Type_commit()`. The element `count` $= 1$ because we transfer this single stride-vector and `deriv` $= 1$ because the data-type is derivative. The base of the vector `buf` is the one-dimensional index $fdisp(f, b_x, e_i^l(y), 0)$ of the element $[e_i^l(y)][b_x][0]$ of the field-array f, which is given by `Field_Disp()`.

The elements for $d = 1$ are to transfer a set of σ_y^w line segments of χ_x^o grid points perpendicular to y-axis unless $\sigma_y^w = 0$. If χ_x^o being the outer extent along x-axis is equal to $\Phi_x(f)$ being the x-dimensional size of the field-array f, transferred data set has contiguous `count` $= \sigma_y^w \cdot \chi_x^o \cdot \varepsilon(f)$ elements of the data type given by `basetype` argument. Otherwise, we have to create a MPI data-type by `MPI_Type_vector()` with `count` $= \sigma_y^w$, `blocklength` $= \chi_x^o \cdot \varepsilon(f)$ and `stride` $= \Phi_x(f) \cdot \varepsilon(f)$ to store it in `type` and then commit it by `MPI_Type_commit()`. In this case, the element `count` $= 1$ because we transfer this stride-vector and `deriv` $= 1$ because the data-type is derivative. In both cases, the base `buf` is the one-dimensional index $fdisp(f, e_o^l(x), b_y, 0)$ of the element $[b_y][e_o^l(x)][0]$ of the field-array f, which is given by `Field_Disp()`.

```
} else if (OH_DIMENSION==2) {
  if ((s=size[OH_DIM_X][lu])==0) {
    bcx->buf = bcx->count = 0;  bcx->type = MPI_DATATYPE_NULL;
  } else {
    MPI_Type_vector(yexti, s*esize, wd, basetype, &(bcx->type));
    MPI_Type_commit(&(bcx->type));  bcx->deriv = 1;
    bcx->count = 1;
    bcx->buf =
      Field_Disp(f,
                 lower ? off[OH_DIM_X][lu] : xyz[OH_DIM_X]+off[OH_DIM_X][lu],
                 exti[OH_DIM_Y][OH_LOWER], 0);
  }
  if ((s=size[OH_DIM_Y][lu])==0) {
    bcy->buf = bcy->count = 0;  bcy->type = MPI_DATATYPE_NULL;
  } else {
    if (xexto==w) {
      bcy->type = basetype;
      bcy->count = s * wd;
```

```
    } else {
      MPI_Type_vector(s, xexto*esize, wd, basetype, &(bcy->type));
      MPI_Type_commit(&(bcy->type));  bcy->deriv = 1;
      bcy->count = 1;
    }
    bcy->buf =
      Field_Disp(f, exto[OH_DIM_X][OH_LOWER],
                 lower ? off[OH_DIM_Y][lu] : xyz[OH_DIM_Y]+off[OH_DIM_Y][lu],
                 0);
  }
```

Finally if $D = 3$, the elements for $d = 0$ are to transfer a set of σ_x^w yz-subplanes of $\chi_z^i \times \chi_y^i$ grid points unless $\sigma_x^w = 0$. Therefore, we at first create a stride-vector for $\chi_y^i \times \sigma_x^w$ strip by `MPI_Type_vector()` as done for $D = 2$ and $d = 0$, then create a structured type by `MPI_Type_struct()` to stack the strips so that first elements of adjacent vectors are $s_{xy} = \Phi_y(f) \cdot \Phi_x(f) \cdot \varepsilon(f) \cdot$ `sizeof(double)` bytes[54]apart from each other, and finally commit it by `MPI_Type_commit()`. The element `count` $= \chi_z^i$ being the z-dimensional inner extent and `deriv` $= 1$ because the data-type is derivative. The base of the planes `buf` is the one-dimensional index $fdisp(f, b_x, e_i^l(y), e_i^l(z))$ of the element $[e_i^l(z)][e_i^l(y)][b_x][0]$ of the field-array f, which is given by `Field_Disp()`.

The elements for $d = 1$ are to transfer a set of σ_y^w xz-subplanes of $\chi_z^i \times \chi_x^o$ grid points unless $\sigma_y^w = 0$. If $\chi_x^o = \Phi_x(f)$, the set can be represented by a set of χ_z^i xy-strips of $\sigma_y^w \cdot \chi_x^o \cdot \varepsilon(f)$ contiguous elements of `basetype` stacked along z-axis with a stride of $\Phi_y(f) \cdot \Phi_x(f) \cdot \varepsilon(f)$, and thus by one stride-vector created and set into `type` element by `MPI_Type_vector()`. Otherwise, the strip is a stride-vector created by `MPI_Type_vector()` with σ_y^w line segments of $\chi_x^o \cdot \varepsilon(f)$ elements with a stride of $\Phi_x(f) \cdot \varepsilon(f)$. Then χ_z^i strips are stacked so that first elements of adjacent strips are s_{xy}-byte apart from each other by `MPI_Type_struct()` to set the strip type into `type` element and by setting `count` element to χ_z^i. In both cases, the MPI data-type in `type` element is commited by `MPI_Type_commit()`, `deriv` is set to 1 because of derivative, and the base of the planes `buf` is the one-dimensional index $fdisp(f, e_o^l(x), b_y, e_i^l(z))$ of the element $[e_i^l(z)][b_y][e_o^l(x)][0]$ of the field-array f, which is given by `Field_Disp()`.

The elements for $d = 2$ are to transfer a set of σ_z^w xy-subplanes of $\chi_y^o \times \chi_x^o$ grid points unless $\sigma_z^w = 0$. The `type` and `count` elements are dependent on if $\chi_x^o = \Phi_x(f)$ and/or $\chi_y^o = \Phi_y(f)$ and thus we have the following four cases.

- If $\chi_x^o = \Phi_x(f)$ and $\chi_y^o = \Phi_y(f)$, the set has contiguous $\sigma_z^w \cdot \chi_y^o \cdot \chi_x^o \cdot \varepsilon(f)$ elements of `basetype`
- If $\chi_x^o = \Phi_x(f)$ but $\chi_y^o \neq \Phi_y(f)$, a xy-subplane has contiguous $\chi_y^o \cdot \chi_x^o \cdot \varepsilon(f)$ elements of `basetype` and then σ_z^w subplanes are stacked with a stride of $\Phi_y(f) \cdot \Phi_x(f) \cdot \varepsilon(f)$. Therefore, the `type` element is created by `MPI_Type_vector()` and `count` $= 1$ to transfer one single stride-vector.
- If $\chi_x^o \neq \Phi_x(f)$ but $\chi_y^o = \Phi_y(f)$, the set of xy-subplanes are considered as the set of $\sigma_z^w \cdot \chi_y^o$ line segments of $\chi_x^o \cdot \varepsilon(f)$ `basetype` elements with a stride of $\Phi_x(f) \cdot \varepsilon(f)$. Therefore, the `type` element is created by `MPI_Type_vector()` and `count` $= 1$ to transfer one single stride-vector.

[54]If `basetype` is `MPI_LONG_LONG_INT`, the gap should be calculated with `sizeof(dint)` but using `sizeof(double)` is safe because they are equivalent.

- If $\chi^o_x \neq \Phi_x(f)$ and $\chi^o_y \neq \Phi_y(f)$, a xy-subplane is the set of χ^o_y line segments of $\chi^o_x \cdot \varepsilon(f)$ `basetype` elements with a stride of $\Phi_x(f) \cdot \varepsilon(f)$, which is created by `MPI_Type_vector()`. Then σ^w_z subplanes are stacked so that first elements of adjacent subplanes are s_{xy}-byte apart from each other by `MPI_Type_struct()` to set the strip type into `type` element and by setting `count` element to σ^w_z.

For the last three cases, we commit the MPI data-type in `type` element by `MPI_Type_commit()` and set `deriv` = 1. Then for all of four cases, the base of xy-subplanes `buf` is the one-dimensional index $fdisp(f, e^l_o(x), e^l_o(z), b_z)$ of the element $[b_z][e^l_o(y)][e^l_o(x)][0]$ of the field-array f, which is given by `Field_Disp()`.

```
} else {
  if ((s=size[OH_DIM_X][lu])==0) {
    bcx->buf = bcx->count = 0;  bcx->type = MPI_DATATYPE_NULL;
  } else {
    MPI_Type_vector(yexti, s*esize, wd, basetype, tmptype);
    MPI_Type_struct(2, bl, dispz, tmptype, &(bcx->type));
    MPI_Type_commit(&(bcx->type));  bcx->deriv = 1;
    bcx->count = zexti;
    bcx->buf =
      Field_Disp(f,
                 lower ? off[OH_DIM_X][lu] : xyz[OH_DIM_X]+off[OH_DIM_X][lu],
                 exti[OH_DIM_Y][OH_LOWER], exti[OH_DIM_Z][OH_LOWER]);
  }
  if ((s=size[OH_DIM_Y][lu])==0) {
    bcy->buf = bcy->count = 0;  bcy->type = MPI_DATATYPE_NULL;
  } else {
    if (xexto==w) {
      MPI_Type_vector(zexti, s*wd, wd*dp, basetype, &(bcy->type));
      bcy->count = 1;
    } else {
      MPI_Type_vector(s, xexto*esize, wd, basetype, tmptype);
      MPI_Type_struct(2, bl, dispz, tmptype, &(bcy->type));
      bcy->count = zexti;
    }
    MPI_Type_commit(&(bcy->type));  bcy->deriv = 1;
    bcy->buf =
      Field_Disp(f, exto[OH_DIM_X][OH_LOWER],
                 lower ? off[OH_DIM_Y][lu] : xyz[OH_DIM_Y]+off[OH_DIM_Y][lu],
                 exti[OH_DIM_Z][OH_LOWER]);
  }
  if ((s=size[OH_DIM_Z][lu])==0) {
    bcz->buf = bcz->count = 0;  bcz->type = MPI_DATATYPE_NULL;
  } else {
    if (xexto==w && yexto==dp) {
      bcz->type = basetype;
      bcz->count = s * wd * dp;
    } else {
      if (xexto==w) {
        MPI_Type_vector(s, wd*yexto, wd*dp, basetype, &(bcz->type));
        bcz->count = 1;
      } else if (yexto==dp) {
        MPI_Type_vector(s*yexto, xexto*esize, wd, basetype, &(bcz->type));
        bcz->count = 1;
```

```
        } else {
          MPI_Type_vector(yexto, xexto*esize, wd, basetype, tmptype);
          MPI_Type_struct(2, bl, dispz, tmptype, &(bcz->type));
          bcz->count = s;
        }
        MPI_Type_commit(&(bcz->type));  bcz->deriv = 1;
      }
      bcz->buf =
        Field_Disp(f, exto[OH_DIM_X][OH_LOWER], exto[OH_DIM_Y][OH_LOWER],
                   lower ? off[OH_DIM_Z][lu] :
                           xyz[OH_DIM_Z]+off[OH_DIM_Z][lu]);
    }
  }
}
```

Note that the data types for each D can be created by `MPI_Type_create_subarray()` giving it the following arguments, for example, if $D = 3$ and $d = 2$.

$$\begin{aligned}
\texttt{ndims} &= 4 \\
\texttt{array_of_sizes} &= \{\Phi_z(f), \Phi_y(f), \Phi_x(f), \varepsilon(f)\} \\
\texttt{array_of_subsizes} &= \{\sigma_z^w, \chi_y^o, \chi_x^o, \varepsilon(f)\} \\
\texttt{array_of_starts} &= \{b_z, e_o^l(y), e_o^l(x), 0\} \\
\texttt{order} &= \texttt{MPI_ORDER_C} \\
\texttt{oldtype} &= \texttt{basetype}
\end{aligned}$$

However, it is not sure that any MPI implementations take care of special cases such as $\chi_x^o = \Phi_x(f)$ and $\chi_y^o = \Phi_y(f)$ to create $\sigma_z \cdot \chi_y^o \cdot \chi_x^o \cdot \varepsilon(f)$ contiguos `basetype` elements as `MPI_Type_contiguous()` does. Therefore, we check such condisions and use `MPI_Type_vector()` and `MPI_Type_struct()` only if necessary.

4.7.13 `clear_border_exchange()`

`clear_border_exchange()` The function `clear_border_exchange()`, called from `init_fields()` and `transbound3()`, reinitalizes elements of `BorderExc`$[c][1][d][w]$`.{send,recv}` for the boundary communication of the secondary subdomain of the local node for all $c \in [0, C)$, $d \in [0, D)$ and $w \in \{0, 1\}$. The essential part of the reinitialization is to free derivative data types in `type` elements by `MPI_Type_free()` if `deriv` $= 1$. The other essentially required operation is to let `count` element be -1 to indicate that the communication parameters for the secondary subdomain are not set and thus we need to set them by `set_border_exchange()` when the result of type c communication is to broadcasted by `oh3_exchange_borders()`. The other operations to let `buf` and `deriv` be 0 and to let `type` be `MPI_DATATYPE_NULL` are reasonable but not necessary.

```
void
clear_border_exchange() {
  int ne=nOfExc, e, d, lu;
  struct S_borderexc (*bx)[2][OH_DIMENSION][2] = BorderExc;

  for (e=0; e<ne; e++) {
    for (d=0; d<OH_DIMENSION; d++) {
      for (lu=OH_LOWER; lu<=OH_UPPER; lu++) {
```

```
        if (bx[e][1][d][lu].send.deriv)
          MPI_Type_free(&bx[e][1][d][lu].send.type);
        if (bx[e][1][d][lu].recv.deriv)
          MPI_Type_free(&bx[e][1][d][lu].recv.type);
        bx[e][1][d][lu].send.buf =   bx[e][1][d][lu].recv.buf = 0;
        bx[e][1][d][lu].send.count = bx[e][1][d][lu].recv.count = -1;
        bx[e][1][d][lu].send.deriv = bx[e][1][d][lu].recv.deriv = 0;
        bx[e][1][d][lu].send.type =  bx[e][1][d][lu].recv.type =
          MPI_DATATYPE_NULL;
      }
    }
  }
}
```

4.7.14 oh3_grid_size()

oh3_grid_size_()
oh3_grid_size()

The API function `oh3_grid_size_()` for Fortran and `oh3_grid_size()` for C provide a simulator body calling them with the means to specify the grid size of each dimension if the real coordinate for particle is different from integer coordinate for subdomains. Its `size`$[D]$ argument array has the scale factor γ_d in its element $[d]$.

For each $d \in [0, D)$, the function lets `Grid`$[d]$`.gsize` $= \gamma_d$ and `Grid`$[d]$`.rgsize` $= 1/\gamma_d$, and updates `SubDomainsFloat`$[m][d][\beta] = \delta_d^\beta(m)$ for all $m \in [0, N)$ and `Grid`$[d]$'s elements, namely `fcoord`$[\beta] = \Delta_d^\beta$, `fsize` $= \delta_d^{\max}$ and `light.fthresh` $= \Delta_d^-$, by multiplying them by γ_d. It also updates and `Grid`$[d]$'s elements `light.rfsize` $= 1/\delta_d^{\min}$ and `light.rfsizeplus` $= 1/(\delta_d^{\min}+1)$ by dividing them by γ_d. If irregular process coordinate, in addition, `SubDomainDesc`$[m]$`.coord`$[d]$`.fc`$[\beta] = \delta_d^\beta(m)$ for all $m \in [0, N)$ are also updated by multiplying them by γ_d.

```
void
oh3_grid_size_(double size[OH_DIMENSION]) {
  oh3_grid_size(size);
}
void
oh3_grid_size(double size[OH_DIMENSION]) {
  int d, n, nn=nOfNodes;
  for (d=0; d<OH_DIMENSION; d++) {
    double s = (Grid[d].gsize = size[d]);
    Grid[d].rgsize = 1 / s;
    for (n=0; n<nn; n++) {
      SubDomainsFloat[n][d][OH_LOWER] *= s;
      SubDomainsFloat[n][d][OH_UPPER] *= s;
    }
    Grid[d].fcoord[OH_LOWER] *= s;
    Grid[d].fcoord[OH_UPPER] *= s;
    Grid[d].fsize *= s;
    Grid[d].light.rfsize /= s;
    Grid[d].light.rfsizeplus /= s;
    Grid[d].light.fthresh *= s;
    if (SubDomainDesc) {
      for (n=0; n<nn; n++) {
        SubDomainDesc[n].coord[d].fc[OH_LOWER] *= s;
        SubDomainDesc[n].coord[d].fc[OH_UPPER] *= s;
```

```
      }
    }
  }
}
```

4.7.15 `oh3_transbound()` and `transbound3()`

`oh3_transbound_()` `oh3_transbound()`

The API function `oh3_transbound_()` for Fortran and `oh3_transbound()` for C provide a simulator body calling them with the load-balanced particle transfer mechanism of level-2 or level-1 library together with subdomain-related functions of level-3's own. The meanings of their two arguments, `currmode` and `stats`, and return value in $\{-1, 0, 1\}$ are perfectly equivalent to those of the level-1 and level-2 counterparts `oh1_transbound[_]()` and `oh2_transbound[_]()`. Also similarly to the counterparts, their bodies only have a simple call of `transbound3()` but the third argument `level` is 3 to indicate the function is called from level-3 API functions.

```
int
oh3_transbound_(int *currmode, int *stats) {
  return(transbound3(*currmode, *stats, 3));
}
int
oh3_transbound(int currmode, int stats) {
  return(transbound3(currmode, stats, 3));
}
```

`transbound3()`

The function `transbound3()`, called from `oh3_transbound_()` or `oh3_transbound()`, at first calls its level-2 counterpart `transbound2()` usually, i.e., `excludeLevel2` is false, or the level-1 counterpart `transbound1()` if the level-3 library was initialized by `oh13_init()` and thus `excludeLevel2` is true.

Then if the `transbound1()` or `transbound2()` assigned a parent to the local node different from before the call, i.e., `RegionId`[1] before and after the call are different, we call `clear_border_exchange()` to reinitialize `BorderExc`$[c][1][d][w]$ for all $c \in [0, C)$, $d \in [0, D)$ and $w \in \{0, 1\}$ for the old secondary subdomain, and then call `set_field_descriptors()` to set `FieldDesc`$[f]$`.{bc,red}.size`[1] for all $f \in [0, F)$ for the new secondary subdomain giving `FieldTypes`$[F][7]$ and `SubDomains`$[m][D]][2]$ to the function where m is the new secondary subdomain identifier. Note that these function are not called if old/new value of `RegionId`[1] are negative meaning that we were or will be in primary mode or the local node was or will be the root of the family tree.

Finally the function returns the value from `transbound1()` or `transbound2()` to the caller.

```
static int
transbound3(int currmode, int stats, int level) {
  int oldp=RegionId[1], newp;

  currmode = excludeLevel2 ? transbound1(currmode, stats, 1) :
                             transbound2(currmode, stats, level);
  newp = RegionId[1];
  if (oldp!=newp) {
    if (oldp>=0)  clear_border_exchange();
```

```
    if (newp>=0)  set_field_descriptors(FieldTypes, SubDomains[newp], 1);
  }
  return(currmode);
}
```

4.7.16 Macro `Map_Particle_To_Neighbor()`

`Map_Particle_To_Neighbor()` The macro `Map_Particle_To_Neighbor`$(\&x, m, d, k, 3^d)$, used in three versions of `oh3_map_particle_to_neighbor()` does;

$$k \leftarrow \begin{cases} k - 3^d & x < \delta_d^l(m) \cdot \gamma_d \\ k & \delta_d^l(m) \cdot \gamma_d \le x < \delta_d^u(m) \cdot \gamma_d \\ k + 3^d & \delta_d^u(m) \cdot \gamma_d < x \end{cases}$$

referring to `SubDomainsFloat`$[m][d][\beta] = \delta_d^\beta(m) \cdot \gamma_d$, so that k finally has the index of `Neighbors`$[p][3^D]$ for the subdomain which the particle whose d-th dimensional integer coordinate is x_I. Thus the subdomain is expected to be neighboring to the subdomain m being the primary subdomain n ($p = 0$) or the secondary subdomain $parent(n)$ ($p = 1$) of the local node n.

It also checks if $x < \Delta_d^l \cdot \gamma_d =$ `Grid`$[d]$`.fcoord`$[0]$ or $x \ge \Delta_d^u \cdot \gamma_d =$ `Grid`$[d]$`.fcoord`$[1]$. If so and the lower/upper boundary condition of the system domain is not periodic, i.e., `Boundaries`$[m][d][\{0,1\}] \neq 0$, the macro makes its user function return to its caller with -1 to indicate particle is out-of-bound. If periodic, on the other hand, the floating point coordinate x of the particle is incremented/decremented by $(\Delta_d^u - \Delta_d^l) \cdot \gamma_d$ so that the particle moves to the opposite end along the d-th dimensional axis of the system domain.

```
#define Map_Particle_To_Neighbor(XYZ,RID,DIM,N,INC) {\
  double xyz=*XYZ;\
  if (xyz<SubDomainsFloat[RID][DIM][OH_LOWER]) {\
    N -= INC;\
    if (xyz<Grid[DIM].fcoord[OH_LOWER]) {\
      if (Boundaries[RID][DIM][OH_LOWER])  return(-1);\
      *XYZ += Grid[DIM].fcoord[OH_UPPER] - Grid[DIM].fcoord[OH_LOWER];\
    }\
  } else if (xyz>=SubDomainsFloat[RID][DIM][OH_UPPER]) {\
    N += INC;\
    if (xyz>=Grid[DIM].fcoord[OH_UPPER]) {\
      if (Boundaries[RID][DIM][OH_UPPER])  return(-1);\
      *XYZ -= Grid[DIM].fcoord[OH_UPPER] - Grid[DIM].fcoord[OH_LOWER];\
    }\
  }\
}
```

4.7.17 Macro `Neighbor_Id()`

`Neighbor_Id()` The macro `Neighbor_Id()`, used in three versions of `oh3_map_particle_to_neighbor()`, translates its argument m' in `Neighbors`$[][]$ into $m = m'$ if $m' \ge 0$, $m = -(m' + 1)$ if $-N \le m' < 0$, or $m = -1$ if $m' < -N$, to have the real neighboring subdomain identifier m, or -1 to indicate out-of-bounds.

```
#define Neighbor_Id(N) ((n=(N))<0 ? ((n=-n-1)<nOfNodes ? n : -1) : n)
```

4.7.18 oh3_map_particle_to_neighbor()

oh3_map_particle_to_neighbor_()
oh3_map_particle_to_neighbor()

The API functions oh3_map_particle_to_neighbor_()[55] for Fortran and oh3_map_particle_to_neighbor() for C find the subdomain m, which is neighboring to the primary ($\texttt{ps} = p = 0$) or secondary ($\texttt{ps} = p = 1$) subdomain, should accommodate the particle whose coordinate values are pointed by the arguments x, y (if $D \geq 2$) and z (if $D = 3$), and is returned to the caller. If such a subdomain is not found due to that the particle is moving out-of-bounds, the function returns -1 instead of the subdomain identifier. The function also modifies the coordinate pointed by the arguments if the particle has moved crossing periodic boundary planes of the system domain, so that it *jumps* to the coordinate point corresponding to the opposite boundary planes.

Since the function takes D arguments for the particle coordinate, we have three versions of each function. In all versions, the Fortran API oh3_map_particle_to_neighbor_()[56] simply calls C counterpart oh3_map_particle_to_neighbor() and returns what the C counterpart returns. Also in all versions, oh3_map_particle_to_neighbor() for the local node n invokes the macro Map_Particle_To_Neighbor() D times giving arguments $\&x = \{\texttt{x}, \texttt{y}, \texttt{z}\}[d]$, $m = \texttt{RegionId}[p] \in \{n, parent(n)\}$, d, k and 3^d to its $(d{+}1)$-th invocation where $k = (3^D - 1)/2 = \sum_{d=0}^{D-1} 3^d$ at initial. Since the macro modifies k as;

$$k \leftarrow \begin{cases} k - 3^d & x < \delta_d^l(m) \cdot \gamma_d \\ k & \delta_d^l(m) \cdot \gamma_d \leq x < \delta_d^u(m) \cdot \gamma_d \\ k + 3^d & \delta_d^u(m) \cdot \gamma_d < x \end{cases}$$

we have $k = \sum_{d=0}^{D-1} \nu_d 3^d$ ($\nu_d \in [0, 2]$) corresponding to the process coordinate ($\pi_0{+}\nu_0{-}1$, $\ldots, \pi_{D-1} + \nu_{D-1} - 1$) where $(\pi_0, \ldots, \pi_{D-1})$ is for that of the subdomain m. Therefore, the target subdomain l or the out-of-bound indicator -1 is obtained from $l' = \texttt{Neighbors}[p][k]$ by the following calculation with macro Neighbor_Id(k).

$$l = \begin{cases} -1 & l' < -N \\ -(l'+1) & -N \leq l' < 0 \\ l' & 0 \leq l' \end{cases}$$

```
#if OH_DIMENSION==1
int
oh3_map_region_to_adjacent_node_(double *x, int *ps) {
  return(oh3_map_particle_to_neighbor(x, *ps));
}
int
oh3_map_particle_to_neighbor_(double *x, int *ps) {
  return(oh3_map_particle_to_neighbor(x, *ps));
}
int
oh3_map_particle_to_neighbor(double *x, int ps) {
  int rid=RegionId[ps], n=OH_NEIGHBORS>>1;

  Map_Particle_To_Neighbor(x, rid, OH_DIM_X, n, 1);
  return(Neighbor_Id(Neighbors[ps][n]));
}
#elif OH_DIMENSION==2
```

[55] And its aliase oh3_map_region_to_adjacent_node_() for backward compatiblity.
[56] And oh3_map_region_to_adjacent_node_() as well.

```
int
oh3_map_region_to_adjacent_node_(double *x, double *y, int *ps) {
  return(oh3_map_particle_to_neighbor(x, y, *ps));
}
int
oh3_map_particle_to_neighbor_(double *x, double *y, int *ps) {
  return(oh3_map_particle_to_neighbor(x, y, *ps));
}
int
oh3_map_particle_to_neighbor(double *x, double *y, int ps) {
  int rid=RegionId[ps], n=OH_NEIGHBORS>>1;

  Map_Particle_To_Neighbor(x, rid, OH_DIM_X, n, 1);
  Map_Particle_To_Neighbor(y, rid, OH_DIM_Y, n, 3);
  return(Neighbor_Id(Neighbors[ps][n]));
}
#else
int
oh3_map_region_to_adjacent_node_(double *x, double *y, double *z, int *ps) {
  return(oh3_map_particle_to_neighbor(x, y, z, *ps));
}
int
oh3_map_particle_to_neighbor_(double *x, double *y, double *z, int *ps) {
  return(oh3_map_particle_to_neighbor(x, y, z, *ps));
}
int
oh3_map_particle_to_neighbor(double *x, double *y, double *z, int ps) {
  int rid=RegionId[ps], n=OH_NEIGHBORS>>1;

  Map_Particle_To_Neighbor(x, rid, OH_DIM_X, n, 1);
  Map_Particle_To_Neighbor(y, rid, OH_DIM_Y, n, 3);
  Map_Particle_To_Neighbor(z, rid, OH_DIM_Z, n, 9);
  return(Neighbor_Id(Neighbors[ps][n]));
}
#endif
```

4.7.19 Macros `Map_Particle_To_Subdomain()` and `Adjust_Subdomain()`

`Map_Particle_To_Subdomain()` `Adjust_Subdomain()`

The macro `Map_Particle_To_Subdomain`(x, d, π_d), used in three versions of `oh3_map_particle_to_subdomain()`, does;

$$\pi_d \leftarrow \begin{cases} \lfloor (x - \Delta_d^l \cdot \gamma_d)/(\delta_d^{\min} \cdot \gamma_d) \rfloor & x < \Delta_d^- \cdot \gamma_d \\ \Pi_d^- + \lfloor (x - \Delta_d^- \cdot \gamma_d)/((\delta_d^{\min} + 1) \cdot \gamma_d) \rfloor & x \geq \Delta_d^- \cdot \gamma_d \end{cases}$$

to translate the given particle coordinate x into the d-th dimensional process coordinate π_d in the regular process coordinate system where the particle resides if $\Delta_d^l \cdot \gamma_d \leq x < \Delta_d^u \cdot \gamma_d$, referring to

$$\texttt{Grid}[d]\texttt{.fcoord}[\{0,1\}] = \{\Delta_d^l \cdot \gamma_d,\ \Delta_d^u \cdot \gamma_d\}$$
$$\texttt{Grid}[d]\texttt{.light.}\{\texttt{rfsize,rfsizeplus,fthresh,n}\} = \{1/(\delta_d^{\min} \cdot \gamma_d),\ 1/((\delta_d^{\min}+1) \cdot \gamma_d),\ \Delta_d^- \cdot \gamma_d,\ \Pi_d^-\}$$

Otherwise, the macro makes its user function return to its caller with -1 to indicate the particle is out-of-bounds.

The translation above, however, can be inaccurate because neither the division by $\delta_d^{\min} \cdot \gamma_d$ and $(\delta_d^{\min}+1)\cdot\gamma_d$ nor multiplication by their reciprocals done in the implementation give accurate result due to floating-point calculation error. Therefore `oh3_map_particle_to_subdomain()` invokes `Adjust_Subdomain`(x, d, m, Π') where $m = rank(\pi_0, \cdots, \pi_{D-1})$ and $\Pi' = \prod_{i=0}^{d-1} \Pi_i$ to correct the possible errors by the following.

$$m \leftarrow \begin{cases} m - \Pi' & x < \delta_d^l(m) \\ m + \Pi' & x \geq \delta_d^u(m) \\ m & \text{otherwise} \end{cases}$$

```
#define Map_Particle_To_Subdomain(XYZ,DIM,SDOM) {\
  double thresh = Grid[DIM].light.fthresh;\
  if (XYZ<Grid[DIM].fcoord[OH_LOWER] || XYZ>=Grid[DIM].fcoord[OH_UPPER])\
    return(-1);\
  if (XYZ<thresh)\
    SDOM = (XYZ - Grid[DIM].fcoord[OH_LOWER]) * Grid[DIM].light.rfsize;\
  else  SDOM = (int)((XYZ - thresh) * Grid[DIM].light.rfsizeplus) + \
               Grid[DIM].light.n;\
}
#define Adjust_Subdomain(XYZ,DIM,SDOM,INC) {\
  if (XYZ<SubDomainsFloat[SDOM][DIM][OH_LOWER])  SDOM-=INC;\
  else if (XYZ>=SubDomainsFloat[SDOM][DIM][OH_UPPER])  SDOM+=INC;\
}
```

4.7.20 `oh3_map_particle_to_subdomain()`

`oh3_map_particle_to_subdomain_()` `oh3_map_particle_to_subdomain()`

The API function `oh3_map_particle_to_subdomain_()`[57] for Fortran and `oh3_map_particle_to_subdomain()` for C find the subdomain m, which should accommodate the particle whose coordinate is given by the arguments `x`, `y` (if $D \geq 2$) and `z` (if $D = 3$), and is returned to the caller. If such a subdomain is not found due to that the particle is moving out-of-bounds, the function returns -1 instead of the subdomain identifier.

Since the function takes D arguments for the particle coordintate, we have three versions of each function. In all versions, the Fortran API `oh3_map_particle_to_subdomain_()`[58] simply calls C counterpart `oh3_map_particle_to_subdomain()` and returns what the C counterpart returns. Also in all versions, `oh3_map_particle_to_subdomain()` for the local node n calls `map_irregular_subdomain()` with `x`, `y` and `z` (or 0 if $D < 3$) to have m (or -1) and to return to the caller with it, if `SubDomainDesc` $\neq$ `NULL` meaning irregular process coordinate. Otherwise, i.e., `SubDomainDesc` = `NULL` meaning regular process coordinate, the function invokes the macro `Map_Particle_To_Subdomain()` D times giving arguments $x_d = \{\texttt{x}, \texttt{y}, \texttt{z}\}[d]$, d and π_d to have π_d for all $d \in [0, D)$ from which the return value is approximated by $m = rank(\pi_0, \ldots, \pi_{D-1})$, which is, for example, $\pi_0 + \Pi_0 \cdot (\pi_1 + \Pi_1 \cdot \pi_2)$ if $D = 3$. Then the macro `Adjust_Subdomain()` is invoked D times with arguments x_d, d, m and $\prod_{i=0}^{d-1} \Pi_d$ to let m be its neighbor including itself to correct the floating-point calculation error in `Map_Particle_To_Subdomain()`.

```
#if OH_DIMENSION==1
int
```

[57] And its aliase `oh3_map_region_to_node_()` for backward compatiblity.
[58] And `oh3_map_region_to_node_()` as well.

```
oh3_map_region_to_node_(double *x) {
  return(oh3_map_particle_to_subdomain(*x));
}
int
oh3_map_particle_to_subdomain_(double *x) {
  return(oh3_map_particle_to_subdomain(*x));
}
int
oh3_map_particle_to_subdomain(double x) {
  int sdx;

  if (SubDomainDesc)  return(map_irregular_subdomain(x, 0.0, 0.0));
  Map_Particle_To_Subdomain(x, OH_DIM_X, sdx);
  Adjust_Subdomain(x, OH_DIM_X, sdx, 1);
  return(sdx);
}
#elif OH_DIMENSION==2
int
oh3_map_region_to_node_(double *x, double *y) {
  return(oh3_map_particle_to_subdomain(*x, *y));
}
int
oh3_map_particle_to_subdomain_(double *x, double *y) {
  return(oh3_map_particle_to_subdomain(*x, *y));
}
int
oh3_map_particle_to_subdomain(double x, double y) {
  int sdx, sdy, sd, nx=Grid[OH_DIM_X].n;

  if (SubDomainDesc)  return(map_irregular_subdomain(x, y, 0.0));
  Map_Particle_To_Subdomain(x, OH_DIM_X, sdx);
  Map_Particle_To_Subdomain(y, OH_DIM_Y, sdy);
  sd = sdx + nx * sdy;
  Adjust_Subdomain(x, OH_DIM_X, sd, 1);
  Adjust_Subdomain(y, OH_DIM_Y, sd, nx);
  return(sd);
}
#else
int
oh3_map_region_to_node_(double *x, double *y, double *z) {
  return(oh3_map_particle_to_subdomain(*x, *y, *z));
}
int
oh3_map_particle_to_subdomain_(double *x, double *y, double *z) {
  return(oh3_map_particle_to_subdomain(*x, *y, *z));
}
int
oh3_map_particle_to_subdomain(double x, double y, double z) {
  int sdx, sdy, sdz, sd, nx=Grid[OH_DIM_X].n, nxy=nx*Grid[OH_DIM_Y].n;

  if (SubDomainDesc)  return(map_irregular_subdomain(x, y, z));
  Map_Particle_To_Subdomain(x, OH_DIM_X, sdx);
  Map_Particle_To_Subdomain(y, OH_DIM_Y, sdy);
  Map_Particle_To_Subdomain(z, OH_DIM_Z, sdz);
```

```
  sd = sdx + nx * sdy + nxy * sdz;
  Adjust_Subdomain(x, OH_DIM_X, sd, 1);
  Adjust_Subdomain(y, OH_DIM_Y, sd, nx);
  Adjust_Subdomain(z, OH_DIM_Z, sd, nxy);
  return(sd);
}
#endif
```

4.7.21 map_irregular_subdomain()

map_irregular_subdomain()

The function `map_irregular_subdomain()`, called from three versions of `oh3_map_particle_to_subdomain()`, simply calls `map_irregular()` passing its own arguments `x`, `y` and `z` for a particle position, together with `dim` $= 0$, `from` $= 0$ and `n` $= N$ to let it search the subdomain containing the particle from the whole members of `SubDomainDesc`$[N]$ starting from the dimension 0.

```
int
map_irregular_subdomain(double x, double y, double z) {
  return(map_irregular(x, y, z, OH_DIM_X, 0, nOfNodes));
}
```

4.7.22 map_irregular()

map_irregular()

The recursive function `map_irregular()`, called from `map_irregular_subdomain()` and `map_irregular()` itself, tries to find the subdomain containing the particles whose $(d+k)$-th coordinate is given by the argument `p`k, where $d =$ `dim` and $k \in [0, 2]$ (and `p`$k = 0$ for k s.t. $d + k \geq D$), from `SubDomainDesc`$[[i_0, i_n)]$ where $i_0 =$ `from` and $i_n = i_0 +$ `n`.

`SubDomainDesc`$[[i_0, i_n)]$ is ascendingly ordered by $\delta^l_d(i(m)) + \delta^u_d(i(m))$ or equivalently $(\delta^l_d(i(m)) + \delta^u_d(i(m))) \cdot \gamma_d$, where $\{i(m),\ \delta^l_d(i(m)) \cdot \gamma_d,\ \delta^u_d(i(m)) \cdot \gamma_d\} =$ `SubDomainDesc`$[m]$`.{id, coord`$[d]$`.fc`$[\{0,1\}]\}$, corresponging to the midpoint plane of the lower/upper boundary planes, and a subdomain m such that $\delta^l_d(i(m)) \cdot \gamma_d \leq x =$ `p0` $< \delta^u_d(i(m)) \cdot \gamma_d$ should satisfy the following because $(\delta^u_d(i(m)) - \delta^l_d(i(m))) \cdot \gamma_d \leq \delta^{\max}_d \cdot \gamma_d =$ `Grid`$[d]$`.fsize`.

$$\frac{\delta^l_d(i(m)) + \delta^u_d(i(m))}{2} - \frac{\delta^{\max}_d}{2} \leq x/\gamma_d < \frac{\delta^l_d(i(m)) + \delta^u_d(i(m))}{2} + \frac{\delta^{\max}_d}{2}$$
$$\iff 2x - \delta^{\max}_d \cdot \gamma_d < (\delta^l_d(i(m)) + \delta^u_d(i(m))) \cdot \gamma_d \leq 2x + \delta^{\max}_d \cdot \gamma_d$$

Therefore, $j_0 = \min\{m \,|\, (\delta^l_d(i(m)) + \delta^u_d(i(m))) \cdot \gamma_d > 2x - \delta^{\max}_d \cdot \gamma_d\}$ and $j_n = \min\{m \,|\, (\delta^l_d(i(m)) + \delta^u_d(i(m))) \cdot \gamma_d > 2x + \delta^{\max}_d \cdot \gamma_d\}$ can be found by a binary search in `SubDomainDesc`$[[i_0, i_n)]$ and then `SubDomainDesc`$[[j_0, i_n)]$ by calling `map_irregular_range()` twice giving it $2x - \delta^{\max}_d \cdot \gamma_d$ and $2x + \delta^{\max}_d \cdot \gamma_d$ to limit the targets to find $\{m \,|\, \delta^l_d(i(m)) \leq x/\gamma_d < \delta^u_d(i(m))\} \subseteq [j_0, j_n)$.

Then we traverse `SubDomainDesc`$[[j_0, j_n)]$ only for those `SubDomainDesc`$[j]$`.coord`$[d]$`.h` $= j$ skipping its successors such that $\delta^l_d(i(j')) = \delta^l_d(i(j))$ and $\delta^u_d(i(j')) = \delta^u_d(i(j))$ where $j' \in (j, j+l)$ and $l =$ `SubDomainDesc`$[j]$`.coord`$[d]$`.n`. Note that the minimality of j_0 assured that `SubDomainDesc`$[j_0]$`.coord`$[d]$`.h` $= j_0$, i.e., j_0 is the head subdomain in a wall or pillar.

Then for each j such that $\delta^l_d(i(j)) \leq x/\gamma_d < \delta^u_d(i(j))$ if $d < D - 1$, we recursively call `map_irregular()` itself giving it `p1`, `p2` and 0 for the particle position, $d + 1$, and j and l to specify the wall or pillar in which the search takes place. Then we simply return the

return value to the caller if it is non-negative and thus the target subdomain identifier, or we continue the traversal otherwise.

On the other hand, if $d = D-1$ and we find $\delta_d^l(i(j)) \le x/\gamma_d < \delta_d^u(i(j))$, $i(j)$ is returned to the caller as the target subdomain identifier.

Finally, if we could not find the target in the traversal (including the case of empty $[j_0, j_n)$), we return -1 to indicate the search failure.

```
static int
map_irregular(double p0, double p1, double p2, int dim, int from, int n) {
  double size=Grid[dim].fsize;
  int to=from+n, lo, up, i;
  struct S_subdomdesc *sd = SubDomainDesc;

  lo = map_irregular_range(p0*2.0-size, dim, from, to);
  up = map_irregular_range(p0*2.0+size, dim, lo, to);
  for (i=lo; i<up; ) {
    int n = sd[i].coord[dim].n;
    if (p0>=sd[i].coord[dim].fc[OH_LOWER] &&
        p0< sd[i].coord[dim].fc[OH_UPPER]) {
      if (dim<OH_DIMENSION-1) {
        int ret = map_irregular(p1, p2, 0.0, dim+1, i, n);
        if (ret>=0)  return(ret);
      }
      else
        return(sd[i].id);
    }
    i += n;
  }
  return(-1);
}
```

4.7.23 map_irregular_range()

map_irregular_range()

The function map_irregular_range(), called solely from map_irregular(), finds $m_{\min} = \min\{m \,|\, i_0 \le m < i_n,\ (\delta_d^l(i(m)) + \delta_d^u(i(m))) \cdot \gamma_d > x'\}$ from SubDomainDesc$[[i_0, i_n)]$ where $x' =$ p $= 2x \pm \delta_d^{\max} \cdot \gamma_d$ for the d-th dimensional coordinate x of a particle, $d =$ dim, $i_0 =$ from, $i_n =$ to, and $\{i(m),\ \delta_d^l(i(m)) \cdot \gamma_d,\ \delta_d^u(i(m)) \cdot \gamma_d\} =$ SubDomainDesc$[m]$.{id, coord$[d]$.fc$[\{0,1\}]$}.

At first, if $i_0 = i_n$ to mean $[i_0, i_n) = \emptyset$ or $(\delta_d^l(i(i_n{-}1)) + \delta_d^u(i(i_n{-}1))) \cdot \gamma_d \le x'$ to mean that no subdomains satisfiy the criterion, we return i_n to indicate nothing is found. On the other hand if $(\delta_d^l(i(i_0)) + \delta_d^u(i(i_0))) \cdot \gamma_d > x'$ to mean that i_0 satisfies the criterion, we simply return i_0 being the minimum one.

Otherwise, i.e., $(\delta_d^l(i(i_0)) + \delta_d^u(i(i_0))) \cdot \gamma_d \le x' < (\delta_d^l(i(i_n{-}1)) + \delta_d^u(i(i_n{-}1))) \cdot \gamma_d$, we start a binary search to find $m_{\min}$ starting with $j_0 = i_0$, $j_n = i_n$ and $j = \lfloor (j_0 + j_n)/2 \rfloor$ to examine j satisfies the criterion. Then we let $j_n \leftarrow j$ if satisfied, while $j_0 \leftarrow j$ otherwise, and let $j = \lfloor (j_0 + j_n)/2 \rfloor$ again, and repeat this process while $j_0 < j$. Since at initial j_0 is unsatisfiable while $j_n - 1$ is satisfiable, it is assured that we should reach $j = j_0$ and $j + 1 = j_n = m_{\min}$ to return j_n as the result.

```
static int
map_irregular_range(double p, int dim, int from, int to) {
```

```
  struct S_subdomdesc *sd = SubDomainDesc;
  int i;

  if (from==to) return(to);
  if (p<sd[from].coord[dim].fc[OH_LOWER]+sd[from].coord[dim].fc[OH_UPPER])
    return(from);
  if (p>=sd[to-1].coord[dim].fc[OH_LOWER]+sd[to-1].coord[dim].fc[OH_UPPER])
    return(to);
  for (i=(from+to)>>1; from<i; i=(from+to)>>1) {
    if (p<sd[i].coord[dim].fc[OH_LOWER]+sd[i].coord[dim].fc[OH_UPPER])
      to = i;
    else
      from = i;
  }
  return(to);
}
```

4.7.24 oh3_bcast_field()

`oh3_bcast_field_()` `oh3_bcast_field()`

The API functions `oh3_bcast_field_()` for Fortran and `oh3_bcast_field()` for C provide a simulator body calling them with a safe mechanism of broadcast communications in primary/secondary families of the local node. The broadcasts are performed on a field-array type f = `ftype` whose origins are pointed by `pfld` and `sfld` for the primary and secondary subdomains respectively. Both functions simply call `oh1_broadcast()` giving it the bases of the subarrays to be broadcasted which are offset by `FieldDesc`$[f']$`.bc.base` from the origins, their sizes `FieldDesc`$[f']$`.bc.size`$[p]$ for primary ($p = 0$) and secondary subdomains, and the data type `MPI_DOUBLE` commonly for primary/secondary ones, where $f' = f - 1$ for Fortan API while $f' = f$ for C API.

```
void
oh3_bcast_field_(void *pfld, void *sfld, int *ftype) {
  int base=FieldDesc[*ftype-1].bc.base;
  int *size=FieldDesc[*ftype-1].bc.size;

  oh1_broadcast((double*)pfld+base, (double*)sfld+base, size[0], size[1],
                MPI_DOUBLE, MPI_DOUBLE);
}
void
oh3_bcast_field(void *pfld, void *sfld, int ftype) {
  int base=FieldDesc[ftype].bc.base;
  int *size=FieldDesc[ftype].bc.size;

  oh1_broadcast((double*)pfld+base, (double*)sfld+base, size[0], size[1],
                MPI_DOUBLE, MPI_DOUBLE);
}
```

4.7.25 oh3_reduce_field()

`oh3_reduce_field_()` `oh3_reduce_field()`

The API functions `oh3_reduce_field_()` for Fortran and `oh3_reduce_field()` for C provide a simulator body calling them with a safe mechanism of summing-up reductions in primary/secondary families of the local node. The reductions are performend on a field-array type f = `ftype` whose origins are pointed by `pfld` and `sfld` for the primary and

secondary subdomains respectively. Both functions simply call `oh1_reduce()` giving it the bases of the subarrays to be reduced which are offset by `FieldDesc`$[f']$`.red.base` from the origins, their sizes `FieldDesc`$[f']$`.red.size`$[p]$ for primary ($p = 0$) and secondary subdomains, and the data type `MPI_DOUBLE` and operator `MPI_SUM` commonly for primary/ secondary ones, where $f' = f - 1$ for Fortan API while $f' = f$ for C API.

```
void
oh3_reduce_field_(void *pfld, void *sfld, int *ftype) {
  int base=FieldDesc[*ftype-1].red.base;
  int *size=FieldDesc[*ftype-1].red.size;

  oh1_reduce((double*)pfld+base, (double*)sfld+base, size[0], size[1],
             MPI_DOUBLE, MPI_DOUBLE, MPI_SUM, MPI_SUM);
}
void
oh3_reduce_field(void *pfld, void *sfld, int ftype) {
  int base=FieldDesc[ftype].red.base;
  int *size=FieldDesc[ftype].red.size;

  oh1_reduce((double*)pfld+base, (double*)sfld+base, size[0], size[1],
             MPI_DOUBLE, MPI_DOUBLE, MPI_SUM, MPI_SUM);
}
```

4.7.26 `oh3_allreduce_field()`

`oh3_allreduce_field_()` `oh3_allreduce_field()`

The API functions `oh3_allreduce_field_()` for Fortran and `oh3_allreduce_field()` for C are simply all-reduce versions of `oh3_reduce_field[_]()`. Therefore their implementations are different from the reduce-counterparts just in the point that they call `oh1_all_reduce()` instead of `oh1_reduce()`.

```
void
oh3_allreduce_field_(void *pfld, void *sfld, int *ftype) {
  int base=FieldDesc[*ftype-1].red.base;
  int *size=FieldDesc[*ftype-1].red.size;

  oh1_all_reduce((double*)pfld+base, (double*)sfld+base, size[0], size[1],
                 MPI_DOUBLE, MPI_DOUBLE, MPI_SUM, MPI_SUM);
}
void
oh3_allreduce_field(void *pfld, void *sfld, int ftype) {
  int base=FieldDesc[ftype].red.base;
  int *size=FieldDesc[ftype].red.size;

  oh1_all_reduce((double*)pfld+base, (double*)sfld+base, size[0], size[1],
                 MPI_DOUBLE, MPI_DOUBLE, MPI_SUM, MPI_SUM);
}
```

4.7.27 `oh3_exchange_borders()`

`oh3_exchange_borders_()` `oh3_exchange_borders()`

The API functions `oh3_exchange_borders_()` for Fortran and `oh3_exchange_borders()` for C provide a simulator body calling them with a mechanism of boundary communications

type c = `ctype` of field-array pointed by `pfld` between adjacent primary subdomains. It also performs broadcast communications in primary/secondary families of the local node to send receiving planes of the primary field-array to its children and to receive receiving planes of the secondary field-array pointed by `sfld` from its parent, if `bcast` is true.

```
void
oh3_exchange_borders_(void *pfld, void *sfld, int *ctype, int *bcast) {
  oh3_exchange_borders(pfld, sfld, *ctype-1, *bcast);
}
void
oh3_exchange_borders(void *pfld, void *sfld, int ctype, int bcast) {
  MPI_Status st;
  int d, lu;
  double *pf=(double*)pfld, *sf=(double*)sfld;
```

First we perform downward ($w = 0$) and upward ($w = 1$) communications for d-th dimensional boundary planes for all $d \in [0, D)$ and in ascending order of d from $d = 0$. Each communication is to send lower/upper sending planes to the node `Adjacent`$[d][w]$ referring to the communication parameters in `BorderExc`$[c][0][d][w]$`.send` and to receive them into upper/lower receiving planes from the node `Adjacent`$[d][1-w]$ with the parameters in `BorderExc`$[c][0][d][w]$`.recv`. The communication is performed by `MPI_Sendrecv()` if `count` elements of both parameter sets are positive, by `MPI_Send()` if only `send.count` is positive, or by `MPI_Recv()` if only `recv.count` is positive.

```
  for (d=0; d<OH_DIMENSION; d++) {
    for (lu=OH_LOWER; lu<=OH_UPPER; lu++) {
      int dst=Adjacent[d][lu], src=Adjacent[d][1-lu];
      struct S_borderexc *bx=&BorderExc[ctype][0][d][lu];
      int scount=bx->send.count;
      int rcount=bx->recv.count;
      if (scount && rcount)
        MPI_Sendrecv(pf+bx->send.buf, scount, bx->send.type, dst, 0,
                     pf+bx->recv.buf, rcount, bx->recv.type, src, 0,
                     MCW, &st);
      else if (scount)
        MPI_Send(pf+bx->send.buf, scount, bx->send.type, dst, 0, MCW);
      else if (rcount)
        MPI_Recv(pf+bx->recv.buf, rcount, bx->recv.type, src, 0, MCW, &st);
    }
  }
```

Next and finally, we perform the broadcast communications of receiving planes if we are in secondary mode (`currMode` mod $2 \neq 0$) and broadcasting is required by `bcast` $\neq 0$. Before broadcasting, we call `set_border_exchange()` to set up the type c communication parameters for the secondary subdomain if `BorderExc`$[c][1][0][0]$`.send.count` < 0 to mean that the local node just has been assigned a new secondary subdomain. Note that the `type` argument of `set_border_exchange()` is always `MPI_DOUBLE` because the exceptional case requiring `MPI_LONG_LONG_INT` for per-grid histogram with position-aware particle management does not perform the broadcast of receiving planes. Then we perform the broadcast communications for all $d \in [0, D)$ and $\beta = \{0, 1\}$ by `oh1_broadcast()` giving it arguments based on the parameters for the primary subdomain `BorderExc`$[c][0][d][\beta]$ and those for the secondary subdomain `BorderExc`$[c][1][d][\beta]$.

```
    if (Mode_PS(currMode) && bcast) {
      if (RegionId[1]>=0 &&
          BorderExc[ctype][1][OH_DIM_X][OH_LOWER].send.count<0)
        set_border_exchange(ctype, 1, MPI_DOUBLE);
      for (d=0; d<OH_DIMENSION; d++) {
        for (lu=OH_LOWER; lu<=OH_UPPER; lu++) {
          struct S_borderexc *bxp=&BorderExc[ctype][0][d][lu];
          struct S_borderexc *bxs=&BorderExc[ctype][1][d][lu];
          oh1_broadcast(pf+bxp->recv.buf, sf+bxs->recv.buf,
                        bxp->recv.count, bxs->recv.count,
                        bxp->recv.type, bxs->recv.type);
        }
      }
    }
  }
```

4.8 Level-4p Library Overview

The level-4p library is an extension of OhHelp for *position-aware* particle management so that particles in a particular *grid-voxel* are accommodated by a particular node responsible of the primary/secondary subdomain including the grid-voxel, as long as it is not so congested. Moreover, particles in each $pbuf(p, s)$ are sorted by their *grid-position*s, being the one-dimensional indices of their resident grid-voxels of (conceptual) D-dimensional array, so that the simulator body captures particles in a particular grid-voxel with the help of *per-grid histogram* of particle population provided by the level-4p library function `oh4p_transbound()`.

The fundamental mechanism of position-aware particle management is fairly simple as summarized below.

1. Simulator body calls `oh4p_map_particle_to_neighbor()` or `oh4p_map_particle_to_subdomain()` for each primary ($p = 0$) or secondary ($p = 1$) particle of species s at its move to a grid-position g so that the library accumulates the population in the grid-voxel at g in the *local* per-grid histogram entry namely $\mathcal{P}_L(p, s, g) =$ `NOfPGrid`$[p][s][g]$.

2. In `oh4p_transbound()`, the per-grid histograms in a helpand-helper family are accumulated and then its entries at boundary planes are exchanged between neighbors by `exchange_population()` to have the *complete* per-grid histogram named $\mathcal{P}_T(p, s, g) =$ `NOfPGridTotal`$[p][s][g]$ in the helpand of each subdomain.

3. Each helpand scans its per-grid histogram for its primary subdomain to assign a set of consecutive grid-voxels and the particles resinding in them to its primary family members by `make_recv_list()` so that the number of particles accommodated by each member node is approximately equal to that the OhHelp load balancing mechanisim requires. Then the assignment is broadcasted to helpers, exchanged between neighboring helpands and then broadcasted again to helpers of neighbors so that they recognize the destination of each particle they accommodate by `make_send_sched()`. The sending and receiving amount of particles are then exchanged by `exchange_xfer_amount()` among neighboring family menbers.

4. Prior to the particle transfer communication taken by `xfer_particles()`, each node scans its primary and secondary particles sorting those which will stay in the node by `move_and_sort_primary()` or `move_and_sort_secondary()` which moves particles from `Particles[]` to `SendBuf[]`. Then after the particle transfer, received particles are sorted by `sort_received_particles()` to have completely sorted particle buffer in `SendBuf[]` which becomes `Particles[]` in the next simulation step exchanging their roles.

The real implementaion is, however, a little bit more complicated due to various subjects summarized as follows.

primary mode If we will be in primary mode in the next simulation step, the particle transfer is simpler than above because all particles in a subdomain will be sent to the node responsible of the subdomain as primary one.

particle sorting The sorting prior to the particle transfer explained above minimizes the number of scans of `Particles[]`, i.e., we need just one scan. However it requires that `SendBuf[]` have the space large enough for both the particles accommodated in the next simulation step and those to be sent to other nodes. Since it is not ensured that `SendBuf[]`

is large enough though likely, we need to reverse the order of the sorting and transfer if insufficient. That is, we have to transfer particles in non-position-aware manner to have all particles to be accommodated in `Particles[]` and then sort and move them to `SendBuf[]` by `sort_particles()`.

anywhere accommodation Since we cannot have the complete per-grid histogram if we have anywhere accommodation or we need a histogram as large as the whole simulation space, the particle transfer in anywhere accommodation mode takes place in two phases; at first particles are transferred in ordinary non-position-aware manner only aware of their residing subdomains; then the second phase transfer takes place in each helpand-helper family in position-aware manner. In order to minimize the size of per-grid histogram, we have to define normal accmmodation more restrictedly so that it means all particles reside in their orignial subdomain or in the adjacent boundary plane of one of its neighbors.

hot-spot We cannot be perfectly position-aware if a grid-voxel is too congested, or a node to which the voxel is assigned should have too many particles which can be all in simulated system in the most extreme case. Therefore, we could have to split the set of particles in a too congested *hot-spot* into subsets whose cardinalities are at least a threshold $P_{hot} = \texttt{gridOverflowLimit}/2$ given by the simulator body. The threshold plays two roles; every split particle subset for a grid-voxel is large enough and not less than P_{hot} to ensure the satisfaction of, e.g., the law of large numbers for Monte Carlo collision in the grid-voxel; and the particle population for a node is not too large and the excess over that expected by OhHelp balancing mechanism is less than $2P_{hot}$.

When we have a hot-spot, the nodes which have and will have the particles in them should know how many particles reside and will reside in each node involved. Since we cannot gather/scatter all per-grid histograms of involved nodes or the total size of them can be as large as the histogram for the whole simulation space, we have to gather/scatter only the histogram entry of the hot-spot. Furthermore, we have to be careful to perform the gather/scatter to avoid unnecessarily frequent or system-wide collective communications and also to avoid deadlocks or unnecessary serializations on possibliy multiple inter-family collective communications. Therefore, we need to design a sophisticated hot-spot management scheme and a set of functions such as `gather_hspot_recv()`, `gather_hspot_send()`, `scatter_hspot_send()` and `scatter_hspot_recv()`.

rebalance In order to fully exploit the restricted definition of normal accommodation and to minimize the amount of inter-node communications for a subdomain as well as the number of nodes involved in them, we have to strictly manage the set of nodes which can send/receive particles residing in the subdomain. This management requires a special care when the helpand-helper tree is reconfigured because, for example, a node n may have to send particles to another node which *becomes* a helper of a neighbor of the old helpand of n, or n may have to receive particles from another node which *was* a helper of a neighbor of the new helpand of n. These inter-family communication with transitional state of the family tree requires us to keep the neighbors of old helpand in `Neighbors`[2], to keep the transtive neighboring configuration in `RealDstNeighbors`[1][] and `RealSrcNeighbors`[1][] by `update_real_neighbors()`, to do additional work for the new helpand different from the old one in `make_send_sched()`, `gather_hspot_send()` and `scatter_hspot_recv()`, and so on.

4.9 Header File ohhelp4p.h

The header file of level-4p library, ohhelp4p.h, `#define`s a few macros for per-grid histogram and grid-position. Then it declares global variables and their `struct`ures used in level-4p library codes to keep per-grid histograms and their shapes, to make the particle transfer schedule, to store neighboring node information, and to check the boundary conditions of the system domain. Finally it gives prototypes of API functions.

4.9.1 Constants

At first we define the following constants.

`OH_PGRID_EXT`

- The constant `OH_PGRID_EXT` $= e^g = 1$ is the inner extension of per-grid histogram to mean that particles in a subdomain can move at most one grid outside the subdomain. Though it is very unlikely that we have a PIC simulator in which paticles can travel two or more grids with position-aware particle management, modifying this definition may cope with such a imaginary implementation. Note that the outer extension of per-grid histogram is $2e^g$ to have additional receiving plane(s) outside the sending plane(s) both of which are e^g thick.

`OH_NBR_SELF`

- The constant `OH_NBR_SELF` $= \sum_{d=0}^{D-1} 3^d = \lfloor 3^D/2 \rfloor$ is the index of `Neighbors`$[3^D]$ of a node and its relatives for the node itself. It is used to know whether we refer to the node itself when, for example, we scan its neighbors.

Here we revisit a few other constants/switches related to level-4p extension.

`OH_LIB_LEVEL_4P`

- The switch `OH_LIB_LEVEL_4P` can be defined in oh_config.h to declare that the OhHelp library should be configured with level-4p extension, as discussed in §3.3.

`OH_POS_AWARE`

- The switch `OH_POS_AWARE` can be defined in ohhelp1.h to make lower level libraries position-aware. So far it is equivalent to `OH_LIB_LEVEL_4P` because ohhelp1.h `#define`s it iff `OH_LIB_LEVEL_4P` is defeined as well as discussed in §4.2.2, but may be not in future with further or another extensions.

`STATS_TB_SORT`

- The constant `STATS_TB_SORT` can be defined in oh_stats.h as the timing statistics key to measure the time for particle sorting as discussed in §3.10.1.

```
#define OH_PGRID_EXT 1
#define OH_NBR_SELF (OH_NEIGHBORS>>1)
```

4.9.2 Macros for Grid-Position

For the particle sorting, we need that the particle `struct`ure `S_particle` has the grid-position of the particle. Moreover, with normal accommodation, we need to know the neighbor index k of the subdomain m where the particle resides. To minimize the spatial impact on the particle buffers, we encode the grid-position and k or m in the `nid` element of `S_particle`, rather than add an element for grid-position nor k. More specifically, the `nid` element i of a particle residing in the subdomain m, which can be the k-th neighbor of the

local node n's primary/secondary subdomain $n' \in \{n, parent(n)\}$, and in the subdomain's grid-voxel of index g has the following.

$$gidx(x,y,z) = \begin{cases} x & D=1 \\ x + (\delta_x^{\max} + 4e^g)y & D=2 \\ x + (\delta_x^{\max} + 4e^g)(y + (\delta_y^{\max} + 4e^g)z) & D=3 \end{cases}$$

$$\Gamma = \lfloor \log_2 gidx(\delta_x^{\max}-1, \delta_y^{\max}-1, \delta_z^{\max}-1) \rfloor + 1$$

$$\sigma = \begin{cases} k & m \text{ is } k\text{-th neighbor of } n' \text{ definitely} \\ m + 3^D & \text{otherwise} \end{cases}$$

$$i = \sigma \cdot 2^\Gamma + g$$

That is, Γ has the minimum number of bits to represent the largest possible one-dimensional index of per-grid histogram whose origin is for $(0, \cdots, 0)$ of the local coordinate of a subdomain and size is $G = \prod_{d=0}^{D-1}(\delta_d^{\max} + 4e^g)$ including e^g thick sending and receiving planes at lower and upper boundaries of the subdomain. Therefore, the *subdomain code* σ for `nid` i can be obtined by $\lfloor i/2^\Gamma \rfloor$ or by shifting right i by Γ bits, while g is (i mod Γ) or is extracted by a bitwise-AND of i and $2^\Gamma - 1$. Then the subdomain identifier m is obtained from σ by;

$$m = \begin{cases} \texttt{AbsNeighbors}[p][\sigma] & \sigma < 3^D \\ \sigma - 3^D & \sigma \geq 3^D \end{cases}$$

where `AbsNeighbors`$[p][k]$ is the subdomain identifier of k-th neighbor of the local node's primary ($p = 0$) or secondary ($p = 1$) subdomain.

In addition, the `nid` element i can temporarily have a special value in its subdomain code part, $(N + 3^D) + \sigma$, for secondary particles injected into the subdomain m represented by σ, to distinguish them from primary injected particles. Therefore, the largest possible i is $(2(N + 3^D) - 1) \cdot 2^\Gamma$ which should not be greater than the largest `int` value, $2^{31} - 1$, or we need to represent `nid` by 64-bit integer `long␣long␣int`.

Prior to showing the macros defined in ohhelp4p.h, here we revisit a switch, a data type, variables and macros defined in other header files.

`OH_BIG_SPACE`

- The switch `OH_BIG_SPACE` can be defined in oh_config.h to declare that per-grid histograms are too large to represent `nid` by `int` and thus it should be `long␣long␣int`.

`OH_nid_t`

- The data type `OH_nid_t` is `int` if `OH_BIG_SPACE` is undefined by default, or `long␣long␣int` otherwise. The `typedef` of this data type is in oh_part.h, while its Fortran counterpart oh_type.F90 just has `#ifdef`/`#else`/`#endif` construct to declare the `nid` element with corresponding size.

`logGrid` `gridMask`

- The integer variable `logGrid` and `gridMask` declared in ohhelp2.h have Γ and $2^\Gamma - 1$ respectively. The values are assigned to them by `init4p()` and the variables are referred to only through the macros discussed below.

`AbsNeighbors`

- The two dimensional array `AbsNeighbors`$[2][3^D]$ declared in ohhelp2.h has the following value in its element $[p][k]$.

$$\texttt{AbsNeighbors}[p][k] = \begin{cases} \texttt{Neighbors}[p][k] & \texttt{Neighbors}[p][k] \geq 0 \\ -(\texttt{Neighbors}[p][k] + 1) & \texttt{Neighbors}[p][k] < 0 \end{cases}$$

That is the array is an always-positive version of `Neighbors`$[][]$ unaware of multiple occurrences of subdomains in a neighbor set. Therefore, the elements $[0][]$ are initialized

by `init4p()` through its callee `update_neighbors()` after `Neighbors`[0][] are initialized, and [1][] are initialized/updated in `rebalance4p()` or `exchange_particles4p()` when `Neighbors`[1][] are set to neighbors of the newly assigned local node's secondary subdomain through their callee `update_neighbors()`. Besides the initializer/updater `update_neighbors()`, the array is reffered to by `oh4p_map_particle_to_neighbor()` directly and other functions through the macro `Subdomain_Id()` or `Neighbor_Subdomain_Id()`.

`Decl_Grid_Info()`

- The macro `Decl_Grid_Info()` with no arguments, defined in ohhelp2.h, declares local variables named `gridmask` and `loggrid` to *cache* the corresponding global variables `gridMask` and `logGrid` in them respectively, and `subdomid` to have σ temporarily, so that the local variables are referred to by the related macros. The macro is used in the following functions;

  `rebalance4p()`, `count_population()`, `sort_particles()`, `move_and_sort_primary()`, `sort_received_particles()`, `move_to_sendbuf_sec4p()`, `move_to_sendbuf_uw4p()`, `move_to_sendbuf_dw4p()`, `move_and_sort_secondary()`, `oh4p_map_particle_to_neighbor()`, `oh4p_map_particle_to_subdomain()`, `oh4p_remove_mapped_particle()`.

`Subdomain_Id()`

- The macro `Subdomain_Id`(i, p), defined in ohhelp2.h, extracts the subdomain code σ of `nid` element i of a primary ($p = 0$) or secondary ($p = 1$) particle by calculating $\sigma = \lfloor i/2^\Gamma \rfloor$ and then translates it to the subdoamin identifier represented by σ. This macro is used in `move_and_sort_primary()` and `oh4p_remove_mapped_particle()`, when it is not sure that $\sigma \in [0, 3^D)$ due to secondary injected particles for the former and anywhere-accommodated ones for the latter.

`Primarize_Id()`

- The macro `Primarize_Id`(π, m), defined in ohhelp2.h, removes the secondary particle flag attached to the particle pointed by π by subtracting $(N + 3^D) \cdot 2^\Gamma$ from its `nid`. It also gives the identifier m of the subdomain into which the particle is injected. This macro is used in `rebalance4p()`, `move_and_sort_primary()` and `oh4p_remove_mapped_particle()`, which need m as well as the primarization.

Note that the last three macros neither do anything nor are defined at all if `OH_POS_AWARE` is not defined.

Now we show a few macros used only by functions in level-4p library.

`Grid_Position()`

- The macro `Grid_Position`(i) extracts the grid-position part of i by performing bitwise-AND on it with $2^\Gamma - 1$ = `gridMask`. This macro is used in `rebalance4p()`, `count_population()`, `sort_particles()`, `move_and_sort_primary()`, `sort_received_particles()`, `oh4p_remove_mapped_particle()` directly, and in `move_to_sendbuf_sec4p()`, `move_to_sendbuf_uw4p()`, `move_to_sendbuf_dw4p()` and `move_and_sort_secondary()` through the macro `Move_Or_Do()`.

`Combine_Subdom_Pos()`

- The macro `Combine_Subdom_Pos`(σ, g) combines the subdomain code σ and the grid-position g to have $\sigma \cdot 2^\Gamma + g$. This macro is used in `count_population()`, `oh4p_map_particle_to_neighbor()` and `oh4p_map_particle_to_subdomain()`.

`Primarize_Id_Only()`

- The macro `Primarize_Id_Only`(π) removes the flag for secondary injected particles as its relative `Primarize_Id()` does, but does not give the subdomain identifier to its

invoker which does not care that. This macro is used in `move_to_sendbuf_sec4p()` and `move_and_sort_secondary()`.

`Secondarize_Id()`

- The macro `Secondarize_Id`(π) flags that the particle pointed by π is injected to the secondary subdomain of the local node, or other (neighbor of secondary subdomain, very likely) subdomain as a secondary particle, by adding $(N+3^D)\cdot 2^\Gamma$ to its `nid`. This macro is used in `rebalance4p()`, `oh4p_map_particle_to_neighbor()` and `oh4p_map_particle_to_subdomain()`.

`Secondary_Injected()`

- The macro `Secondary_Injected`(i) is replaced with true iff i being the `nid` of a particle has a subdomain code not less than $N + 3^D$, i.e., the particle is injected as a secondary one. This macro is used in `rebalance4p()`. `move_to_sendbuf_sec4p()` and `move_and_sort_secondary()`.

`Neighbor_Subdomain_Id()`

- The macro `Neighbor_Subdomain_Id`(i, p) is a simplified version of `Subdomain_Id()` to be used if it is assured that the subdomain code part in i is in $[0, 3^D)$, i.e., if the macro is used with normal accommodation or after the first phase non-position-aware particle transfer with anywhere one, and i is of a particle not being secondary injected or that primarized. Therefore, the macro is simply replaced with `AbsNeighbors`$[p][\lfloor i/2^\Gamma \rfloor]$. This macro is used in `move_and_sort_primary()` directly, and in `move_to_sendbuf_sec4p()`, `move_to_sendbuf_uw4p()`, `move_to_sendbuf_dw4p()` and `move_and_sort_secondary()` through the macro `Move_Or_Do()`.

```
#define Grid_Position(ID)         ((ID)&gridmask)
#define Combine_Subdom_Pos(ID, G) (((OH_nid_t)(ID)<<loggrid) + (G))
#define Primarize_Id_Only(P) \
  (P)->nid -= (OH_nid_t)(nOfNodes+OH_NEIGHBORS)<<loggrid
#define Secondarize_Id(P) \
  (P)->nid += (OH_nid_t)(nOfNodes+OH_NEIGHBORS)<<loggrid
#define Secondary_Injected(ID) \
  ((ID>>loggrid)>=nOfNodes+OH_NEIGHBORS)
#define Neighbor_Subdomain_Id(ID, PS) \
  AbsNeighbors[PS][(ID)>>loggrid]
```

4.9.3 Per-Grid Histograms and Related Variables

Next, we declare the following arrays of per-grid histograms and related variables.

`PbufIndex`

- The element of $[p][s]$ of the integer array `PbufIndex`$[2][S]$ has the head index of $pbuf(p, s)$ in `Particles`[]. That is, each of its element has the following value for the local node n.

$$\texttt{PbufIndex}[p][s] = \sum_{q=0}^{p-1}\sum_{t=0}^{S-1} \texttt{TotalP}[q][t] + \sum_{t=0}^{s-1} \texttt{TotalP}[p][t]$$

  Note that the array has an additional conceptual element [2][0] defined by the equation above and thus having Q_n = `totalParts`. The array is initialized by `oh4p_init()` with `NULL` and then allocated by the first call of `transbound4p()` which also sets all elements in each call, while referred to by `oh4p_map_particle_to_neighbor()`, `oh4p_map_particle_to_subdomain()` and `oh4p_remove_mapped_particle()` through the macro `Check_Particle_Location()`.

NOfPGrid
NOfPGridTotal

- The `dint` type double pointer arrays `NOfPGrid`[2] and `NOfPGridTotal`[2] are for the per-grid histograms of conceptually three-dimensional arrays of $[2][S][G]$ whose element $[p][s][g]$ has the number of primary ($p = 0$) or secondary ($p = 1$) particles of species s in the grid-voxel whose index is g, namely $\mathcal{P}_L(p, s, g)$ and $\mathcal{P}_T(p, s, g)$ respectively. Their body `dint` arrays and pointer arrays are allocated by `init4p()` using `Allocate_NOfPGrid()` which also zero-clears their bodies. `NOfPGrid`[][][] is also zero-cleared by `transbound4p()` as one of its post-process to give the base of the counting. Then `oh4p_map_particle_to_neighbor()` or `oh4p_map_particle_to_subdomain()` increments entries of `NOfPGrid`[][][] for particles accommodated by the local node, while `oh4p_remove_mapped_particle()` may decrement some of them to cancel the increments for particles which were once recoginized existing but then vanish. Therefore at the call of `oh4p_transbound()`, `NOfPGrid`[][][] should have the per-grid histogram after the one-step travel of particles.

  Then in the functions called from `transbound4p()`, these two arrays play the following roles depending on the mode in the last and next simulation steps (m_c and m_n) and the accommodation pattern a.

  - a = normal, m_n = primary, m_c = primary
    The complete per-grid histogram is bulit in `NOfPGrid`[0][][] by `exchange_population()` through the neighboring communication of its boundary plane. Then `NOfPGrid`[][][] is referred to by `sort_particles()` or `move_and_sort_primary()` for sorting.
  - a = normal, m_n = primary, m_c = secondary
    The complete per-grid histogram is built in `NOfPGridTotal`[0][][] by summing up `NOfPGrid`[][][] in the primary family members and by exchanging boundary planes of neighbors in `exchange_population()` and its callee `reduce_population()`. Then `NOfPGridTotal`[0][][] is referred to by `sort_particles()` or `move_and_sort_primary()` for sorting, while `NOfPGrid`[][][] is kept unchanged but not referred to.
  - a = normal, m_n = secondary, m_c = primary
    The contents of `NOfPGrid`[0][][] is copied into `NOfPGridTotal`[0][][] in which the complete per-grid histogram is built by `exchange_population()`. Then `NOfPGridTotal`[0][][] is referred to by `sched_recv()` to determine the distribution of primary particles among primary family members, and then broadcased to new helpers' `NOfPGridTotal`[1][][] in `make_recv_list()` so that helpers knows the number of particles in each grid-voxel which they have to host in `make_send_sched_body()`. On the other hand, `NOfPGrid`[0][][] is kept unchanged to be referred to know the number of particles in each grid-voxel, by `make_send_sched_body()`, `gather_hspot_send_body()`, `scatter_hspot_send()` and `scatter_hspot_recv_body()`, in case that they are sent out to other nodes.
  - a = normal, m_n = secondary, m_c = secondary
    The complete per-grid histogram is built in `NOfPGridTotal`[0][][] by summing up `NOfPGrid`[][][] in the primary family members and by exchanging boundary planes of neighbors in `exchange_population()` and its callee `reduce_population()`. Then both arrays are referred to as discussed above for m_c = primary.
  - a = anywhere, m_n = primary
    Since the contents of `NOfPGrid`[0][][] is not useful, the complete per-grid his-

togram is rebuilt in it by `count_population()` after the non-position-aware particle transfer. Then it is referred to by `sort_particles()` for sorting.

– a = anywhere, m_n = secondary
Since the contents of `NOfPGrid[][][]` is not useful, the local per-grid histogram is rebuilt in it by `count_population()` after the non-position-aware particle transfer. Then the complete per-grid histogram is built in `NOfPGridTotal`[0]`[][]` by `reduce_population()` keeping `NOfPGrid[][][]` unchanged. Then both arrays are referred to as discussed in the case of a = normal, m_n = secondary and m_c = primary.

In addition, each of two arrays has another role. If the next step is in secondary mode, `NOfPGrid[][][]` is modified by `make_send_sched_body()` and `scatter_hspot_recv_body()` to have one of the followings after its original contents are examined.

– 0 means that the particles in the grid-voxel stays in the local node.
– A positive number $(pS + s)N + m + 1$ means that the particles of species s in the grid-voxel is sent to the node m as its primary ($p = 0$) or secondary ($p = 1$) particle.
– A negative number $-(i + 1)$ means that the hot-spot sending schedule for the particles in the grid-voxel is found in `CommList`[i].

The functions `move_to_sendbuf_sec4p()`, `move_to_sendbuf_uw4p()`, `move_to_sendbuf_dw4p()` and `move_and_sort_secondary()` refer to `NOfPGrid[][][]` with this role.

On the other hand, `NOfPGridTotal[][][]` is modified by `sort_particles()`, `move_and_sort_primary()` or `move_and_sort_secondary()` to have the sorting index of `SendBuf[]` to which the particle in the grid-voxel is moved. That is, for the local node n, it is initialized as follows and then its element is incremented each time a particle in the grid-voxel is moved by the functions above and `sort_received_particles()`.

$$n(p') = \begin{cases} n & p' = 0 \\ parent(n) & p' = 1 \end{cases}$$

$$\delta_d(m) = \delta_d^u(m) - \delta_d^l(m)$$
$$\mathcal{R}_d(p') = [0,\ \delta_d(n(p')))$$
$$\mathcal{G}(p') = \{g \,|\, g = gidx(x, y, z),\ x \in \mathcal{R}_x(p'),\ y \in \mathcal{R}_y(p'),\ z \in \mathcal{R}_z(p')\}$$
$$\texttt{NOfPGridTotal}[p][s][g] = \sum_{p'=0}^{p-1} \sum_{s'=0}^{S-1} \sum_{g' \in \mathcal{G}(p')} \texttt{NOfPGridOut}[p'][s'][g'] + \sum_{s'=0}^{s-1} \sum_{g' \in \mathcal{G}(p)} \texttt{NOfPGridOut}[p][s'][g'] + \sum_{g' \in \mathcal{G}(p),\ g' < g} \texttt{NOfPGridOut}[p][s][g']$$

`NOfPGridOut`

- The `int` type double pointer array `NOfPGridOut`[2] is for the local per-grid histogram of conceptually three-dimensional array of [2][S][G] whose element [p][s][g] has the number of primary ($p = 0$) or secondary ($p = 1$) particles of species s in the grid-voxel whose index is g, namely $\mathcal{P}_O(p, s, g)$. The per-grid histogram reflects the particle transfer and is given to the simulator body so that it captures particles in a particular grid-voxel.

The body `int` array of `NOfPGridOut[][][]` is given by the simulator body through the double-pointer argument of `pghgram` of `oh4p_per_grid_histogram()`, or is allocated by the function using `Allocate_NOfPGrid()` if the argument points `NULL`. In both cases, its pointer arrays are allocated by the function and the body is zero-cleared. Then if the next step is in primary mode, each entry is set to the number of particles by `sort_particles()` or `move_and_sort_primary()`. Otherwise, each entry is set by `make_send_sched_body()`, `scatter_hspot_send()` or `scatter_hspot_recv_body()`, and then referred to by `sort_particles()`, `move_and_sort_primary()` or `move_and_sort_secondary()` to build the sorting index in `NOfPGridTotal[][][]`.

`S_griddesc` `GridDesc`

- The array `GridDesc`[3] of the `S_griddesc` structure with the following elements has the shape information of per-grid histogram for a subdomain m.
  - `x`, `y` and `z` have $\delta_d(m) = \delta_d^u(m) - \delta_d^l(m)$ where $d = 0$ for `x`, $d = 1$ for `y`, and $d = 2$ for `z`. That is, each element has the upper bound of m's local coordinate for the corresponding dimension. Note that if $D < 3$, `y` and/or `z` is set to 0. Also note that if the sudomain m does not exist because, for example, it is the secondary subdomain of the root of helpand-helper tree, these elements are set to $-4e^g$ to ensure that adding any possible upper bound offset in $[0, 2e^g)$ to it should give a result not greater than $-2e^g$, the absolute lower bound of the D-dimensional index of per-grid histogram.
  - `w`, `d` and `h` have $\delta_d^{\max} + 4e^g$ where $d = 0$ for `w`, $d = 1$ for `d`, and $d = 2$ for `h`. That is, each element has the size of conceptual D-dimensional element array of the per-grid histogram. The values of the elements are common for all `GridDesc[]` array elements. Note that if $D < 3$, `d` and/or `h` is set to 1.
  - `dw` is `w` $\times$ `d`. Therefore, the one dimensional per-grid histogram index $gidx(x, y, z)$ is $x + \texttt{d} \cdot y + \texttt{dw} \cdot z$.

  Each array element is set by `set_grid_descriptor()` called by one of the following functinos.
  - [0] for the primary subdomain is permanently set by the call from `init4p()`.
  - [1] for the secondary subdomain is set by the call from `rebalance4p()` or `exchange_particles4p()` to reflect the helpand-helper reconfiguration with normal or anywhere accommodation respectively.
  - [2] for the newly assigned secondary subdomain due to the helpand-helper reconfiguration is set by the call from `make_recv_list()`, keeping [1] unchanged for the transitional state.

  Then the array is referred to by `oh4p_per_grid_histogram()`, `exchange_population()`, `sched_recv()`, `make_send_sched_body()`, `gather_hspot_recv()`, `oh4p_map_particle_to_neighbor()`, `oh4p_map_particle_to_subdomain()` directly, and in `transbound4p()`, `add_population()`, `count_population()`, `sort_particles()`, `move_and_sort_primary()` and `move_and_sort_secondary()` through the macro `For_All_Grid()` or `For_All_Grid_Abs()`.

```
EXTERN int *PbufIndex;                                  /* [2*ns+1] */
EXTERN dint **NOfPGrid[2], **NOfPGridTotal[2];          /* [2][ns][z][y][x] */
EXTERN int **NOfPGridOut[2];                            /* [2][ns][z][y][x] */
```

```
struct S_griddesc {
  int x, y, z, w, d, h, dw;
};
EXTERN struct S_griddesc GridDesc[3];
```

In addition, we use the following variables a little bit diffenently from their usages in lower level libraries.

Particles
SendBuf

- Each of two particle buffers `Particles`[P_{lim}] and `SendBuf`[P_{lim}] is now a half of the larger buffer of $2P_{lim}$ particles. Therefore, the simulator body must give the (double) pointer to this larger buffer through the argument `pbuf` of `oh4p_init()`, or will receive the pointer to the buffer allocated by `oh4p_init()`. Moreover, `Particles`[] and `SendBuf`[] exchanges their roles each time `oh4p_transbound()` is called. That is, `Particles`[] has all partcles accommodated by the local node at the call of the function, but those which stay in the local node in the next step are moved to `SendBuf`[] together with those received from other nodes to make `SendBuf`[] becomes `Particles`[] in the next step by `transbound4p()`. Therefore, simulator body must switch the paticle buffer which it processes from/to the first half to/from the second half each time it calls `oh4p_transbound()`.

  The functions `sort_particles()`, `move_and_sort_primary()`, `sort_received_particles()`, `move_to_sendbuf_sec4p()`, `move_to_sendbuf_uw4p()`, `move_to_sendbuf_dw4p()`, `move_and_sort_secondary()` and `xfer_particles()` refer to and/or modify both `Particles`[] and `SendBuf`[] directly or indirectly, while `count_population()`, `oh4p_map_particle_to_neighbor()`, `oh4p_map_particle_to_subdomain()`, `oh4p_inject_particle()` and `oh4p_remove_mapped_particle()` refer to and/or modify only `Particles`[] directly or indirectly.

nOfLocalPLimit

- The variable `nOfLocalPLimit` = P_{lim} is now calculated by `oh4p_max_local_particles()` to let it

$$P_{lim} = \max(\lceil \overline{P}(100+\alpha)/100 \rceil\ ,\ \overline{P} + \texttt{minmargin}) + 4P_{hot}$$

  to ensure we have the margins of $2P_{hot}$ for each of primary and secondary particle sets. Moreover, `oh4p_max_local_particles()` keeps its calculation result in a variable `nOfLocalPLimitShadow` private to level-4p functions so that `init4p()` ensures that `oh4p_max_local_particles()` had been called and its result is not less than that given by `maxlocalp` arguement of `init4p()`. The variable is referred to by level-4p functions `try_primary4p()`, `exchange_particles4p()` and `oh4p_inject_particle()`.

NOfPLocal

- The per-subdomain local particle population histogram `NOfPLocal`[2][S][N] does not change its role but it is now private to level-4p library. Therefore, `oh4p_init()` does not has the argument `nphgram` which lower-level's counterparts have, and counting the per-subdomain population is perfectly up to the library functions, `oh4p_map_particle_to_neighbor()`, `oh4p_map_particle_to_subdomain()` and `oh4p_remove_mapped_particle()`. In level-4p library `NOfPLocal`[][][] is referred to by `transbound4p()`, `try_primary4p()` and `move_and_sort_primary()`.

RecvBufBases

- The pointer array `RecvBufBases`[2][S] each element [p][s] of which points $rbuf(p,s)$ does not change its role but it is now has one extra element conceptually [2][0] so that `sort_received_particles()` can know the tail of $rbuf(p,s)$ by referring to the element at (real) one-dimensional index [pS+s+1].

This extra element is set by `move_and_sort_primary()` or `move_and_sort_secondary()` and referred to by `sort_received_particles()`, while other elements are referred to also by them and `move_to_sendbuf_sec4p()`, `move_to_sendbuf_uw4p()` and `xfer_particles()`.

- Besides `Particles[]`, `SendBuf[]`, `nOfLocalPLimit`, `NOfPLocal[][][]` and `RecvBufBases[][]`, some other variables for particle buffers and population are also used in the level-4p functions in their original meanings as follows.
  - `NOfPrimaries[][]` in `make_recv_list()`, `sched_recv()` and `move_and_sort_primary()`.
  - `TotalPGlobal[]` in `try_primary4p()`.
  - `TotalP[][]` by `transbound4p()`, `move_and_sort_primary()`, `move_to_sendbuf_uw4p()`, `move_to_sendbuf_dw4p()` and `move_and_sort_secondary()`.
  - `TotalPNext[][]` by `transbound4p()`, `make_send_sched()`, `make_send_sched_body()`, `scatter_hspot_send()`, `scatter_hspot_recv_body()`, `count_population()`, `sort_particles()`, `move_and_sort_primary()`, `move_to_sendbuf_uw4p()` and `move_to_sendbuf_dw4p()`.
  - `primaryParts` in `try_primary4p()`, `move_to_sendbuf_sec4p()` and `move_and_sort_secondary()` together with the pointer to its shadow `secondaryBase`, while in `count_population()` solitarily.
  - `totalParts` in `transbound4p()` with `totalLocalParticles`, in `move_to_sendbuf_sec4p()`, `oh4p_map_particle_to_subdomain()`, `oh4p_inject_particle()` and `oh4p_remove_mapped_particle()` directly, and in `oh4p_map_particle_to_neighbor()` through the macro `Check_Particle_Location()`.
  - `SendBufDisps[][]` in `move_and_sort_primary()`.
  - `nOfInjections` in `transbound4p()`, `rebalance4p()`, `move_and_sort_primary()`, `move_to_sendbuf_sec4p()`, `move_and_sort_secondary()`, `oh4p_inject_particle()` and `oh4p_remove_mapped_particle()` directly, and in `oh4p_map_particle_to_neighbor()` and `oh4p_map_particle_to_subdomain()` through the macro `Check_Particle_Location()`.
  - `InjectedParticles[][]` in `transbound4p()`, `rebalance4p()`, `move_and_sort_primary()`, `oh4p_map_particle_to_neighbor()`, `oh4p_map_particle_to_subdomain()` and `oh4p_remove_mapped_particle()`.

`nOfFields` `FieldTypes` `FieldDesc`

- The integer/structure arrays for field-arrays, `FieldTypes`$[F+1][7]$ and `FieldDesc`$[F]$, and their size $F =$ `nOfFields` are extended to have one extra element for per-grid histogram. That is, `init4p()` *intercepts* its argument `ftypes` to make its substance `FieldTypes[][]` have the following additional entry $f = F - 1$ to its tail.
  - $[0] = \varepsilon(f) = 1$ means that an entry of per-grid histogram has one (32-bit or 64-bit integer) element.
  - $[1{:}2] = \{e_l(f), e_u(f)\} = \{0, 0\}$ means per-grid histogram does not have any special extensions.

– [3:4] = $\{e^b_l(f), e^b_u(f)\} = \{0, 0\}$ means the broadcast of per-grid histogram does not have any extensions.

– [5:6] = $\{e^r_l(f), e^r_u(f)\} = \{-e^g, e^g\}$ means the reduction of per-grid histogram should include its sending planes of e^g thick.

Then the `FieldTypes[][]` is passed to `init3()`, which calls `init_fields()` to allocate and initialize `FieldDesc[]` including the extra field of per-grid histogram but not to allocate `FieldTypes[][]` because it is allocated by `init4p()`. This call of `init_fields()` also makes `fsizes`$[F-1][][]$ has the D-dimensional size information of per-grid histogram by which the simulator body can allocate and access `NOfPGridOut[][][]` associated with the argument `pghgram` of `oh4p_per_grid_histogram()`.

In addition, `init4p()` calls `adjust_field_descriptor()` after the call of `init3()` to add $(S-1)\prod_{d=0}^{D-1} \Phi_d(F-1) = (S-1)G$ to `FieldDesc`$[F-1]$`.{bc,red}.size`[0] so that the broadcast and reduction for per-grid histogram are performed on the whole of $[p][S][G]$ rather than the one array element $[p][s][G]$. This adjustment is also made by `update_descriptors()` called from `exchange_particles4p()` or `make_recv_list()` when the helpand-helper reconfiguration gives a new helpand to the local node and we had anywhere or normal accommodation respectively. Note that the elements of `FieldDesc`$[F-1]$`.{bc,red}` are not referred to by the level-3 API functions of collective communication, but by level-4p functions `reduce_population()` and `make_recv_list()` because the element data type of per-grid histogram is `MPI_LONG_LONG_INT` rather than `MPI_DOUBLE`[59].

`nOfExc`
`BoundaryCommFields`
`BoundaryCommTypes`
`BorderExc`

- The integer/structure arrays for the boundary communication of field-arrays, `BoundaryCommFields`$[C+1]$, `BoundaryCommTypes`$[C][B][2][3]$ and `BorderExc`$[C][2][D][2]$, and their size $C =$ `nOfExc` are extended to have one extra element for per-grid histogram. That is, `init4p()` *intercepts* its arguments `cfields` and `ctypes` to make their substances `BoundaryCommFields[]` and `BoundaryCommTypes[][][][]` have one additional entry $C-1$ for each, to have $F-1$ for the former and the followings for the latter.

  – $[0][0][] = \{-e^g, e^g, e^g\}$ means that the sending plane(s) of the downward communication is just below the lower boundary plane and receiving plane(s) is just above the upper sending plane(s).

  – $[0][1][] = \{0, -2e^g, e^g\}$ means that the sending plane(s) of the upward communication is just above the upper boundary plane and receiving plane(s) is just below the lower sending plane(s).

  Note that `BoundaryCommTypes`$[C-1][b][][]$ for all $b \in [1, B)$ are set to 0 to mean no communication is performed for non-periodical system boundaries.

  Then the `BoundaryCommFields[]` and `BoundaryCommTypes[][][][]` are passed to `init3()`, which calls `init_fields()` to allocate and initialize `BorderExc[][][][]` but not to allocate `BoundaryCommFields[]` and `BoundaryCommTypes[][][][]` because they are allocated by `init4p()`. In addition, `init_fields()` takes special care of `BorderExc`$[C-1][][][]$ to make the base type of the boundary communication `MPI_LONG_LONG_INT` rather than `MPI_DOUBLE`. Therefore, we may use `oh3_exchange_borders()` in `exchange_population()` for the boundary communication of per-grid

[59] The difference is essential for the reduction but maybe not for the broadcast because both of `MPI_LONG_LONG_INT` and `MPI_DOUBLE` are 64-bit wide. However daring to use `oh3_bcast_field()` is not very attractive because `oh1_broadcast()` is simple enough.

histogram knowing that `oh3_exchange_borders()` assumes the buffer pointers are `double` but this erroneous assumption is not harmful because `sizeof(double)` = `sizeof(dint)` = 8.

4.9.4 Variables for Particle Transfer Scheduling

The next variable group is for the particle transfer scheduling. Before showing them, we revisit the following variables whose usages are slightly different from those in lower level libraries.

`S_commlist`
`CommList`
`SecRList`
`RLIndex`

- As done in the level-1 library, we build the secondary mode particle transfer schedule in the array of `S_commlist` structure `CommList[]`. However, some of the structure elements have meanings different from those in level-1 as follows.
  - `rid` is the ID r of the node by which particles specified by the record should be accommodated.
  - `region` is the grid-position g of the last member of the grid-voxel set, the particles in which should be accommodated by r. That is, r will accommodate particles in the grid-voxels whose indices are in $(g', g]$ where g' is `region` of the previous record or -1 if the record in question is the first one.
  - `tag` is 0 for primary particles of r or NS for its secondary ones. In addition the element can be -1 if the record is in hot-spot sending block to indicate that the hot-spot record is for the particles to be accommodated by the local node.
  - `count` is 0 if the last grid-voxel at g is not a hot-spot. Otherwise it has the number of particles in a hot-spot at g to be accommodated by r, and the record is followed by records with the same g and non-zero `count` to specify the set of nodes which also accommodates particles in the hot-spot and the number of particles for them.
  - `sid` is meaningful only for a hot-spot record and has the zero-origin ordinal of the hot-spot in a subdomain, unless the record is the tail of the sequence of hot-spot receivers and has -1 to indicate it is the tail[60]. In hot-spot sending block introduced later, however, each record has the number of receivers to receive a hot-spot particles of a species accommodated by a node.

  As for the blocks in `CommList[]`, they are similar to those in level-1 but are differnt from them in various aspects as follows.

  primary receiving block is build by each node for particles in its primary subdomain to be accommodated by the node itself or its helpers. For a subdomain n, the records for each member of $F(n)$ appear at most twice, as the last host of a hot-spot and the first host of the subsequent grid-voxels. Therefore, the size of this block is at most $2|F(n)| \leq 2N$.

  primary sending block is exchanged by neighboring node (subdomain) pairs. A node receives the whole primary receiving block from each neighbor for particles sent from the family members rooted by the node to the family members rooted by the neighbor. Since we avoid receiving a primary receiving block twice or more from a neighbor and a node can appear at most two primary receiving blocks as a helpand and a helper, the size of this block is at most $2 \times 2 \times N = 4N$.

[60] The ordinal is meaningful only in the head record of the sequence. For non-hot-spot records, the ordinal is of course meaningless but has that of the next hot-spot if any.

The integer array `RLIndex`$[3^D + 1]$ has the `CommList`'s index of the first record of primary receiving block received from k-th neighbor unless the neighbor is the local node itself for which the index is 0 to mean the primary receiving block built by the local node itself. In addition, if a node appears twice or more as neighbors of the local node, all the elements of `RLIndex`[] for the node commonly have the index of the sole primary receiving block received from the node. Another remark is that `RLIndex`$[3^D]$ has the index of the record just following the primary sending block, or the combined size of primary receiving and primary sending blocks in other words.

secondary receiving block for a node is the copy of primary receiving block of its helpand which broadcasts the block to its helpers to show them particle accommodations for its primary subdomain and thus helper's secondary subdomain. Therefore, the size of this block is at most $2N$. The pointer `SecRList` points the head of this block as done in level-1 library.

secondary sending block for a node is the copy of primary sending block of its helpand which broadcasts the block to its helpers to show them particle transmissions to the subdomains neighboring to its primary subdomain and thus helper's secondary subdomain. Therefore, the size of this block is at most $4N$.

The integer array `SecRLIndex`$[3^D + 1]$ shown later is the copy of `RLIndex`[] of the helpand and thus has the offset from `SecRList` of each primary receiving block which the helpand receives from its neighbor.

alternative secondary receiving block for a node is the copy of primary receiving block of its helpand which broadcasts the block to its helpers which become its family members by rebalancing. A node must refer to both of secondary receiving blocks gotten from its old and new helpand because the former may have particle transmissions for its old secondary subdomain. The size of this block is at most $2N$. The pointer `AltSecRList` shown later points the head of this block.

hot-spot sending block for a node is for the records by which the node sends particles in the hot-spots in its primary or secondary subdomain or a subdomain neighboring them. Unlike other blocks, a record in this block is bulit for a species. Since a node can have all hot-spots in its primary and secondary subdomains including their sending planes and every node can host four hot-spots, two for its primary subdomain and other two for secondary one, this block can have $4NS$ records.

The total of the maximum size of each block is $(14 + 4S)N$ and can be greater than $2 \cdot 3^D(NS + 1) + N(S + 3)$ that level-1 requires if $D = 1$ and $S < 4$. Therefore `init1()` takes care of the possiblity and allocates `CommList`$[(14+4S)N]$ if the former is greater.

`NOfSend` `NOfRecv`

- Unlike non-position-aware case, the particle transfer schedule built in `CommList`[] is not complete because it lacks sender information. This is due to that the node responsible of a subdomain as its primary one does not know the individual particle population of each grid-voxel and each node having it. On the other hand, scanning `CommList`[] makes each node know which node should accommodate (a part of) particles in a grid-voxel the node has. Therefore, at first we build a per-receiver histogram of sending particles in each node and then exchange them among nodes in neighboring family members to build a per-sender histogram of receiving particles to have the complete sending/receiving schedule.

We use `NOfSend`[2][S][N] for the per-receiver sending histogram so that its element [p][s][m] has the number of particles of species s which the local node should send to the receiver node m as m's (not the local node's) primary ($p = 0$) or secondary ($p = 1$) particles. Each element is accumulated by `make_send_sched_body()` for particles in non-hot-spot grid-voxels and `scatter_hspot_recv_body()` for those in hot-spots. Then we perform a hand-made all-to-all communication among neighboring family members in `exchange_xfer_amount()` to exchange `NOfSend`[] to have the per-sender receiving histogram in `NOfRecv`[2][S][N] so that its element [p][s][m] has the number of particles of species s which the local node should receive from the sender node m as the local node's (not m's) primary ($p = 0$) or secondary ($p = 1$) particles.

In addition, `NOfSend`[p][s][m] then acts as the index of a portion of `SendBuf`[], $sbuf(p, s, m)$, to which a particle of species s to be sent to m as m's primary ($p = 0$) or secondary ($p = 1$) particle is moved. This role is similar to `SendBufDisps`[s][m] for $sbuf(s, m)$ but we need to the additional dimension for [p] because the lower level's per-subdomain configuration would shuffle particles of different destinations. The role change is done by `set_sendbuf_disps4p()`, and then `move_to_sendbuf_uw4p()`, `move_to_sendbuf_dw4p()` and `move_and_sort_secondary()` increments an element each time a particle is moved from `Particles`[] to `SendBuf`[] for sending.

Then particles are sent in `xfer_particles()` referring to `NOfSend`[][][] for the send count, each element referred to and thus possibly having non-zero is zero-cleared for the accumulation in the next call of `transbound4p()`. All the entries, in addtion, are also zero-cleared in `init4p()` at the very beginning and before the first call of `transbound4p()`.

`NOfRecv`[][][] also has another role in which its first half is used as an array of [N][S]. In the scattering communication for the particle populations in a hot-spot in the local node's primary subdomain, the node sets the element [k][s] to the number of particles the node itself or its helper should accommodate, where k is the ordinal of the accommodating node in the hot-spot member nodes. This map is locally manipulated by `scatter_hspot_send()`.

`Requests` `Statuses`

- The usage of `Requests`[] and `Statuses`[] to keep the requests/statuses of asynchronous MPI communications is not changed but its required size could be a little bit larger than that for particle transfer $4NS$ discussed in §4.4.2. That is, in the hot-spot gathering communication, a node can post `MPI_Irecv()` from 2^D neighbors the half of which can be identital each other. Since a neighbor m may have a large family of $F(m) = N$ members, the total number of posted `MPI_Irecv()` can be $(2^D/2)N = 2^{D-1}N$ and thus $4N$ in three-dimensional simulation with $D = 3$. At the same time, the node may also be involved in the hot-spot scattering communication with $3^D - 1$ neighbors of its primary and secondary subdomains, and its helpand from which it can receive two messages. Therefore, the maximum number of pending `MPI_Irecv()` is $4N + 2 \cdot (3^D - 1) + 2 = 4N + 2 \cdot 3^D$ which can be larger than $4NS$ with $S = 1$ and/or a small N. Therfore, `init2()` allocates the arrays of $4NS + 2 \cdot 3^D$ instead of $4NS$.

  `Requests`[] is referred to in;

  > `gather_hspot_recv()`, `gather_hspot_send()`, `scatter_hspot_send()`, `scatter_hspot_recv()`, `exchange_xfer_amount()` and `xfer_particles()`,

  while `Statuses`[] is referred to in;

scatter_hspot_send(), scatter_hspot_recv(), exchange_xfer_amount() and xfer_particles().

Now we show the variables and struct data types for the particle transfer scheduling.

gridOverflowLimit

- The integer variable gridOverflowLimit is set to $2P_{hot}$ by oh4p_max_local_particles() based on its argument hsthresh $= P_{hot}$. Since P_{hot} is the minimum cardinality of each subset split from the set of particles in a hot-spot grid-voxel, we cannot split the set if its cardinality is less than $2P_{hot}$. In other words, we may split the hot-spot set if adding it to the set for a member of primary family of the local node makes its accommodating particle population exceed what the balancing mechanism expect by $2P_{hot}$, and must do it to keep the excess over $P_{\max}$ is less than $4P_{hot}$, i.e., $2P_{hot}$ for each of primary and secondary particle sets. This variable is referred to solely in sched_recv() through the macro Sched_Recv_Check().

AltSecRList

- The S_commlist-type pointer AltSecRList is let point the head of alternative secondary receiving block in CommList[] by make_recv_list(). Then it is referred to by make_send_sched().

SecRLIndex

- The integer array SecRLIndex[$3^D + 1$] has the index of secondary receiving and secondary sending blocks in CommList[] in its element [k] for the k-th neighbor of the local node's helpand if $k < 3^D$, while the element [3^D] has the index of the block following secondary sending block, i.e., alternative secondary receiving or hot-spot sending block, or the combined size of secondary receiving and secondary sending blocks in other words. The array is obtained from the helpand by its broadcast of RLIndex[] in make_recv_list(), and then is referred to by make_send_sched().

S_recvsched_context

- The struct named S_recvsched_context is to keep the execution context of the function sched_recv() with the following elements, which are initialized by the caller make_recv_list() and then referred to and updated by sched_recv().
  - x, y and z are the local coordinate of the grid-voxel in per-grid histogram to be vistied, while g is its one-dimensional index.
  - hs is the number of hot-spots which have already visted.
  - nptotal is the number of particles which have already processed.
  - nplimit is the total number of particles which the nodes already visited are expected to accommodate by the balancing algorithm.
  - carryover is the number of particles in the visiting hot-spot which have not been assigned to nodes yet.
  - cptr is the pointer to a record in CommList[] to be built.

S_hotspot
HotSpotList
HotSpotTop

- The array HotSpotList[$2N + 2 \cdot 3^D + 1$] of the S_hotspot structure with the following elements keeps all hot-spots which the local node is involved in the last and/or the next simulation steps.
  - g is the one-dimensional index of the hot-spot.
  - n is the number of nodes which should accommodate the particles in the hot-spot.
  - lev is the zero-origin ordinal of the hot-spot in the subdomain to which it belongs.

– `self` is true iff the hot-spot is in the *interior* of the primary/secondary subdomain of the local node, i.e., not in *exterior* being its sending planes/edges/vertices, and the local node must accommodate some particles in the hot-spot.

– `comm` is the pointer to the head record for the hot-spot in primary receiving block at first then that in hot-spot sending block of `CommList[]`.

– `next` is the pointer to the succeeding `S_hotspot` element in the subdomain to which the hot-spot is belongs.

A node can be involved in all hot-spots in the simulated space domain, while the number of hot-spots is at most $2N$ because a node can be the first node of at most two hot-spots in its primary and secondary subdomains. Since the local node has one dummy element in `HotSpotList[]` for each neighbor of its primary and secondary subdomains including themselves and the newly assigned secondary subdomain in transitional state of helpand-helper reconfiguration, we need $2 \cdot 3^D + 1$ dummy elements. Therefore, the size of `HotSpotList[]` is $2N + 2 \cdot 3^D + 1$, with which `init4p()` allocates the array.

The `S_hotspot` type pointer `HotSpotTop` points the first unused element of `HotSpotList[]`. Therefore, `make_send_sched()` lets it be equal to `HotSpotList` to mean all elements in it are available. The function also increments the variable to acquire the dummy element for a subdomain. Then `make_send_sched_body()` increments it too when the function enqueues a hot-spot for a subdomain.

`S_hotspotbase` `HotSpot`

- The array `HotSpot`$[3][3^D]$ of the `S_hotspotbase` structure holds queues of `S_hotspot` elements in `HotSpotList[]` for each neighboring subdomain of the local node. The array element $[p][k]$ is for the hot-spots in the k-th neighbor of the local node's primary ($p = 0$) or secondary ($p = 1$) subdomain, in which the local node is involved. In addition the element $[2][\lfloor 3^D/2 \rfloor]$ is for the hot-spots in the newly assigned secondary subdomain in transitional state of helpand-helper reconfiguration. The structure has the following elements.

  – `head` is the pointer to the queue head on which the functions `gather_hspot_recv()`, `gather_hspot_send_body()`, `scatter_hspot_send()` and `scatter_hspot_recv_body()` operate.

  – `tail` is the pointer to the dummy element at queue tail. The enqueue operation by `make_send_sched_body()` takes place by making the dummy element active and acquiring a new dummy elemnent from `HotSpotTop`.

`HSRecv`

- The integer pointer array `HSRecv`$[3^D]$ is for a conceptually three-dimensional array of $[3^D][N][S]$ whose element $[k][m][s]$ is to receive the number of particles of species s in a hot-spot in the k-th boundary plane of the local node's primary subdomain accomodated by the node m being its helper, a neighbor or a helper of the neighbor. Note that $\lfloor 3^D/2 \rfloor$-th boundary plane is not actually boundary plane but the inner cuboid of the primary subdomain excluding real boundary planes.

  This array looks too large just for one hot-spot but is designed to cope with complicated situations that one single node involved in the hot-spot belonging to a subdomain in multiple aspects. That is, a node can be responsible of a neighbor subdomain as its primary one and another neighbor subdomain as its secondary one. More complicatedly, one single subdomain can acts as multiple neighbors of the local node's subdomain when we have periodic system-boundary condition and just one or two

nodes rank along an axis, in the case of which the hot-spot appears multiple times in different location of the per-grid histogram of the subdomain.

The body array of $[3^D][N][S]$ is allocated by `init4p()` which also initializes the pointer array so that its elements point appropriate elements. Then `gather_hspot_recv()` initiates the receiving of the hand-made gathering communication to the array, while `scatter_hspot_send()` examines received particle populations.

`HSSend`

- The integer array `HSSend`$[S]$ acts as the send buffer for the gathering communication for the particle population of a hot-spot. For each hot-spot at g, `gather_hspot_send_body()` copies $\mathcal{P}_L(p, s, g)$ = `NOfPGrid`$[p][s][g]$ into `HSSend`$[s]$ and sends the whole of `HSSend`[] to the node responsible of the subdomain including the hot-spot as its primary subdomain, if the local node accommodates primary ($p = 0$) and/or secondary ($p = 1$) particles in the hot-spot. The array is allocated by `init4p()`.

`HSRecvFromParent`

- The integer array `HSRecvFromParent`$[S]$ acts as the receive buffer for the scattering communication of the particle population in a hot-spot. That is, the local node having a hot-spot in its secondary subdomain receives the number of particels of species s which the node has to accommodate in the element of $[s]$. The receiving operation is initiated by `gather_hspot_send_body()` and the received populations are examined by `scatter_hspot_recv_body()`. The array is allocated by `init4p()`.

`HSReceiver`

- The integer array `HSReceiver`$[S]$ is locally used by `scatter_hspot_send()` to remember the ordinal of the hot-spot receiver to which a node sends its particle of species s in the element $[s]$. The array is allocated by `init4p()`.

`T_Hgramhalf`

- The `MPI_Datatype` variable `T_Hgramhalf` has the MPI data-type for a slice $[p][*][m]$ in `NOfSend`[][][] and `NOfRecv`[][][] to send/receive the particle populations the node m should accommodate as its primary ($p = 0$) or secondary ($p = 1$) particles. The value of this variable is created by `MPI_Type_vector()` called in `init4p()` so that the type has S elements with the stride of N, and is used in `exchange_xfer_amount()`.

```
EXTERN int gridOverflowLimit;
EXTERN struct S_commlist *AltSecRList;
EXTERN int SecRLIndex[OH_NEIGHBORS+1];

struct S_recvsched_context {
  int x, y, z, g, hs;
  dint nptotal, nplimit, carryover;
  struct S_commlist *cptr;
};
struct S_hotspot {
  int g, n, lev, self;
  struct S_commlist *comm;
  struct S_hotspot *next;
};
EXTERN struct S_hotspot *HotSpotList, *HotSpotTop;      /* [2*nn+2*3^D+1] */
struct S_hotspotbase {
  struct S_hotspot *head, *tail;
};
EXTERN struct S_hotspotbase HotSpot[3][OH_NEIGHBORS];

EXTERN int *HSRecv[OH_NEIGHBORS];                       /* [3^D][nn][ns] */
```

```
EXTERN int *HSSend, *HSRecvFromParent, *HSReceiver;     /* [ns] */
EXTERN MPI_Datatype T_Hgramhalf;
```

4.9.5 Variables for Neighboring Information

Next, we declare arrays to hold neighboring information.

FirstNeighbor

- When `make_recv_list()` receives primary receiving blocks from neighbors, we need to know not only a node appears twice or more in `SrcNeighbors[]` but also the ordinal of its first occurrence so that `RLIndex`$[k]$ for the second or following occurrence of k-th neighbor has `RLIndex`$[k']$ where $k' < k$ and `SrcNeighbors`$[k] = -($`SrcNeighbors`$[k'] + 1)$. The array `FirstNeighbor`$[3^D]$ is for this and its element $[k]$ has k if $m =$ `SrcNeighbors`$[k] \geq 0$ or $m = -N - 1$ to mean the first occurrence, or k' such that $m = -($`SrcNeighbors`$[k'] + 1)$. The array is let have these values by `init4p()`.

GridOffset

- The array `GridOffset`$[2][3^D]$ has the offset $\mathit{goff}(k)$ to translate a grid-position of the k-th neighbor m of the local node n's primary ($p = 0$) or secondary ($p = 1$) subdomain $n' \in \{n, \mathit{parent}(n)\}$ into the corresponding grid-position of n' in the element $[p][k]$. That is, when (x, y, z) in m's local coordinate corresponds to (x', y', z') of n', $\mathit{gidx}(x', y', z') = \mathit{gidx}(x, y, z) + \mathit{goff}(k)$. The d-th dimensional origin $x^0_d(m)$ of the subdomain m is at $\delta^l_d(m)$ and thus $x^0_d(m)$ is at $x^0_d(m, n') = \delta^l_d(m) - \delta^l_d(n')$ in the local coordinate of n'. Therefore, for the k-th neighbor m of n', $x^0_d(m, n')$ is calculated by;

$$x^0_d(m, n') = \delta^l_d(m) - \delta^l_d(n') = \begin{cases} \delta^l_d(m) - \delta^l_u(m) = \delta_d(m) & \nu_d = 0 \\ \delta^l_d(n') - \delta^l_d(n') = 0 & \nu_d = 1 \\ \delta^u_d(n') - \delta^l_d(n') = \delta_d(n') & \nu_d = 2 \end{cases}$$

  where $k = \sum_{d=0}^{D-1} \nu_d 3^d$, and thus $\mathit{goff}(k) = \mathit{gidx}(x^0_0(m, n'), \ldots)$. The values $[p][]$ are initialized/updated when `Neighbors`$[p][]$ is initialized/updated by the function `update_neighbors()`, called from `init4p()` for $[0][]$ and from `rebalance4p()` and `exchange_particles4p()` for $[1][]$. Besides the initializer/updater `update_neighbors()`, the array is reffered to through the macro `Local_Grid_Position()` invokded in the macro `Move_Or_Do()` and in the function `oh4p_remove_mapped_particle()`.

S_realneighbor
RealDstNeighbors
RealSrcNeighbors

- The arrays `RealDstNeighbors`$[2][2]$ and `RealSrcNeighbors`$[2][2]$ of `S_realneighbor` structure have the sets of nodes in the neighboring families of the local node. The structure element `nbor`$[N]$ is the array of a node set and `n` has its cardinality.

  The element `RealDstNeighbors`$[0][p]$ has the nodes responsible of the subdomain neighboring the local node's primary *and* secondary subdomains as their primary ($p = 0$) *or* secondary ($p = 1$) subdomains. This means that particles accommodated by the local node will be sent to them as their primary ($p = 0$) or secondary ($p = 1$) particles. On the other hand, the element `RealSrcNeighbors`$[0][p]$ has the nodes responsible of the subdomain neighboring the local node's primary ($p = 0$) *or* secondary ($p = 1$) subdomain as their primary *and* secondary subdomains. This means that particle that the local node will accommodate as its primary ($p = 0$) or secondary ($p = 1$) ones are sent from them.

The elements RealDstNeighbors[1][p] and RealSrcNeighbors[1][p] have same meanings as their [0][p] counterparts but they are for transitional state of helpand-helper reconfiguration. Therefore, RealDstNeighbors[1][p] should have the helpand ($p = 0$) or its *new* helpers ($p = 1$) of the neighbor subdomains of the local node's primary subdomain and its *old* secondary subdomain. Similarly but inversely, RealSrcNeighbors[1][p] should have the helpand and its *old* helpers of the neighbor subdomains of the local node's primary subdomain ($p = 0$) or its *new* secondary subdomain ($p = 1$).

The arrays are allocated by init4p(), are updated by update_real_neighbors() and its callee upd_real_nbr(), and are referred to by exchange_xfer_amount(), set_sendbuf_disps4p() and xfer_particles().

In addition we slightly modified the definitions of a few arrays declared in level-1 library as follows.

Neighbors

- We add the element array [2][] to Neighbors[3][N] so that it temporarily has the neighbors of the local node's helpand by build_new_comm(). The added element Neighbors[2][] is referred to by upd_real_nbr() to construct RealSrcNeighbors[1][1] and then copied into Neighbors[1][] by rebalance4p().

  Besides this extra role, Neighbors[0][] and Neighbors[1][] are used with the original meaning in make_send_sched(), update_neighbors() and gather_hspot_send().

TempArray

- In update_real_neighbors(), we have to keep track the occurrence of the nodes in the set RealDstNeighbors[k][p].nbor[] and RealSrcNeighbors[k][p].nbor[] for each particular k but for all $p \in [0, 1]$. Therefore, we need an array of [4][N] and thus let init4p() allocate $4N$ elements for TempArray[] for this purpose. The whole part of TempArray[$4N$] is referred to by update_real_neighbors() and its callee upd_real_nbr(), while init4p() uses the first N elements to build FirstNeighbor[].

- Besides two arrays above, we use the following neighbor arrays in the orignial meanings.
  - DstNeighbors[] in make_recv_list() and gather_hspot_recv().
  - SrcNeighbors[] in init4p() and make_recv_list().

```
EXTERN int FirstNeighbor[OH_NEIGHBORS], GridOffset[2][OH_NEIGHBORS];
struct S_realneighbor {
  int n, *nbor;
};
EXTERN struct S_realneighbor RealDstNeighbors[2][2], RealSrcNeighbors[2][2];
```

4.9.6 Variable for Boundary Condition

BoundaryCondition

The last variable is BoundaryCondition[D][2] being the substance of the oh4p_init()'s argument bcond to have the boundary condition of the lower ($\beta = 0$) or upper ($\beta = 1$) system boundary plane parpendicular to d-th dimensional axis in the element [d][β]. The array is initialized in init4p() and is referred to by the macro Map_Particle_To_Subdomain() used in oh4p_map_particle_to_subdomain().

```
EXTERN int BoundaryCondition[OH_DIMENSION][2];
```

4.9.7 Function Prototypes

The next and last block is to declare the prototypes of the API function pairs each of which consists of API for Fortran and C, as listed below.

- The function `oh4p_init[_]()` initializes data strucutures of the level-4p and lower level libraries.
- The function `oh4p_max_local_particles[_]()` defines P_{hot}, the minimum cardinality of a subset split from a hot-spot, and calculates P_{lim}, the size of the particle buffer `Particles[]`, based on P_{hot} and other parameters.
- The function `oh4p_per_grid_histogram[_]()` defines an array to be associated with per-grid histogram.
- The function `oh4p_transbound[_]()` at first performs what its level-1 counterpart `oh1_transbound[_]()` does to have the fundamental particle assignment for load balancing, and then modifies it to have position-aware particle distribution by the level-4p's own particle transfer.
- The function `oh4p_map_particle_to_neighbor[_]()` finds the subdomain in which a given particle resides, providing that the subdomain is a neighbor of the primary/secondary subdomain of the local node, and maintains per-subdomain and per-grid histograms of particle population.
- The function `oh4p_map_particle_to_subdomain[_]()` finds the subdomain in which a given particle resides, allowing that the subdomain is not necessary to be a neighbor of the primary/secondary subdomain of the local node, and maintains per-subdomain and per-grid histograms of particle population.
- The function `oh4p_inject_particle[_]()` injects a particle and place it at the bottom of `Particles[]` maintaining per-subdomain and per-grid histograms of particle population.
- The function `oh4p_remove_mapped_particle[_]()` removes a particle which has been mapped to a subdomain or been injected into a subdomain.
- The function `oh4p_remap_particle_to_neighbor[_]()` does what functions `oh4p_remove_mapped_particle()` and `oh4p_map_particle_to_neighbor()` do.
- The function `oh4p_remap_particle_to_subdomain[_]()` does what functions `oh4p_remove_mapped_particle()` and `oh4p_map_particle_to_subdomain()` do.

As done in §4.2.11, §4.4.5 and §4.6.6, prior to showing the function prototypes, we show the fourth part of the header files ohhelp_c.h for C-coded simulators and ohhelp_f.h for Fortran-coded ones, which define the aliases of level-4p API functions. In the `#else` part of `#if␣OH_LIB_LEVEL=3`, at first they `#define` the aliases of API functions if `OH_LIB_LEVEL_4P` is defined.

```
#else
#ifdef OH_LIB_LEVEL_4P
#define \
oh_init(A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,A15,A16,A17,A18,
A19,A20,A21,A22) \
oh4p_init(A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,A15,A16,A17,A18,
A19,A20,A21,A22)
```

```
#define oh_max_local_particles(A1,A2,A3,A4) \
        oh4p_max_local_particles(A1,A2,A3,A4)
#define oh_per_grid_histogram(A1) oh4p_per_grid_histogram(A1)
#define oh_transbound(A1,A2)      oh4p_transbound(A1,A2)
#define oh_map_particle_to_neighbor(A1,A2,A3) \
        oh4p_map_particle_to_neighbor(A1,A2,A3)
#define oh_map_particle_to_subdomain(A1,A2,A3) \
        oh4p_map_particle_to_subdomain(A1,A2,A3)
#define oh_inject_particle(A1,A2) oh4p_inject_particle(A1,A2)
#define oh_remove_mapped_particle(A1,A2,A3) \
        oh4p_remove_mapped_particle(A1,A2,A3)
#define oh_remap_particle_to_neighbor(A1,A2,A3) \
        oh4p_remap_particle_to_neighbor(A1,A2,A3)
#define oh_remap_particle_to_subdomain(A1,A2,A3) \
        oh4p_remap_particle_to_subdomain(A1,A2,A3)
```

Then ohhelp_c.h gives the prototypes of the functions above, which are also given in ohhelp4p.h[61], while their Fortran versions are given in oh_mod4p.F90 as shown in §3.7.

```
void oh4p_init(int **sdid, const int nspec, const int maxfrac, int **totalp,
               struct S_particle **pbuf, int **pbase, const int maxlocalp,
               void *mycomm, int **nbor, int *pcoord, int **sdoms, int *scoord,
               const int nbound, int *bcond, int **bounds, int *ftypes,
               int *cfields, int *ctypes, int **fsizes,
               const int stats, const int repiter, const int verbose);
int  oh4p_max_local_particles(const long long int npmax, const int maxfrac,
                              const int minmargin, const int hsthresh);
void oh4p_per_grid_histogram(int **pghgram);
int  oh4p_transbound(int currmode, int stats);
int  oh4p_map_particle_to_neighbor(struct S_particle *part, const int ps,
                                   const int s);
int  oh4p_map_particle_to_subdomain(struct S_particle *part, const int ps,
                                    const int s);
int  oh4p_inject_particle(const struct S_particle *part, const int ps);
void oh4p_remove_mapped_particle(struct S_particle *part, const int ps,
                                 const int s);
int  oh4p_remap_particle_to_neighbor(struct S_particle *part, const int ps,
                                     const int s);
int  oh4p_remap_particle_to_subdomain(struct S_particle *part, const int ps,
                                      const int s);
```

Then ohhelp4p.h continues prototype declaration for Fortran API functions.

```
void oh4p_init_(int *sdid, const int *nspec, const int *maxfrac, int *totalp,
                struct S_particle *pbuf, int *pbase, const int *maxlocalp,
                struct S_mycommf *mycomm, int *nbor, int *pcoord, int *sdoms,
                int *scoord, const int *nbound, int *bcond, int *bounds,
                int *ftypes, int *cfields, int *ctypes, int *fsizes,
                const int *stats, const int *repiter, const int *verbose);
int  oh4p_max_local_particles_(const dint *npmax, const int *maxfrac,
                               const int *minmargin, const int *hsthresh);
void oh4p_per_grid_histogram_(int *pghgram);
```

[61] Prototypes of `oh4p_max_local_particles()` in ohhelp_c.h and ohhelp4p.h are slightly different, i.e., the type of its first argument is `long long int` in the former, while in the latter is `dint`.

```
int  oh4p_transbound_(int *currmode, int *stats);
int  oh4p_map_particle_to_neighbor_(struct S_particle *part, const int *ps,
                                    const int *s);
int  oh4p_map_particle_to_subdomain_(struct S_particle *part, const int *ps,
                                     const int *s);
int  oh4p_inject_particle_(const struct S_particle *part, const int *ps);
void oh4p_remove_mapped_particle_(struct S_particle *part, const int *ps,
                                  const int *s);
int  oh4p_remap_particle_to_neighbor_(struct S_particle *part, const int *ps,
                                      const int *s);
int  oh4p_remap_particle_to_subdomain_(struct S_particle *part, const int *ps,
                                       const int *s);
```

4.10 C Source File ohhelp4p.c

4.10.1 Header File Inclusion

The first job done in ohhelp4p.c is the inclusion of the header files ohhelp1.h, ohhelp2.h, ohhelp3.h and ohhelp4p.h. Before the inclusion of ohhelp1.h, ohhelp2.h and ohhelp3.h, we `#define` the macro `EXTERN` as `extern` so as to make variables declared in the files external, but after that we make it `#undef`'iend and then `#define` it as empty so as to provide variables declared in ohhelp4p.h with their homes, as discussed in §4.2.3.

```
#define EXTERN extern
#include "ohhelp1.h"
#include "ohhelp2.h"
#include "ohhelp3.h"
#undef  EXTERN
#define EXTERN
#include "ohhelp4p.h"
```

4.10.2 Function Prototypes

The next and last job to do prior to macro and function definitions is to declare the prototypes of the following functions private for the level-4p library.

- The function `init4p()` is the body of `oh4p_init()`.
- The function `transbound4p()` is the body of `oh4p_transbound()`.
- The function `try_primary4p()` performs position-aware particle transfer in primary mode after calling its level-1 couterpart `try_primary1()` to check if we will be in primary mode in the next step.
- The function `try_stable4p()` performs position-aware particle transfer in secondary mode after calling its level-1 couterpart `try_stable1()` to check if we can keep the helpand-helper configuration.
- The function `rebalance4p()` performs position-aware particle transfer in secondary mode after calling its level-1 counterpart `rebalance1()` to establish a new helpand-helper configuration.
- The function `exchange_particles4p()` is the core of position-aware particle transfer in secondary mode.
- The function `exchange_population()` gathers particle population in each grid-voxel to build per-grid histogram.
- The function `add_population()` adds particle populatoin in each grid-voxel in receiving planes to that in boundary planes.
- The function `mpi_allreduce_wrapper()` is the wrapper function of `MPI_Allreduce()` to call it with the API of `MPI_Reduce()`.
- The function `reduce_population()` sums per-grid histograms of the family members.

- The function `make_recv_list()` scans per-grid histogram to build primary receiving block, and then exchanges the block between neighbors to have primary sending block and broadcast them for secondary receiving, secondary sending and alternative secondary receiving blocks for helpers.
- The function `sched_recv()` scans per-grid histogram to determine the set of grid-voxels to be hosted by a node and to find a hot-spot.
- The function `make_send_sched()` scans primary receiving, primary sending, secondary receiving, secondary sending and alternative secondary receiving blocks to determine the node to which the local node sends the particles in each grid-voxel and processes all hot-spots after the scan.
- The function `make_send_sched_body()` scans a primary receiving block created by the local node itself, or that given from a neighbor, the helpand or a neighbor of the helpand to determine the node which accommodates the particles in each grid-voxel and to find hot-spots in the block.
- The function `gather_hspot_recv()` initiates the gather reception to have all accommodation data of a hot-spot.
- The function `gather_hspot_send()` scans hot-spots having a specific ordinal in all neighboring subdomains to send their accommodaion data.
- The function `gather_hspot_send_body()` sends the accommodation data of a hot-spot and initates scatter receptions to have receivers of particles in it and, if the local node should host it, the amount of particles to accommodate.
- The function `scatter_hspot_send()` builds the accommodations of a hot-spot and sends its sending and receiving schedules to the nodes involoved.
- The function `scatter_hspot_recv()` scans hot-spots having a specific ordinal in all neighboring subdomains to examine their sending and receiving schedules.
- The function `scatter_hspot_recv_body()` examines the sending and receiving schedule of a hot-spot.
- The function `update_descriptors()` updates elements in `FieldDesc[][][]` and `BorderExc[][][][]` for the secondary subdomain newly assigned to the local node by rebalancing.
- The function `update_neighbors()` initializes/updates `AbsNeighbors[][]` and `GridOffset[][]` for the local node's primary or secondary subdomain.
- The function `set_grid_descriptor()` sets an element of `GridDesc[]` according to a subdomain.
- The function `adjust_field_descriptor()` adjusts `FieldDesc`$[F-1]$`.{bc,red}.size[]` for the broadcast and reduction of per-grid histogram.
- The function `update_real_neighbors()` updates `RealDstNeighbors[][]` and `RealSrcNeighbors[][]`.
- The function `upd_real_nbr()` updates an element array of `RealDstNeighbors[][]` or `RealSrcNeighbors[][]`.

- The function `exchange_xfer_amount()` performs a hand-made all-to-all communication to send `NOfSend[][][]` and to receive it to `NOfRecv[][][]`.
- The function `count_population()` accumulates the number of particles in each grid-voxel in primary and secondary subdomains to have the local per-grid histogram.
- The function `sort_particles()` performs bucket sorting on `Particles[]` to have sorted result in `SendBuf[]`.
- The function `move_and_sort_primary()` moves all particles in `Particles[]` to `SendBuf[]` sorting those staying in the local node, if we are in primary mode in next simulation step.
- The function `sort_received_particles()` moves all received particles in each $rbuf(p, s)$ to `SendBuf[]` sorting them.
- The function `move_to_sendbuf_sec4p()` is the level-4p counterpart of `move_to_sendbuf_secondary()` to move particles to be sent to `SendBuf[]` and pack those to stay in the local node in `Particles[]`.
- The function `move_to_sendbuf_uw4p()` is the level-4p counterpart of `move_to_sendbuf_uw()` to move particles to be sent to `SendBuf[]` and pack those to stay in the local node shifting upward in `Particles[]`.
- The function `move_to_sendbuf_dw4p()` is the level-4p counterpart of `move_to_sendbuf_dw()` to move particles to be sent to `SendBuf[]` and pack those to stay in the local node shifting downward in `Particles[]`.
- The function `move_and_sort_secondary()` moves all particles in `Particles[]` to `SendBuf[]` sorting those staying in the local node, if we are in secondary mode in next simulation step.
- The function `set_sendbuf_disps4p()` is the level-4p counterpart of `set_sendbuf_disps()` to updates entries of `NOfSend[][][]` so that each of its entry has the displacement of the head of $sbuf(p, s, m)$.
- The function `xfer_particles()` performs a hand-made all-to-all communication to exchange particles.

```
static void init4p(int **sdid, const int nspec, const int maxfrac,
                   int **totalp, struct S_particle **pbuf, int **pbase,
                   int maxlocalp, struct S_mycommc *mycommc,
                   struct S_mycommf *mycommf, int **nbor, int *pcoord,
                   int **sdoms, int *scoord, const int nbound, int *bcond,
                   int **bounds, int *ftypes, int *cfields, const int cfid,
                   int *ctypes, int **fsizes,
                   const int stats, const int repiter, const int verbose);
static int  transbound4p(int currmode, int stats, const int level);
static int  try_primary4p(const int currmode, const int level,
                          const int stats);
static int  try_stable4p(const int currmode, const int level, const int stats);
static void rebalance4p(const int currmode, const int level, const int stats);
static void exchange_particles4p(const int currmode, const int level,
                                 int reb, int oldp, const int newp,
```

```
                                const int stats);
static void exchange_population(const int currmode, const int nextmode);
static void add_population(dint *npd, const int xl, const int xu,
                           const int yl, const int yu, const int zl,
                           const int zu, const int src);
static int  mpi_allreduce_wrapper(void *sendbuf, void *recvbuf,
                                  const int count, MPI_Datatype datatype,
                                  MPI_Op op, const int root, MPI_Comm comm);
static void reduce_population(int (*mpired)(void*, void*, int,
                                            MPI_Datatype, MPI_Op, int,
                                            MPI_Comm));
static struct S_commlist* make_recv_list(const int currmode,
                                         const int level, const int reb,
                                         const int oldp, const int newp,
                                         const int stats);
static void sched_recv(const int currmode, const int reb, const int get,
                       const int stay, const int nid, const int tag,
                       struct S_recvsched_context *context);
static void make_send_sched(const int currmode, const int reb, const int pcode,
                            const int oldp, const int newp,
                            struct S_commlist *hslist, int *nacc, int *nsend);
static void make_send_sched_body(const int psor2, const int n, const int sdid,
                                 const int self, const int sender,
                                 struct S_commlist *rlist, int *maxhs,
                                 int *naccs, int *nsendptr);
static int  gather_hspot_recv(const int currmode, const int reb,
                              const struct S_hotspot *hs);
static void gather_hspot_send(const int hsidx, const int pcode, const int rreq,
                              const int nfrom, const int nto,
                              struct S_commlist **hslist, int *sreqptr);
static void gather_hspot_send_body(const int hsidx, const int psor2,
                                   const int n, int dst, const int sender,
                                   struct S_commlist **hslist,
                                   MPI_Request *reqs, int *sreqptr);
static void scatter_hspot_send(const int rreq, int *nacc,
                               struct S_commlist **hslist);
static int  scatter_hspot_recv(const int hsidx, const int pcode,
                               const int rreq, const int sreq, const int nfrom,
                               const int nto, int *nacc, int *nsend);
static void scatter_hspot_recv_body(const int hsidx, const int psor2,
                                    const int n, int *naccptr, int *nsendptr);
static void update_descriptors(const int oldp, const int newp);
static void update_neighbors(const int ps);
static void set_grid_descriptor(const int idx, const int nid);
static void adjust_field_descriptor(const int ps);
static void update_real_neighbors(const int mode, const int dosec,
                                  const int oldp, const int newp);
static void upd_real_nbr(const int root, const int psp, const int pss,
                         const int nbr, const int dosec, struct S_node *node,
                         struct S_realneighbor rnbrptr[2], int *occur[2]);
static void exchange_xfer_amount(const int trans, const int psnew);
static void count_population(const int nextmode, const int psnew,
                             const int stats);
static void sort_particles(dint ***npg, const int nextmode, const int psnew,
```

```
                            const int stats);
static void move_and_sort_primary(dint ***npg, const int psold,
                                  const int stats);
static void sort_received_particles(const int nextmode, const int psnew,
                                    const int stats);
static void move_to_sendbuf_sec4p(const int psold, const int trans,
                                  const int oldp, const int *nacc,
                                  const int nsend, const int stats);
static void move_to_sendbuf_uw4p(const int ps, const int mysd, const int cbase,
                                 const int nbase);
static void move_to_sendbuf_dw4p(const int ps, const int mysd, const int ctail,
                                 const int ntail);
static void move_and_sort_secondary(const int psold, const int psnew,
                                    const int trans, const int oldp,
                                    const int *nacc, const int stats);
static void set_sendbuf_disps4p(const int trans);
static void xfer_particles(const int trans, const int psnew,
                           struct S_particle *sbuf);
```

In addition, we use the following lower level library functions.

- The function `mem_alloc()` allocates a memory space by `malloc()`. It is called from `init4p()` directly or through the macro `Allocate_NOfPGrid()`.
- The function `mem_alloc_error()` aborts the simulation due to the memory shortage reporting its cause. It is called from `oh4p_max_local_particles()`.
- The function `errstop()` aborts the simulation due to an error detected by all processes reporting given error message. It is called from `init4p()`.
- The function `local_errstop()` aborts the simulation due to an error detected by the local process reporting given error message. It is called from `oh4p_inject_particle()` directly and from `oh4p_map_particle_to_neighbor()`, `oh4p_map_particle_to_subdomain()` and `oh4p_remove_mapped_particle()` through the macro `Check_Particle_Location()`.
- The function `transbound1()` is the body of `oh1_transbound()`. It is called from `transbound4p()`.
- The function `try_primary1()` is to examine whether particle distribution among subdomains is balanced well and thus we can perform the simulation in primary mode. It is called from `try_primary4p()`.
- The function `try_stable1()` is to examine whether particle distribution among nodes is balanced well and thus we can keep the current helpand-helper configuration. It is called from `try_stable4p()`.
- The function `rebalance1()` is to (re)build the helpand-helper configuration to cope with an unacceptable load imbalance. It is called from `rebalance4p()`.
- The function `build_new_comm()` is to build communicators for the helpand-helper families built by `rebalance1()`. It is called from `make_recv_list()`.
- The function `exchange_primary_particles()` is the core of the particle transfer in primary mode. It is called from `try_primary4p()`.

- The function `move_to_sendbuf_primary()` moves particles to be transferred from `Particles[]` to `SendBuf[]` and packs those remaining in `Particles[]` in primary mode. It is called from `try_primary4p()`.
- The function `set_sendbuf_disps()` calculates each entry of `SendBufDisps[][]`. It is called from `move_and_sort_primary()`.
- The function `exchange_particles()` is the core of the particle transfer in secondary mode. It is called from `exchange_particles4p()`.
- The function `init3()` is the body of `oh3_init()`. It is called from `init4p()`.
- The function `set_field_descriptors()` sets `FieldDesc`$[f]$`.{bc,red}.size`$[p]$ for all $f \in [0, F)$ and given $p \in \{0, 1\}$. It is called from `update_descriptors()`.
- The function `clear_border_exchange()` initializes `BorderExc`$[c][1][d][\beta]$`.{send, recv}` for all $c \in [0, C)$, $d \in [0, D)$ and $\beta \in \{0, 1\}$, or reinitializes them for the subdomain which the local node has had as the secondary one but discarded by rebalancing. It is called from `update_descriptors()`.
- The function `map_irregular_subdomain()` finds the subdomain of irregular process coordinate in which a particle resides. It is called from `oh4p_map_particle_to_subdomain()`.

4.10.3 Macros `If_Dim()`, `For_Z()`, `For_Y()`, `Do_Z()`, `Do_Y()`, `Coord_To_Index()` and `Index_To_Coord()`

Before starting to define functions, we define macros generally used in level-4p functions. The first group is for dimension dpendent operations.

`If_Dim()` The macro `If_Dim`(d, e_t, e_f) gives the expression e_t if $d < D$, while e_f is given otherwise. Though the macro is expanded to the trinary expression examining $d < D$, it is expected compilers transform the macro into e_t or e_f because d is a constant, `OH_DIM_Y` or `OH_DIM_Z`. The macro is used in `init4p()`, `gather_hspot_recv()`, `set_grid_descriptor()` and `oh4p_map_particle_to_subdomain()`.

```
#define If_Dim(D, ET, EF)  (OH_DIMENSION>D ? (ET) : (EF))
```

The other macros in this group have dimension dependent definitions and thus defined in `#if`/`#else`/`#endif` construct to examine `OH_DIMENSION`.

`For_Y()` `For_Z()` The macro `For_Y`(i, c, n) and `For_Z`(i, c, n) are expanded to the statement i if $D < 2$ and $D < 3$ respectively, while they are expanded to the for-loop header `for(`i`;`c`;`n`)` to construct a for-loop for the dimension 2 (y) or 3 (z). They are used in the macros `For_All_Grid()`, `For_All_Grid_Abs()` and `For_All_Grid_From()`.

`Do_Z()` `Do_Y()` The macro `Do_Y`(a) and `Do_Z`(a) are expanded to nothing if $D < 2$ and $D < 3$ respectively, while they are expanded to a otherwise. They are used in `oh4p_map_particle_to_neighbor()` and `oh4p_map_particle_to_subdomain()` for the shorthand of;

```
#if OH_DIM_d<OH_DIMENSION
a
#endif
```

where $d =$ `Y` or $d =$ `Z`.

`Coord_To_Index()` The macro `Coord_To_Index`$(x, y, z, w, d{\cdot}w)$ is expanded to $x + y \cdot w + z \cdot d \cdot w$ to give the one-dimensional index of the element $[x]$, $[y][x]$ or $[z][y][x]$ in a (conceptual) D-dimensional array of $[w]$, $[d][w]$ or $[h][d][w]$, i.e. $gidx(x, y, z)$. The array size paramenter d is assumed to be 0 if $D \leq 2$ and w to be 0 if $D = 1$. The macro is used in the macro `For_All_Grid()`, `For_All_Grid_Abs()` and `Allocate_NOfPGrid()`, and functions `sched_recv()`, `make_send_sched_body()`, `oh4p_map_particle_to_neighbor()` and `oh4p_map_particle_to_subdomain()`.

`Index_To_Coord()` The macro `Index_To_Coord`$(i, x, y, z, w, d{\cdot}w)$ assigns values to x, y, z such that $i = x + y \cdot w + z \cdot d \cdot w$ to translate the one-dimensional index i to its corresponding element $[x]$, $[y][x]$ or $[z][y][x]$ in a (conceptual) D-dimensional array of $[w]$, $[d][w]$ or $[h][d][w]$, i.e., $i = gidx(x, y, z)$. The variable y and/or z will have 0 if $D < 2$ or $D < 3$ respectively. The macro is used in `gather_hspot_recv()`.

```
#define For_Y(LINIT, LCONT, LNEXT) LINIT;
#define For_Z(LINIT, LCONT, LNEXT) LINIT;
#define Do_Y(ACT)
#define Do_Z(ACT)
#if OH_DIMENSION==1
#define Coord_To_Index(GX, GY, GZ, W, DW)  (GX)
#define Index_To_Coord(IDX, GX, GY, GZ, W, DW) {\
  GX = (IDX);  GY = 0;  GZ = 0;\
}
#else
#undef  For_Y
#define For_Y(LINIT, LCONT, LNEXT) for(LINIT; LCONT; LNEXT)
#undef  Do_Y
#define Do_Y(ACT) ACT
#if OH_DIMENSION==2
#define Coord_To_Index(GX, GY, GZ, W, DW)  ((GX) + (GY)*(W))
#define Index_To_Coord(IDX, GX, GY, GZ, W, DW)  {\
  const int idx=(IDX), w=(W);\
  GX = idx % w;  GY = idx / w;  GZ = 0;\
}
#else
#undef  For_Z
#define For_Z(LINIT, LCONT, LNEXT) for(LINIT; LCONT; LNEXT)
#undef  Do_Z
#define Do_Z(ACT) ACT
#define Coord_To_Index(GX, GY, GZ, W, DW)  ((GX) + (GY)*(W) + (GZ)*(DW))
#define Index_To_Coord(IDX, GX, GY, GZ, W, DW)  {\
  const int idx=(IDX), w=(W), dw=(DW);\
  GX = idx % w;  GY = (idx % dw) / w;  GZ = idx / dw;\
}
#endif
#endif
```

4.10.4 Macros `Decl_For_All_Grid()`, `For_All_Grid()`, `For_All_Grid_Abs()`, `The_Grid()`, `Grid_X()`, `Grid_Y()` and `Grid_Z()`

The next group of generally used macros are for traversing per-grid histogram.

`Decl_For_All_Grid()` The macro `Decl_For_All_Grid()` declares the following special local variables for `For_All_Grid()`, `For_All_Grid_Abs()` and `For_All_Grid_From()`, whose names have a common prefix `fag_`, for the traversal of grid-voxels (x, y, z) where $x \in [x_0, x_1)$, $y \in [y_0, y_1)$ and $z \in [z_0, z_1)$;

- `x1`, `y1` and `z1` have x_1, y_1 and z_1 respecively.
- `xidx`, `yidx` and `zidx` have x, y and z respectively.
- `gx`, `gy` and `gz` have $gidx(x, y, z)$, $gidx(x_0, y, z)$ and $gidx(x_0, y_0, z)$ respectively.
- `w` has `GridDesc[].w` $= \delta_x^{\max}+4e^g$ and `dw` has `GridDesc[].dw` $= (\delta_x^{\max}+4e^g)(\delta_y^{\max}+4e^g)$.

The macro is used in functions that use `For_All_Grid()`, `For_All_Grid_Abs()` or `For_All_Grid_From()`.

```
#define Decl_For_All_Grid()\
  int fag_x1, fag_y1, fag_z1;\
  int fag_xidx, fag_yidx, fag_zidx, fag_gx, fag_gy, fag_gz;\
  int fag_w, fag_dw;
```

`For_All_Grid()` `For_All_Grid_Abs()` The macro `For_All_Grid`$(p, x_0, y_0, z_0, x_1, y_1, z_1)$ constracts nested for-loops to traverse grid-voxels (x, y, z) in the per-grid histogram of local node n's primary $(p = 0)$ or secondary $(p = 1)$ subdomains, where $x \in [x_0, \delta_x(m)+x_1)$, $y \in [y_0, \delta_y(m)+y_1)$ and $z \in [z_0, \delta_z(m)+z_1)$, and $m = n$ or $m = parent(n)$. The macro `For_All_Grid_Abs`$(p, x_0, y_0, z_0, x_1, y_1, z_1)$ acts similarly but the ranges are $x \in [x_0, x_1)$, $y \in [y_0, y_1)$ and $z \in [z_0, z_1)$. Though their definitions are fairly complicated due to the special variable names with `fag_` and dimension dependent definitions, the expansion results are not so jumbled as shown in the `For_All_Grid()`'s example with $D = 3$ below.

$$\begin{aligned}
&\texttt{for(}z = z_0,\ x_1 = \delta_x(m) + x_1,\ \ y_1 = \delta_y(m) + y_1,\ z_1 = \delta_z(m) + z_1,\\
&\qquad\quad w = \delta_x^{\max}(m) + 4e^g,\ d = \delta_y^{\max}(m) + 4e^g,\ g_z = gidx(x_0, y_0, z_0)\texttt{;}\\
&\qquad z < z_1\texttt{;}\ \ z\texttt{++},\ g_z = g_z + d \cdot w\texttt{)}\\
&\quad\ \texttt{for(}y = y_0,\ g_y = g_z\texttt{;}\ \ y < y_1\texttt{;}\ \ y\texttt{++},\ g_y = g_y + w\texttt{)}\\
&\qquad\texttt{for(}x = x_0,\ g_x = g_y\texttt{;}\ \ x < x_1\texttt{;}\ \ x\texttt{++},\ g_x\texttt{++)}
\end{aligned}$$

Note that if $D < 3$, the outer for-loops are replaced with their initialization part by `For_Z()` and `For_Y()`, and thus $D = 1$ case of the example above is as follows effectively.

$$\begin{aligned}
&x_1 = \delta_x(m) + x_1\texttt{;}\\
&\texttt{for(}x = x_0,\ g_x = gidx(x_0)\texttt{;}\ \ x < x_1\texttt{;}\ \ x\texttt{++},\ g_x\texttt{++)}
\end{aligned}$$

The macro `For_All_Grid()` is used in `transbound4p()`, `exchange_population()`, `make_send_sched_body()`, `count_population()`, `sort_particles()`, `move_and_sort_primary()` and `move_and_sort_secondary()`, while `For_All_Grid_Abs()` is used solely in `add_population()`. Note that we have a relative `For_All_Grid_From()` defined afterward and solely used in `sched_recv()`.

```
#define For_All_Grid(PS, X0, Y0, Z0, X1, Y1, Z1)\
  For_Z((fag_zidx=(Z0), fag_x1=GridDesc[PS].x+(X1),\
         fag_y1=GridDesc[PS].y+(Y1), fag_z1=GridDesc[PS].z+(Z1),\
         fag_w=GridDesc[PS].w, fag_dw=GridDesc[PS].dw,\
         fag_gz=Coord_To_Index(X0,Y0,Z0,fag_w,fag_dw)),\
```

```
        (fag_zidx<fag_z1), (fag_zidx++,fag_gz+=fag_dw))\
    For_Y((fag_yidx=(Y0), fag_gy=fag_gz),\
          (fag_yidx<fag_y1), (fag_yidx++,fag_gy+=fag_w))\
      for (fag_xidx=(X0),fag_gx=fag_gy; fag_xidx<fag_x1; fag_xidx++,fag_gx++)
#define For_All_Grid_Abs(PS, X0, Y0, Z0, X1, Y1, Z1)\
  For_Z((fag_zidx=(Z0), fag_x1=(X1), fag_y1=(Y1), fag_z1=(Z1),\
         fag_w=GridDesc[PS].w, fag_dw=GridDesc[PS].dw,\
         fag_gz=Coord_To_Index(X0,Y0,Z0,fag_w,fag_dw)),\
        (fag_zidx<fag_z1), (fag_zidx++,fag_gz+=fag_dw))\
    For_Y((fag_yidx=(Y0), fag_gy=fag_gz),\
          (fag_yidx<fag_y1), (fag_yidx++,fag_gy+=fag_w))\
      for (fag_xidx=(X0),fag_gx=fag_gy; fag_xidx<fag_x1; fag_xidx++,fag_gx++)
```

The_Grid() Grid_X() Grid_Y() Grid_Z()

The macro `The_Grid()` is to use in the body part of `For_All_Grid()` and its relatives to give $gidx(x, y, z)$ stored in `fag_gx` but without referring to the special variable name. The other special variables `fag_xidx`, `fag_yidx` and `fag_zidx` for x, y and z can be also referred to by the macros `Grid_X()`, `Grid_Y()` and `Grid_Z()`. The macro `The_Grid()` is used in all functions using `For_All_Grid()`, `For_All_Grid_Abs()` or `For_All_Grid_From()`, while `Grid_X()`, `Grid_Y()` and `Grid_Z()` are used solely in the macro `Sched_Recv_Return()` in `sched_recv()` which uses its own variation `For_All_Grid_From()`.

```
#define The_Grid()  (fag_gx)
#define Grid_X()  (fag_xidx)
#define Grid_Y()  (fag_yidx)
#define Grid_Z()  (fag_zidx)
```

4.10.5 Constants URN_PRI, URN_SEC and URN_TRN

URN_PRI URN_SEC URN_TRN

The last group of macro definitions is for constants of the operation mode given to `update_real_neighbors()`. It updates only `RealDstNeighbors`[0][0] and `RealSrcNeighbors`[0][0] if its `mode` argument is `URN_PRI` = 0 to mean the execution mode changes to primary mode from other (including undefined) and it is called from `init4p()` or `try_primary4p()`. If it is called from `exchange_particles4p()` with `URN_SEC` = 1 to mean helper-helpand reconfiguration took place but its transitional state is not necessary to be aware of because of anywhere accommodatation, both of [0][0] and [0][1] are updated but not for [1][]. Otherwise, i.e., if it is called from `make_recv_list()` with `URN_TRN` = 2 to mean we have to be aware of transitional state of helper-helpand configuration, all array elements are updated. The contants are referred to by the functions stated above.

```
#define URN_PRI 0
#define URN_SEC 1
#define URN_TRN 2
```

4.10.6 oh4p_init() and init4p()

oh4p_init_() oh4p_init()

The API functions `oh4p_init_()` for Fortran and `oh4p_init()` for C receive a set of array/structure variables through which level-1 to level-4p library functions communicate with the simulator body, and a few integer parameters to specify the behavior of the library.

The functions have the same argument set as `oh3_init[_]()` but `nphgram` is excluded as discussed in §4.9.3. Therefore the argument addition and modification for the call of `init4p()` and setting `specBase` to 0 or 1 are almost same as those in `oh3_init[_]()` discussed in §4.7.3, but `init4p()` has neither of `nphgram`, `rcounts`, `scounts`, nor `skip2`.

```
void
oh4p_init_(int *sdid, const int *nspec, const int *maxfrac, int *totalp,
           struct S_particle *pbuf, int *pbase, const int *maxlocalp,
           struct S_mycommf *mycomm, int *nbor, int *pcoord, int *sdoms,
           int *scoord, const int *nbound, int *bcond, int *bounds,
           int *ftypes, int *cfields, int *ctypes, int *fsizes,
           const int *stats, const int *repiter, const int *verbose) {
  specBase = 1;
  init4p(&sdid, *nspec, *maxfrac, &totalp, &pbuf, &pbase, *maxlocalp, NULL,
         mycomm, &nbor, pcoord, &sdoms, scoord, *nbound, bcond, &bounds,
         ftypes, cfields, -1, ctypes, &fsizes,
         *stats, *repiter, *verbose);
}
void
oh4p_init(int **sdid, const int nspec, const int maxfrac, int **totalp,
          struct S_particle **pbuf, int **pbase, const int maxlocalp,
          void *mycomm, int **nbor, int *pcoord, int **sdoms, int *scoord,
          const int nbound, int *bcond, int **bounds, int *ftypes,
          int *cfields, int *ctypes, int **fsizes,
          const int stats, const int repiter, const int verbose) {
  specBase = 0;
  init4p(sdid, nspec, maxfrac, totalp, pbuf, pbase, maxlocalp,
         (struct S_mycommc*)mycomm, NULL, nbor, pcoord, sdoms, scoord, nbound,
         bcond, bounds, ftypes, cfields, 0, ctypes, fsizes,
         stats, repiter, verbose);
}
```

`Allocate_NOfPGrid()` Prior to give the definition of `init4p()`, we have to define the macro `Allocate_NOfPGrid` (π, h, t, σ, ν) used in the function to allocate and initialize a per-grid histogram, namely `NOfPGrid`$[2][S][\sigma]$ and `NOfPGridTotal`$[2][S][\sigma]$. The macro allocates the body of the per-grid histogram, pointed by π and having $2 \cdot S \cdot \sigma$ elements, by `mem_alloc()`. It also allocates a pointer array of $[2][S]$, whose element $[p][s]$ points the element $[p][s][gidx(2e^g, 2e^g, 2e^g)]$ corresponding to $(0, 0, 0)$ of the body array, by `mem_alloc()`. The arguments to call `mem_alloc()` have element type t and the name ν of the array to be included in the error message in case of memory shortage. Then it zero-clears all elements in the body array and makes the pointers $h[0]$ and $h[1]$ points $[0][]$ and $[1][]$ of the pointer array respectively. The argument t is `dint` to allocate `NOfPGrid`$[][][][]$ and `NOfPGridTotal`$[][][][]$ in `init4p()`, but `int` for the allocation of `NOfPGridOut`$[][][][]$ done in `oh4p_per_grid_histogram()`, the sole user other than `init4p()`.

```
#define Allocate_NOfPGrid(BODY, NPG, TYPE, SIZE, MSG) {\
  const int ns2 = nOfSpecies<<1;\
  const int gridsize = SIZE;\
  TYPE *npg = BODY;\
  TYPE **npgp = (TYPE**)mem_alloc(sizeof(TYPE*), ns2, MSG);\
  int s, g, exto=OH_PGRID_EXT<<1;\
```

```
  const int base = Coord_To_Index(exto, exto, exto,\
                                  GridDesc[0].w, GridDesc[0].dw);\
  if (!npg)\
    BODY = npg = (TYPE*)mem_alloc(sizeof(TYPE), ns2*gridsize, MSG) + base;\
  for (s=0; s<ns2; s++,npg+=gridsize) {\
    npgp[s] = npg;\
    for (g=0; g<gridsize; g++)  npg[g-base] = 0;\
  }\
  NPG[0] = npgp;  NPG[1] = npgp + nOfSpecies;\
}
```

`nOfLocalPLimitShadow` Yet another declaration prior to `init4p()` is for the variable `nOfLocalPLimitShadow`, which keeps the return value of `oh4p_max_local_particles()` or has -1 if it has not been called prior to `oh4p_init[_]()`. Then `init4p()` examines this variable to confirm that `oh4p_max_local_particles()` has been called and thus `gridOverflowLimit` $= 2P_{hot}$ has been defined, and that `init4p()`'s argument `maxlocalp` $= P_{lim}$ is not less than the return value kept in the variable. This variable is private to ohhelp4p.c unlike other global variables, because it is just used for the communication between `init4p()` and `oh4p_max_local_particles()` and it must have the initial value -1 prior to the execution.

`init4p()` The function `init4p()`, called from `oh4p_init[_]()` implements the initialization for those API functions. The arguments of the function are almost same as `oh4p_init()` but its `mycomm` is split into two arguments `mycommc` and `mycommf` and there is an additional argument `cfid` as discussed in §4.7.3.

```
static int nOfLocalPLimitShadow = -1;
static void
init4p(int **sdid, const int nspec, const int maxfrac, int **totalp,
       struct S_particle **pbuf, int **pbase, int maxlocalp,
       struct S_mycommc *mycommc, struct S_mycommf *mycommf,
       int **nbor, int *pcoord, int **sdoms, int *scoord,
       const int nbound, int *bcond, int **bounds, int *ftypes, int *cfields,
       const int cfid, int *ctypes, int **fsizes,
       const int stats, const int repiter, const int verbose) {
  int nn, me, nnns, nnns2, n;
  int (*ft)[OH_FTYPE_N] = (int(*)[OH_FTYPE_N])ftypes;
  int *cf = cfields;
  int (*ct)[2][OH_CTYPE_N] = (int(*)[2][OH_CTYPE_N])ctypes;
  int nf, ne, c, b, size, ps, tr;
  int *nphgram = NULL;
  int *hsr, *rnbr;
  dint *npgdummy = NULL,  *npgtdummy = NULL;
  int loggrid;
  dint idmax;
```

We need a few operations prior to call `init3()` for the initialization of lower level data structures, because we modify some of them for level-4p extension. At first, we get N by `MPI_Comm_size()`, and then allocate `TempArray`$[4][N]$ by `mem_alloc()`, whose size is four times as large as that required in lower level libraries as discussed in §4.9.5. We also allocate `Particles`$[P_{lim}]$ and `SendBuf`$[P_{lim}]$ as a contiguous array of $[2P_{lim}]$ if `pbuf` points `NULL`. Otherwise, what `pbuf` points is set to the pointer `Particles` and `SendBuf` is let point the head of the second half of `pbuf`.

```
MPI_Comm_size(MCW, &nn);  nnns = nn * nspec;  nnns2 = nnns << 1;
TempArray = (int*)mem_alloc(sizeof(int), nn<<2, "TempArray");
if (*pbuf)
  Particles = *pbuf;
else
  Particles = *pbuf =
    (struct S_particle*)mem_alloc(sizeof(struct S_particle),
                                  maxlocalp<<1, "Particles");
SendBuf = Particles + maxlocalp;
```

Next, we *intercept* arguments `ftypes[][]`, `cfields[]` and `ctypes[][][][]` to add one element to each of them for per-grid histogram. At first, we scan `ftypes[][]` and `cfields[]` to find their terminators and thus their numbers of elements which are $F-1$ and $C-1$ respectively. Then we allocate `FieldTypes`$[F+1][]$, `BoundaryCommFields`$[C+1]$ and `BoundaryCommTypes`$[C][][][]$ by `mem_alloc()`, and then copy the elements in the argument arrays into them by `memcpy()` for the first and last while by a for-loop for `BoundaryCommFields[]` to adjust the indices of `FieldTypes[][]`.

Then we add the last elements of them as follows and as discussed in §4.9.3.

$$\begin{aligned}
\texttt{FieldTypes}[F-1][] &= \{1,0,0,0,0,-e^g,e^g\} \\
\texttt{BoundaryCommFields}[C-1] &= F-1 \\
\texttt{BoundaryCommTypes}[C-1][b][][] &= \begin{cases} \{\{-e^g,e^g,e^g\},\{0,-2e^g,e^g\}\} & b=0 \\ \{\{0,0,0\},\{0,0,0\}\} & b>0 \end{cases}
\end{aligned}$$

We also add the terminator `FieldTypes`$[F][0]=-1$ and `BoundaryCommFields`$[C]=-1$.

```
for (nf=0; ft[nf][OH_FTYPE_ES]>0; nf++);
for (ne=0; cf[ne]+cfid>=0; ne++);
FieldTypes = (int(*)[OH_FTYPE_N])
             mem_alloc(sizeof(int), (nf+2)*OH_FTYPE_N, "FieldTypes");
BoundaryCommFields = cf =
  (int(*))mem_alloc(sizeof(int), ne+2, "BoundaryCommFields");
BoundaryCommTypes = (int(*)[2][OH_CTYPE_N])
                    mem_alloc(sizeof(int), (ne+1)*nbound*2*OH_CTYPE_N,
                              "BoundaryCommTypes");
memcpy(FieldTypes, ft, sizeof(int)*nf*OH_FTYPE_N);
for (c=0; c<ne; c++)  cf[c] = cfields[c] + cfid;
memcpy(BoundaryCommTypes, ct, sizeof(int)*ne*nbound*2*OH_CTYPE_N);
ft = FieldTypes;  ct = BoundaryCommTypes + ne * nbound;
ft[nf][OH_FTYPE_ES] = 1;
ft[nf][OH_FTYPE_LO] = 0;  ft[nf][OH_FTYPE_UP] = 0;
ft[nf][OH_FTYPE_BL] = 0;  ft[nf][OH_FTYPE_BU] = 0;
ft[nf][OH_FTYPE_RL] = -OH_PGRID_EXT;  ft[nf][OH_FTYPE_RU] = OH_PGRID_EXT;
ft[nf+1][OH_FTYPE_ES] = -1;
cf[ne] = nf;  cf[ne+1] = -1;
ct[0][OH_LOWER][OH_CTYPE_FROM] = -OH_PGRID_EXT;
ct[0][OH_LOWER][OH_CTYPE_TO]   = OH_PGRID_EXT;
ct[0][OH_LOWER][OH_CTYPE_SIZE] = OH_PGRID_EXT;
ct[0][OH_UPPER][OH_CTYPE_FROM] = 0;
ct[0][OH_UPPER][OH_CTYPE_TO]   = -(OH_PGRID_EXT<<1);
ct[0][OH_UPPER][OH_CTYPE_SIZE] = OH_PGRID_EXT;
```

```
for (b=1; b<nbound; b++)
  ct[b][OH_LOWER][OH_CTYPE_FROM] =
    ct[b][OH_LOWER][OH_CTYPE_TO]   =
    ct[b][OH_LOWER][OH_CTYPE_SIZE] =
    ct[b][OH_UPPER][OH_CTYPE_FROM] =
    ct[b][OH_UPPER][OH_CTYPE_TO]   =
    ct[b][OH_UPPER][OH_CTYPE_SIZE] = 0;
```

Now we call `init3()` passing almost all arguments of `init4p()` but with the following exceptions.

- `nphgram` is the pointer to `init4p()`'s local variable of the same name which has `NULL` to let `init3()` allocate `NOfPLocal`[][][], because `init4p()` does not have the argument.
- `rcounts` and `scounts` are `NULL` because they are unnecessary.
- `ftypes`, `cfields` and `ctypes` are `FieldTypes`[][], `BoundaryCommFields`[] and `BoundaryCommTypes`[][][][] respectively, and the arrays themselves are neither allocated nor initialized by `init3()`.
- `skip2` is 0 because we need level-2 initialization.

Note that `cfid` is passed unmodified because we need the original value for initialization of data structures other than `BoundaryCommFields`[]. As for its use to scan the terminator of `BoundaryCommFields`[], `init_fields()` ignores it when `OH_POS_AWARE` is defined.

```
init3(sdid, nspec, maxfrac, &nphgram, totalp, NULL, NULL, pbuf, pbase,
      maxlocalp, mycommc, mycommf, nbor, pcoord, sdoms, scoord, nbound,
      bcond, bounds, (int*)ft, cf, cfid, (int*)BoundaryCommTypes, fsizes,
      stats, repiter, verbose, 0);
```

Next, we confirm that `nOfLocalPLimitShadow` is non-negative and not greater than `maxlocalp` $= P_{lim}$, or in other words `oh4p_max_local_particles()` has been called and its return value is passed (possibly after incremented) to `maxlocalp`. If not, we stop the execution by `errstop()` with an appropriate error message.

```
if (nOfLocalPLimitShadow<0)
  errstop("oh4p_max_local_particles() has to be called before oh4p_init()");
else if (maxlocalp<nOfLocalPLimitShadow)
  errstop("argument maxlocalp %d given to oh4p_init() is less than that "
          "calculated by oh4p_max_local_particles() %d",
          maxlocalp, nOfLocalPLimitShadow);
```

Next we allocate and initialize per-grid histograms and related variables. First, we initialize `PbufIndex` to be `NULL` so that the macro `Check_Particle_Location()` will not refer to it until the first call of `transbound4p()` by which the array is allocated and is given meaningful values. Then, we allocate `NOfPGrid`[2][S][G] and `NOfPGridTotal`[2][S][G] by `Allocate_NOfPGrid()` after setting `GridDesc`[0] for the local node's primary subdomain by `set_grid_descriptor()`. Note that the first argument of `Allocate_NOfPGrid()` is a dummy pointer variable having `NULL` to let the macro allocate the body arrays.

Then we check $gidx(\delta_x^{\max}-1, \delta_y^{\max}-1, \delta_z^{\max}-1)$ being the largest possible one-dimensional index of subdomains is small enough to represent it by `int` when combined with the largest possible subdomain code. That is, we calculate;

$$\Gamma = \lfloor \log_2 gidx(\delta_x^{\max}-1, \delta_y^{\max}-1, \delta_z^{\max}-1) \rfloor + 1$$

and examines if $(2(N+3^D)-1)\cdot 2^\Gamma$ is not greater than `INT_MAX` or `OH_nid_t` is `dint`. If this examination fails to mean `OH_nid_t` is `int` but not large enough to represent the combination of the largest possible grid-position index and subdomain code, we stop the execution by `errstop()` with an appropriate message suggesting to define `OH_BIG_SPACE` . Otherwise, Γ and $2^\Gamma - 1$ are set into `logGrid` and `gridMask` respectively.

Finally, we call `adjust_field_descriptor()` to modify `FieldDesc`$[F-1].\{$`bc`, `red`$\}$.`size`$[0]$ so that collective communications of `NOfPGrid`$[p][][]$ or `NOfPGridTotal`$[p][][]$ are performed on the whole of $[S][G]$ rather than $[s][G]$ with a specific s.

```
  me = myRank;
  PbufIndex = NULL;
  set_grid_descriptor(0, me);
  size = GridDesc[0].dw * GridDesc[0].h;
  Allocate_NOfPGrid(npgdummy, NOfPGrid, dint, size, "NOfPGrid");
  Allocate_NOfPGrid(npgtdummy, NOfPGridTotal, dint, size, "NOfPGridTotal");

  size = Coord_To_Index(Grid[OH_DIM_X].size-1,
                        If_Dim(OH_DIM_Y, Grid[OH_DIM_Y].size-1, 0),
                        If_Dim(OH_DIM_Z, Grid[OH_DIM_Z].size-1, 0),
                        GridDesc[0].w, GridDesc[0].dw);
  for (loggrid=0; size; loggrid++,size>>=1);
  idmax = (dint)(((nn+OH_NEIGHBORS)<<1)-1)<<loggrid;
  if (idmax>INT_MAX && sizeof(OH_nid_t)==sizeof(int)) {
#if OH_DIMENSION==1
    errstop("local grid size (%d+%d) times number of nodes %d "
            "is too large for OH_nid_t=int and thus OH_BIG_SPACE should be "
            "defined.",
            GridDesc[0].w-(OH_PGRID_EXT<<2), OH_PGRID_EXT<<2, nn);
#elif OH_DIMENSION==2
    errstop("local grid size (%d+%d)*(%d+%d) times number of nodes %d "
            "is too large for OH_nid_t=int and thus OH_BIG_SPACE should be "
            "defined.",
            GridDesc[0].w-(OH_PGRID_EXT<<2), OH_PGRID_EXT<<2,
            GridDesc[0].d-(OH_PGRID_EXT<<2), OH_PGRID_EXT<<2, nn);
#else
    errstop("local grid size (%d+%d)*(%d+%d)*(%d+%d) times number of nodes %d "
            "is too large for OH_nid_t=int and thus OH_BIG_SPACE should be "
            "defined.",
            GridDesc[0].w-(OH_PGRID_EXT<<2), OH_PGRID_EXT<<2,
            GridDesc[0].d-(OH_PGRID_EXT<<2), OH_PGRID_EXT<<2,
            GridDesc[0].h-(OH_PGRID_EXT<<2), OH_PGRID_EXT<<2, nn);
#endif
  }
  logGrid = loggrid;  gridMask = (1 << loggrid) - 1;
  adjust_field_descriptor(0);
```

The next targets of allocation and initialization are data structures for particle transfer scheduling. We allocate `HotSpotList`$[2N+2\cdot 3^D+1]$, and the body of `HSRecv`$[3^D][N][S]$,

HSSend[S], HSRecvFromParent[S] and HSReceiver[S] by mem_alloc(). For HSRecv[][][], we initialize the pointer array of [3^D] so that each [k] of them points the element [k][0][0] in the body to have two-dimensional array of [3^D][NS] in reality. Then we create MPI datatype T_Hgramhalf by MPI_Type_vector() and MPI_Type_commit() for a slice [p][$*$][m] in NOfSend[][][] and NOfRecv[][][] as a vector having S elements of MPI_INT with a stride N. Finally, all elements in NOfSend[][][] are zero-cleared as the base of accumulation of sending particle counts in the first call of oh4p_transbound().

```
    HotSpotList = (struct S_hotspot*)mem_alloc(sizeof(struct S_hotspot),
                                               2*nn+2*OH_NEIGHBORS+1,
                                               "HotSpotList");
    hsr = (int*)mem_alloc(sizeof(int), OH_NEIGHBORS*nn*nspec, "HSRecv");
    for (n=0; n<OH_NEIGHBORS; n++,hsr+=nnns)  HSRecv[n] = hsr;
    HSSend = (int*)mem_alloc(sizeof(int), nspec*3, "HSSend");
    HSRecvFromParent = HSSend + nspec;  HSReceiver = HSRecvFromParent + nspec;
    MPI_Type_vector(nspec, 1, nn, MPI_INT, &T_Hgramhalf);
    MPI_Type_commit(&T_Hgramhalf);
    for (n=0; n<nnns2; n++)  NOfSend[n] = 0;
```

The next allocation and initialization are for data structures of neighboring information. First we initiaize FirstNeighbor[k] to k itself if m = SrcNeighbors[k] ≥ 0[62], otherwise the index k' such that SrcNeighbors[k'] $= -(m+1)$ which is kept in TempArray[$-(m+1)$]. Then we call update_neighbors() to initialize AbsNeighbors[0][] and GridOffset[0][] based on the values in Neighbors[0][].

Then we allocate RealDstNeighbors[2][2].nbor[N] and RealSrcNeighbors[2][2].nbor[N] and call update_real_neighbors() with the code URN_PRI to initialize their elements in [0][0] so that they have subdomain identifiers neighboring to the local node's primary subdomain[63].

```
    for (n=0; n<OH_NEIGHBORS; n++) {
      const int snbr = SrcNeighbors[n];
      if (snbr>=0)        FirstNeighbor[n] = TempArray[snbr] = n;
      else if (snbr<-nn)  FirstNeighbor[n] = n;
      else                FirstNeighbor[n] = TempArray[-(snbr+1)];
    }
    update_neighbors(0);
    rnbr = (int*)mem_alloc(sizeof(int), nn*2*2*2, "RealNeighbors");
    for (tr=0; tr<2; tr++)  for (ps=0; ps<2; ps++,rnbr+=nn) {
      RealDstNeighbors[tr][ps].n = RealSrcNeighbors[tr][ps].n = 0;
      RealDstNeighbors[tr][ps].nbor = rnbr;
      RealSrcNeighbors[tr][ps].nbor = rnbr + nn*2*2;
    }
    update_real_neighbors(URN_PRI, 0, -1, -1);
```

Finally, we copy the contents of bcond[D][2] to its substance BoundaryCondition[][] by memcpy(), if SubDomainDesc is NULL to mean regular process coordinate.

```
    if (!SubDomainDesc)
      memcpy(BoundaryCondition, bcond, sizeof(int)*OH_DIMENSION*2);
  }
```

[62] Or if $m = -N - 1 < -N$ for neighbors not existing but the value k is never used in this case.

[63] The second, third and fourth argument of update_real_neighbors(), dosec, oldp and newp, are meaningless in this call.

4.10.7 oh4p_max_local_particles()

oh4p_max_local_particles_()
oh4p_max_local_particles()

The API functions oh4p_max_local_particles_() for Fortran and oh4p_max_local_particles() for C calculate P_{lim} being the maximum number of particles a local node can accommodate and return it to the simulator body calling them. The function takes the arguments for its level-2 counterpart oh2_max_local_particles() and call it to the baseline of P_{lim}, and then add $4P_{hot}$ to P_{lim} where P_{hot} is given through its own last argument hsthresh to allow a node have $2P_{hot}$ extra number of particles for each of primary and secondary particle set above those expected by the load balancing algorithm. This $2P_{hot}$ allowance means the last grid-voxel allocated to a node could have up to $2P_{hot} - 1$ particles because the grid-voxel cannot be split as a hot-spot. Therefore, the function sets $2P_{hot}$ into gridOverflowLimit. The function also examines P_{lim} is less than the maximum positive int-type number INT_MAX $= 2^{31} - 1$, and if it finds excess it stops the execution by mem_alloc_error(). Finally, P_{lim} is stored into nOfLocalPLimitShadow to indicate the function is called and for the consistency check in init4p() against its maxlocalp argument.

```
int
oh4p_max_local_particles_(const dint *npmax, const int *maxfrac,
                          const int *minmargin, const int *hsthresh) {
  return(oh4p_max_local_particles(*npmax, *maxfrac, *minmargin, *hsthresh));
}
int
oh4p_max_local_particles(const dint npmax, const int maxfrac,
                         const int minmargin, const int hsthresh) {
  const dint npl = (dint)oh2_max_local_particles(npmax, maxfrac, minmargin) +
                   ((gridOverflowLimit = hsthresh<<1)<<1);
  const int nplint = npl;

  if (npl>INT_MAX) mem_alloc_error("Particles", 0);
  return((nOfLocalPLimitShadow=nplint));
}
```

4.10.8 oh4p_per_grid_histogram()

oh4p_per_grid_histogram_()
oh4p_per_grid_histogram()

The API functions oh4p_per_grid_histogram_() for for Fortran and oh4p_per_grid_histogram() for C associate the per-grid histogram NOfPGridOut[][][][] to that in the simulator body. The Fortran coded simulator must allocate a $(D+2)$-dimensional array whose leading D-dimensional size is specified in fsizes$[F-1][][]$ given through the argument of oh4p_init_(), and give the origin element of the array $(0, \ldots, 0, 1, 1)$ through the function's sole argument pghgram. On the other hand, C coded simulator may let the function allocate the array by giving a double pointer to NULL to pghgram, or allocate the array by itself and give the double pointer to the array's origin element to pghgram.

The function invokes the macro Allocate_NOfPGrid() to allocate NOfPGridOut$[2][S][G]$ where;

$$G = \texttt{GridDesc}[0].\texttt{dw} \times \texttt{GridDesc}[0].\texttt{h} = ((\delta_x^{\max} + 4e^g)(\delta_y^{\max} + 4e^g)) \times (\delta_z^{\max} + 4e^g)$$

and *returns* the pointer to the conceptual element NOfPGridOut$[0][0][0]\cdots[0]$ through *pghgram if it is NULL, to allocate the pointer array for it for the use in library functions, and to zero-clear its body.

```
void
oh4p_per_grid_histogram_(int *pghgram) {
  oh4p_per_grid_histogram(&pghgram);
}
void
oh4p_per_grid_histogram(int **pghgram) {
  Allocate_NOfPGrid(*pghgram, NOfPGridOut, int, GridDesc[0].dw*GridDesc[0].h,
                    "NOfPGridOut");
}
```

4.10.9 `oh4p_transbound()` and `transbound4p()`

`oh4p_transbound_()` `oh4p_transbound()`

The API function `oh4p_transbound_()` for Fortran and `oh4p_transbound()` for C provide the simulator body calling them with the load-balanced particle transfer mechanism of level-4p and lower level libraries. The meanings of their two arguments, `currmode` and `stats`, and return value in $\{-1, 0, 1\}$ are perfectly equivalent to those of the level-1 to level-3 counterparts `oh1_transbound[_]()`, `oh2_transbound[_]()` and `oh3_transbound[_]()`. Also similarly to the counterparts, their bodies only have a simple call of `transbound4p()` but the third argument `level` is 4 to indicate the function is called from level-4p API functions.

```
int
oh4p_transbound_(int *currmode, int *stats) {
  return(transbound4p(*currmode, *stats, 4));
}
int
oh4p_transbound(int currmode, int stats) {
  return(transbound4p(currmode, stats, 4));
}
```

`transbound4p()`

The function `transbound4p()`, called from `oh4p_transbound[_]()`, has a code structure similar to its level-2 counterpart `transbound2()` especially in its first half. That is, at first it calls its level-1 counterpart `transbound1()` to calculate `NOfPrimaries[][]`, `TotalPGlobal[]`, `nOfParticles` and `nOfLocalPMax` from `NOfPLocal[][][]` of the local node and other nodes, and to have `currmode` which indicates not only the current execution mode but also the accommodation mode, i.e., normal or anywhere. The function also allocates and calculates `TotalP[][]` from `NOfPLocal[][][]` at the first call of it (and thus of `transbound4p()`) and let `primaryParts` and `totalParts` have the number of particles the local node accommodates, i.e., the sum of `TotalP[][]`.

Then it calls functions for the heart of balancing examination also similarly to `transbound2()` but the functions are level-4p's own `try_primary4p()`, `try_stable4p()` and `rebalance4p()`.

```
static int
transbound4p(int currmode, int stats, const int level) {
  int ret=MODE_NORM_SEC;
  const int nn=nOfNodes, ns=nOfSpecies, nnns2=2*nn*ns;
  struct S_particle *tmp;
  int i, ps, s, tp;
  Decl_For_All_Grid();
```

```
stats = stats && statsMode;
currmode = transbound1(currmode, stats, level);

if (try_primary4p(currmode, level, stats))  ret = MODE_NORM_PRI;
else if (!Mode_PS(currmode) || !try_stable4p(currmode, level, stats)) {
  rebalance4p(currmode, level, stats);  ret = MODE_REB_SEC;
}
```

After that, we allocate `PbufIndex`[2][S] together with an additional element [2][0] so that the array has $2S+1$ elements, if it is `NULL` and thus this function has not been called. Then similarly to `transbound2()` again, we clear `NOfPLocal`[][][]; copy `TotalPNext`[][] to its substance `TotalP`[][]; set `totalParts` and its shadow pointed by `totalLocalParticles` to the sum of `TotalP`[p][s] for all $p \in [0,1]$ and $s \in [0,S)$ to memorize the total number of particles the local node accommodates at the beginning of the next simulation step, i.e., before any injections and removals; and clear `InjectedParticles`[0][][] $= q^{\mathrm{inj}}(n)$[][] and `nOfInjections` $= Q_n^{\mathrm{inj}}$ to indicate we have no injected particles at the beginning of the next simulation step. A difference is in the loop to copy `TotalPNext`[][] into `TotalP`[][] in which we let `PbufIndex`[p][s] have the head index of $pbuf(p,s)$. In addition, `PbufIndex`[2][0] is let have `totalParts` as the head index of the region following $pbuf(1,S-1)$.

```
if (!PbufIndex)
  PbufIndex = (int*)mem_alloc(sizeof(int), (ns<<1)+1, "PbufIndex");
for (i=0; i<nnns2; i++) NOfPLocal[i] = 0;
for (s=0,tp=0; s<ns*2; s++) {
  TotalP[s] = TotalPNext[s];  PbufIndex[s] = tp;  tp += TotalPNext[s];
}
PbufIndex[s] = totalParts = *totalLocalParticles = tp;  nOfInjections = 0;
for (s=0; s<ns*2; s++)  InjectedParticles[s] = 0;
```

The next part is very level-4p's own. It zero-clears elements of `NOfPGrid`[p][s][$gidx(x,y,z)$] for $p=0$ if the next execution mode is primary or $p \in [0,1]$ if secondary, for all $s \in [0,S)$ and all $(x,y,z) \in [-ke^g, \delta_x(m)+ke^g) \times [-ke^g, \delta_y(m)+ke^g) \times [-ke^g, \delta_z(m)+ke^g)$ where $m=n$ for $p=0$ or $m=parent(n)$ for $p=1$ for the local node n, and $k=1$ for $p=0$ or the helper-helpand tree is kept, or $k=2$ otherwise, by `For_All_Grid()`. Note that this zero-clear is not only for the subdomain's cuboid and sending planes to give the base of the accumulation of particle population for each grid-voxel, but also includes receiving planes for `NOfPGrid`[1][][][] on helpand-helper reconfiguration. This inclusion is to ensure that the sum of each receiving plane in `NOfPGridTotal`[0][][][] given by the reduction of `NOfPGrid`[][][][] in $F(n)$ has always 0 for all grid-voxels in receiving planes for non-periodic system boundaries through which no boundary communications are taken. That is, all receiving planes in `NOfPGrid`[0][][][] are never modified from its initial state with 0 given by `init4p()`, but receiving planes in `NOfPGrid`[1][][][] could have non-zero elements on the reconfiguration because those in *old* secondary subdomain could have neighbors[64].

```
for (ps=0; ps<=Mode_PS(ret); ps++) {
  const int extio = (ps==1 && ret<0) ? OH_PGRID_EXT<<1 : OH_PGRID_EXT;
  for (s=0; s<ns; s++) {
    dint *npg = NOfPGrid[ps][s];
    For_All_Grid(ps, -extio, -extio, -extio, extio, extio, extio)
```

[64] Moreover, the receiving planes could be old sending planes if the reconfiguration *shrank* the subdomain.

```
        npg[The_Grid()] = 0;
      }
    }
```

Finally, we exchange the role of `Particles[]` and `SendBuf[]` as another level-4p's own operation, and return to the caller with the return value defined in §4.3.10 for `transbound1()`. The return value is also set into `currMode` unless it is negative, i.e., `MODE_REB_SEC` $= -1$ which is replaced with `MODE_NORM_SEC` $= 1$ because the library does not care the rebalancing in the last step but will set bit-1 when it finds anywhere accommodation.

```
  tmp = Particles;  Particles = SendBuf;  SendBuf = tmp;
  currMode = ret<0 ? -ret : ret;
  return(ret);
}
```

4.10.10 `try_primary4p()`

`try_primary4p()` The function `try_primary4p()`, called solely from `transbound4p()`, examines if we can stay in or turn to primary mode. If so, the local node gathers all the particles in its primary subdomain from other nodes and sort them according to their grid-position. The function has three arguments `currmode`, `level` and `stats` whose meanings are perfectly equivalent to those of its level-1 counterpart `try_primary1()`.

First we call the level-1 counterpart `try_primary1()` to examine if the next execution mode is primary. If not, we simply return to its caller `transbound4p()` with the return value `FALSE` to indicate the mode will be secondary. Otherwise, i.e., we will be in primary mode, at first we call `update_real_neighbors()` with the operation code `URN_PRI` to reinitialize the elements `RealDstNeighbors`[0][0] and `RealSrcNeighbors`[0][0] so that they have subdomain identifiers neighboring to the local node's primary subdomains[65], if we were in secondary mode.

```
static int
try_primary4p(const int currmode, const int level, const int stats) {
  const int nn=nOfNodes, ns=nOfSpecies, nnns=nn*ns, me=myRank;
  const int oldp = RegionId[1];
  int i, s, nsend, *np;
  dint ***npg = Mode_PS(currmode) ? NOfPGridTotal : NOfPGrid;

  if (!try_primary1(currmode, level, stats)) return(FALSE);
  if (Mode_PS(currmode))  update_real_neighbors(URN_PRI, 0, -1, -1);
```

Next, if we have anywhere accommodataion, we perform position-aware particle transfer as follows. First we perform non-position-aware transfer by calling `move_to_sendbuf_primary()` and `exchange_primary_particles()` to have all particles to be accommodated by the local node in its primary subdomain. Then we call `count_population()` to have the complete per-grid histogram in `NOfPGrid`[0][][][] only for the primary subdomain by telling it to the function through the argument `psnew` $= 0$[66]. Finally, based on the per-grid histogram, we sort particles by `sort_particles()` telling it that the per-grid histogram is in `NOfPGrid`[][][][] which should be copied into `NOfPGridOut`[][][][] (`nextmode` $= 0$), particles

[65] The second, third and fourth argument of `update_real_neighbors()`, `dosec`, `oldp` and `newp`, are meaningless in this call.

[66] The first argument `nextmode` $= 0$ is only for the statistics and thus meaningless in this call.

to be sorted are primary only (`psnew` = 0), and execution time measurement has already started in `count_population()` even if specified (`stats` = 0).

```
if (Mode_Acc(currmode)) {
  move_to_sendbuf_primary(Mode_PS(currmode), stats);
  exchange_primary_particles(currmode, stats);
  count_population(0, 0, stats);
  sort_particles(NOfPGrid, 0, 0, 0);
```

If the accommodation is normal, on the other hand, we at first call `exchange_population()` to gather the per-grid histograms in neighbors' sending planes, possibly with reduction of those in helpers of the local node and the neighbors, to have the complete per-grid histogram in `NOfPGrid`[0][][][] or `NOfPGridTotal`[0][][][] just depending on the current execution mode `currmode` telling it to the function by `nextmode` = 0. Then we sum up the number of particle to be sent to other nodes, namely P_n^{send}, as follows.

$$\begin{aligned} P_n^{\text{send}} &= \sum_{s=0}^{S-1} \sum_{\substack{m=0 \\ m \neq n}}^{N-1} q(n)[0][s][m] + \sum_{s=0}^{S-1} \sum_{m=0}^{N-1} q(n)[1][s][m] \\ &= \sum_{p=0}^{1} \sum_{s=0}^{S-1} \sum_{m=0}^{N-1} q(n)[p][s][m] - q(n)[0][s][n] \\ &= \sum_{p=0}^{1} \sum_{s=0}^{S-1} \sum_{m=0}^{N-1} \texttt{NOfPLocal}[p][s][m] - \texttt{NOfPLocal}[0][s][n] \end{aligned}$$

Note that the accidental particle travels from the local node's secondary subdomain to primary subdomain are considered as sending them as in lower level libraries.

Then, if $P_n + P_n^{\text{send}} = \texttt{TotalPGlobal}[n] + P_n^{\text{send}} > P_{lim} = \texttt{nOfLocalPLimit}$ to mean that we cannot move all particles in `Particles`[] to `SendBuf`[] with sorting, we perform non-position-aware transfer by `move_to_sendbuf_primary()` and `exchange_primary_particles()` to have all primary particles to be accommodated by the local node in `Particles`[], and then sort them by `sort_particles()` telling it that the per-grid histogram is in `NOfPGrid`[][][][] if we were in primary mode or `NOfPGridTotal`[][][][] otherwise, the histogram should be copied into `NOfPGridOut`[][][][] (`nextmode` = 0), and particles to be sorted are primary only (`psnew` = 0).

Otherwise, i.e., $P_n + P_n^{\text{send}} \leq P_{lim}$, we move particles in `Particles`[] to `SendBuf`[] sorting those staying in the primary subdomain by `move_and_sort_primary()` giving it `NOfPGrid`[][][][] or `NOfPGridTotal`[][][][] depending on whether we are in primary mode and telling it whether it needs to scan secondary particles ($parent(n) \geq 0$ in the last step) or not. Then, after letting `SendBuf` points `SendBuf`[P_n] for $sbuf(0,0)$ temporarily, we send particles in $sbuf(s,m)$ and receive particles to $rbuf(0,s)$ for all $s \in [0,S)$ and $m \in [0,N)$ by `exchange_primary_particles()`. Finally, we move received particles in $rbuf(0,s)$ to `SendBuf`[] sorting them by `sort_received_particles()` telling it that particles to be sorted are primary only (`psnew` = 0)[67].

```
} else {
  exchange_population(currmode, 0);
  for (s=0,nsend=0,np=NOfPLocal; s<ns; s++,np+=nn) {
    for (i=0; i<nn; i++)  nsend += np[i] + np[i+nnns];
```

[67] The first argument `nextmode` = 0 is only for the statistics.

```
      nsend -= np[me];
    }
    if (TotalPGlobal[me]+nsend>(dint)nOfLocalPLimit) {
      move_to_sendbuf_primary(Mode_PS(currmode), stats);
      exchange_primary_particles(currmode, stats);
      sort_particles(npg, 0, 0, stats);
    } else {
      struct S_particle *sbuf=SendBuf;
      move_and_sort_primary(npg, (oldp>=0 ? 1 : 0), stats);
      SendBuf += TotalPGlobal[me];
      exchange_primary_particles(currmode, stats);
      SendBuf = sbuf;
      sort_received_particles(0, 0, stats);
    }
  }
```

Finally we finish this function and return to `transbound4p()` with `TRUE` to tell it we will be in primary mode in the next simulation step, after setting `primaryParts` and its shadow pointed by `secondaryBase` to the total number of particles the local node n accommodates, i.e., the number of particles in the primary subdomain, `TotalPGlobal`[n].

```
  primaryParts = *secondaryBase = TotalPGlobal[me];
  return(TRUE);
}
```

4.10.11 `try_stable4p()`

`try_stable4p()` The function `try_stable4p()`, solely called from `transbound4p()`, examines if the current helpand-helper configuration sustains the particle movements crossing subdomain boundaries which can bring intolerable load imbalance. The examination is done by calling its level-1 counterpart `try_stable1()` simply passing all the arguments of the function itself to the counterpart if we have anywhere accommodation, because the meanings of them are perfectly equivalent to those of the counterpart. Otherwise, i.e., with normal accommodation, we pass the `level` argument making it negative to keep `try_stable1()` from making particle transfer schedule which we will build in `exchange_particles4p()`.

If the examination passes, we perform an all-to-all type particle transfer by calling `exchange_particles4p()` with arguments of the function itself. In addition to them, the third argument `reb` = 0 tells it that helpand-helper configuration is kept, and the fourth and fifth arguments `oldp` = `newp` = `RegionId`[1] show the both of old and new helpand of the local node n are *parent*(n). Then the function returns to `transbound4p()` with the return value of `TRUE`.

```
static int
try_stable4p(const int currmode, const int level, const int stats) {
  if (!try_stable1(currmode, (Mode_Acc(currmode) ? level : -level), stats))
    return(FALSE);
  exchange_particles4p(currmode, level, 0, RegionId[1], RegionId[1], stats);
  return(TRUE);
}
```

4.10.12 `rebalance4p()`

`rebalance4p()` The function `rebalance4p()`, solely called from `transbound4p()`, builds the new family tree to rebalance the load among nodes by calling its level-1 counterpart `rebalance1()` simply passing all the arguments of the function itself to the counterpart if we have anywhere accommodation, because the meanings of them are perfectly equivalent to those of the counterpart. Otherwise, i.e., with normal accommodation, we pass the `level` argument making it negative to keep `rebalance1()` from making particle transfer schedule which we will build in `exchange_particles4p()`, and from building new communicators for the new tree because we still need the old ones.

Then, before the particle transfer by `exchange_particles4p()`, it modifies elements of `InjectedParticles`[0][1][s] for all $s \in [0, S)$, if some particles are injected (`nOfInjections` $= Q_n^{\mathrm{inj}} > 0$), we have anywhere accommodation, and old and new *parent*(n) for the local node n are different, because we have to take care the secondary particles injected into new secondary subdomain accidentally. That is, if we have anywhere accommodation, we will at first perform non-position-aware particle transfer in which secondary particles accidentally in the new secondary subdomain are considered to be staying. On the other hand, `InjectedParticles`[0][1][s] has the number of secondary particles injected into the old secondary subdomain. Therefore, we have to scan all injected particles to let `InjectedParticles`[0][1][s] have the number of secondary particles (`Secondary_Injected()` is true) in the new secondary subdomain so that `move_to_sendbuf_secondary()` picks some of them and let them stay in *pbuf*$(1, s)$. Note that we use `Primarize_Id()` to get the subdomain identifier of each secondary injected particle but immediately recover its original form by `Secondarize_Id()` so that `move_to_sendbuf_secondary()` correctly decode its subdomain code[68]. Also note that this operation is level-4p specific because in lower levels particles injected into a subdomain other than the old secondary one are considered primary.

Another remark is that the old *parent*(n) is obtained from `RegionId`[1] before the call of `rebalance1()` which may update the element. Further, the new *parent*(n) is obtained from `NodesNext`[n] in the new family tree if we have normal accommodation, while we have to refer to `Nodes`[n] with anywhere accommodation because `rebalance1()` has exchanged `Nodes`[] and `NodesNext`[] to let the former has the new tree.

Then we call `exchange_particles4p()` with arguments of this function itself. In addition to them, the third argument `reb` $= 1$ tells it helpand-helper reconfiguration, and the fourth and fifth arguments `oldp` and `newp` have the old and new *parent*(n). Then after the call, we do the followings for (potentially) new helpand and secondary subdomain assigned to the local node by rebalancing, if we had normal accommodation; call `set_grid_descriptor()` to update `GridDesc`[1][] for the secondary subdomain; move `Neighbors`[2][k] for the neighbors of the new *parent*(n), which were temporalily stored to keep the old parent's neighbors in transitional state of helpand-helper reconfiguration, into `Neighbors`[1][k] for all $k \in [0, 3^D)$ as the stable state information; and finally call `update_neighbors()` telling it to update elements in `AbsNeighbors`[1][] and `GridOffset`[1][] for the secondary subdomain. Note that the field descriptors for the secondary subdomain has already been updated in `make_recv_list()`. Also note that, if we had anywhere accommodation, these operations have been performed by `exchange_particles4p()`.

```
static void
rebalance4p(const int currmode, const int level, const int stats) {
```

[68] Of course we can examine the subdomain identifier in a non-destructive manner but to design a specific macro for it is tiresome.

```
  const int me=myRank, ns=nOfSpecies;
  const int oldp = RegionId[1],  amode = Mode_Acc(currmode);
  const int ninj = nOfInjections;
  int s, n, newp;

  rebalance1(currmode, (amode ? level : -level), stats);
  newp = amode ? Nodes[me].parentid : NodesNext[me].parentid;
  if (ninj && amode && oldp!=newp) {
    int *sinj = InjectedParticles + ns;
    const int sbase=specBase;
    int i;
    struct S_particle *p;
    Decl_Grid_Info();
    for (s=0; s<ns; s++)  sinj[s] = 0;
    if (newp>=0) {
      for (i=0,p=Particles+totalParts; i<ninj; i++,p++) {
        const OH_nid_t nid = p->nid;
        int sdid;
        if (Secondary_Injected(nid)) {
          Primarize_Id(p, sdid);  Secondarize_Id(p);
          if (sdid==newp)  sinj[Particle_Spec(p->spec-sbase)]++;
        }
      }
    }
  }
  exchange_particles4p(currmode, level, 1, oldp, newp, stats);
  if (!amode) {
    set_grid_descriptor(1, newp);
    for (n=0; n<OH_NEIGHBORS; n++)  Neighbors[1][n] = Neighbors[2][n];
    update_neighbors(1);
  }
}
```

4.10.13 Macros `Parent_Old()`, `Parent_New()`, `Parent_New_Same()` and `Parent_New_Diff()`

`Parent_Old()`
`Parent_New()`
`Parent_New_Same()`
`Parent_New_Diff()`

Before giving the definition of the function `exchange_particles4p()`, we define four macros to examine the statuses of old and (possibly) new parents of the local node n in the last and next simulation steps encoded in the least significant 3 bits of the function's local variable `pcode` $= \pi$ as follows.

- Bit-2 is 1 and thus `Parent_Old`(π) is true iff the old $parent(n) \geq 0$ meaning the old parent exists.
- Bit-1 is 1 and thus `Parent_New`(π) is true iff the new $parent(n) \geq 0$ meaning the new parent exists.
- Bit-0 is 1 iff the old and new $parent(n)$ is equivalent. Therefore, `Parent_New_Same`(π) is true iff the bit-1 and bit-0 are 1 to mean that the local node has the new parent and unmodified. On the other hand, `Parent_New_Diff`(π) is true iff the bit-1 is 1 but bit-0 is 0 to mean that the local node has the really new parent.

The macros are used in `exchange_particles4p()` and its callees; `Parent_Old()` and `Parent_New_Diff()` in `make_send_sched()`, `gather_hspot_send()` and `scatter_hspot_`

recv(), while Parent_New() solely in exchange_particles4p() and Parent_New_Same() solely in make_send_sched().

```
#define Parent_Old(PCODE)       ((PCODE) & 4)
#define Parent_New(PCODE)       ((PCODE) & 2)
#define Parent_New_Same(PCODE)  (((PCODE) & 3) == 3)
#define Parent_New_Diff(PCODE)  (((PCODE) & 3) == 2)
```

4.10.14 exchange_particles4p()

exchange_particles4p()

The function exchange_particles4p(), called from try_stable4p() and rebalance4p(), performs an all-to-all type position-aware particle transfer when we will be in secondary mode in the next simulation step. The function is given arguments currmode, level and stats whose meanings are same as those of the callers, reb being 0 if called from try_stable4p() while 1 if from rebalance4p(), and oldp and newp are the local node's *parent*(n) in the last and next simulation step respectively.

At first in the variable declaration part, the function determines whether we have to take care of the transitional state of helpand-helper configuration, i.e., whetehr we have normal accommodation and rebalancing took place, and let its local variable trans be true iff so. It also sets the parent status code discussed in §4.10.13 according to the arguments oldp and newp.

```
static void
exchange_particles4p(const int currmode, const int level, int reb,
                     int oldp, const int newp, const int stats) {
  const int trans = !Mode_Acc(currmode) && reb ? 1 : 0;
  int pcode =
    (oldp>=0 ? 4 : 0) + (newp>=0 ? 2 : 0) + (oldp==newp ? 1 : 0);
  int psold, psnew;
  int nacc[2]={0,0}, nsend=0;
  struct S_commlist *hslist;
```

If we have anywhere accommodation, try_stable1() called from try_stable4p() or rebalance1() called from rebalance4p() built particle transfer schdule in CommList[] and set SecRList, SecRLSize and SLHeadTail[] to appropriate values by schedule_particle_exchange() as done in lower level non-position-aware particle transfer. Therefore, we can call exchange_particles() as we do in try_stable2() or rebalance2() with anywhere accommodation[69], to have primary and secondary particles of the local node without position-aware manner. Then if rebalanced, we call the followings for the (potentially) new *parent*(n) assigned to the local node by rebalancing; update_descriptors() giving old and new *parent*(n) to update elements in FieldDesc[] for the new *parent*(n) and to reinitialize BorderExc[][1][][] for old *parent*(n); set_grid_descriptor() to update GridDesc[1] for the new *parent*(n); update_neighbors() to update AbsNeighbors[1][] and GridOffset[1][]; and update_real_neighbors() with the operation code URN_SEC and the new *parent*(n) to update RealDstNeighbors[0][p] and RealSrcNeighbors[0][p] for $p \in [0, 1]$ to reflect the new helpand-helper configuration. Note that the second argument dosec = 0 of update_

[69] The third argument oldparent is the oldp argument of this function, but it is not referred to in exchange_particles when neighboring is 0 to mean anywhere accommodation.

real_neighbors() does not have any effect because it is meaningful when the operation code is URN_TRN.

Then we reinitialize NOfSend[][][] by zero-clearing its all elements as the base of counting in callee functions of make_send_sched(), because make_comm_count() called from try_stable1() or rebalance1() let it have the number of sending particles in the non-position-aware particle transfer so that exchange_particles() refers to it in the all-to-all communication for anywhere accommodation. After that, we call count_population() to have local per-grid histogram in NOfPGrid[p][][] telling it to do for both $p \in [0, 1]$ if the local node has new *parent*(n) (psnew = 1), and then reduce_population() with the function pointer to mpi_allreduce_wrapper() to have the complete per-grid histogram in NOfPGridTotal[p][][] for $p \in [0, 1]$, i.e., not only for the histogram of primary subdomain but also of secondary one by all-reduce communication. We also let reb be 0 because we do not have to take care the helpand-helper reconfiguration and thus make old and new parent same letting pcode have the corresponding code just depending on the existance of the new parent.

On the other hand, if we have normal accommodation, we call exchange_population() forcing it to bulid the complete per-grid histogram in NOfPGridTotal[0][][] regardless the current execution mode currmode.

```
if (Mode_Acc(currmode)) {
  int i;
  const int nnns2 = nOfNodes * nOfSpecies * 2;
  if (reb) {
    exchange_particles(SecRList, SecRLSize, oldp, 0, currmode, stats);
    update_descriptors(oldp, newp);
    set_grid_descriptor(1, newp);
    update_neighbors(1);
    update_real_neighbors(URN_SEC, 0, -1, newp);
  }
  else
    exchange_particles(CommList+SLHeadTail[1], SecSLHeadTail[0], oldp, 0,
                       currmode, stats);
  for (i=0; i<nnns2; i++)  NOfSend[i] = 0;
  count_population(1, (Parent_New(pcode) ? 1 : 0), 0);
  reduce_population(mpi_allreduce_wrapper);
  reb = 0;  oldp = newp;  pcode = newp>=0 ? 7 : 0;
} else
  exchange_population(currmode, 1);
```

Now, regardless of the accommodation mode, we have the complete per-grid histogram in NOfPGridTotal[0][][] and particles in Particles[] which will stay in the local node's primary or secondary subdomain or travel to one of their neighbors. Therefore with this common setting, we call make_recv_list() with arguments currmode, level, (possibly modified) reb and stats, together with the old and (possibly diffenet) new parents to build the receiver-side particle transfer schudule in CommList[] and to obtain its tail and thus the head of hot-spot sending block as the return value. The head is passed to the next call of make_send_sched() with other arguments including the parent status code pcode. The function builds the per-receiver sending histogram in NOfSend[][][] and gives us the number of primary/secondary particles to be accommodated by the local node in the local array nacc[2] and the number of sending particles P_n^{send} in the local variable nsend. Finally we exchange NOfSend[][][] by a hand-made all-to-all communicaion in neighboring families to

have NOfRecv[p][][] for $p \in \{0,1\}$ if the new parent exists or only for $p = 0$ otherwise, by exchange_xfer_amount() which takes care of the transitional helpand-helper configuration if trans is true.

```
psold = Parent_Old(pcode) ? 1 : 0;
psnew = Parent_New(pcode) ? 1 : 0;
hslist = make_recv_list(currmode, level, reb, oldp, newp, stats);
make_send_sched(currmode, reb, pcode, oldp, newp, hslist, nacc, &nsend);
exchange_xfer_amount(trans, psnew);
```

Now we start position-aware particle transfer. If $Q_n + P_n^{\text{send}} > P_{lim} =$ nOfLocalPLimit, where $Q_n =$ nacc[0] + nacc[1], to mean we cannot move all particles in Particles[] to SendBuf[] with sorting, we perform a partially position-aware transfer only takeing care of the node to accommodate particles in each grid-voxel. This is done by move_to_sendbuf_sec4p() being the level-4p version of move_to_sendbuf_secondary(), to which we give an argument psold to indicate whether the local node's old parent exists and thus it should take care secondary particles. Then we call xfer_particles(), whose second argument psnew is true iff the local node's new parent exists and thus we need to receive secondary particles, and third argument sbuf = SendBuf means *sbuf*$(0,0,0)$ is located at the head of SendBuf[] as usual, to have particles to accommodate in Particles[]. Finally, we move them with sorting to SendBuf[] by sort_particles() telling it the per-grid histogram is in NOfPGridOut[][][] and, if the local node's new parent exists (newp = 1), both of primary and secondary particles have to be sorted referring to GridDesc[1] or GridDesc[2] depending on whether helpand-helper configuration is stable (nextmode = 1) or transitional (nextmode = 2). Note that the first argument of sort_particles() is NULL but not referred to in the function.

Otherwise, i.e., $Q_n + P_n^{\text{send}} \le P_{lim}$, we move particles in Particles[] to SendBuf[] sorting those staying in the primary/secondary subdomains by move_and_sort_secondary(). Its argument psold indicates whether it should scan old secondary particles, while psnew indicates whether NOfPGridOut[1][][], NOfPGridTotal[1][][] and RecvBufBases[1][] should be built for new secondary particles. Then we transfer particles by xfer_particles() telling it that *sbuf*$(0,0,0)$ is at SendBuf[Q_n]. Finally, we move received particles in *rbuf*(p,s) to SendBuf[] sorting them by sort_received_particles() telling it that it should sort for both of $p \in [0,1]$ (psnew = 1) or only for $p = 0$ (psnew = 0).

```
    if ((dint)nacc[0]+(dint)nacc[1]+nsend>(dint)nOfLocalPLimit) {
      move_to_sendbuf_sec4p(psold, trans, oldp, nacc, nsend, stats);
      xfer_particles(trans, psnew, SendBuf);
      sort_particles(NULL, trans+1, psnew, stats);
    } else {
      move_and_sort_secondary(psold, psnew, trans, oldp, nacc, stats);
      xfer_particles(trans, psnew, SendBuf+nacc[0]+nacc[1]);
      sort_received_particles(1, psnew, stats);
    }
  }
```

4.10.15 exchange_population()

exchange_population()

The function exchange_population(), called from try_primary4p() and exchange_particles4p() when we have normal accomodation, sums up local per-grid histograms in the local node's primary family if we were in secondary mode, and then gathers the

per-grid histograms in neighbors' sending planes to have the complete per-grid histogram in `NOfPGrid`[0][][][] if we were and will be in primary mode, or in `NOfPGridTotal`[0][][][] otherwise. The execution mode of the last step is given by the argument `currmode` while that of the next step is given by `nextmode`.

```
static void
exchange_population(const int currmode, const int nextmode) {
  const int ns=nOfSpecies;
  int s;
  dint **npg = NOfPGrid[0];
  const int ct=nOfExc-1;
  const int exti = OH_PGRID_EXT,  exto = exti<<1;
  const int x = GridDesc[0].x,  y = GridDesc[0].y,  z = GridDesc[0].z;
  const int w = GridDesc[0].w,  dw = GridDesc[0].dw;
  Decl_For_All_Grid();
```

At first, if we were in secondary mode, we sum up local per-grid histograms `NOfPGrid`[p][][][] in the local node's primary family, where $p = 0$ for the local node while $p = 1$ for its helpers, to have the sum in `NOfPGridTotal`[0][][][] by `reduce_population()` and its argument function `MPI_Reduce()`. On the other hand, if we were in primary mode but will be in secondary mode, we copy elements `NOfPGrid`[0][s][$gidx(x, y, z)$] to the corresonding elements in `NOfPGridTotal`[][][][] using `For_All_Grid()` for all $s \in [0, S)$ and $(x, y, z) \in [-e^g, \delta_x(n)+e^g) \times [-e^g, \delta_y(n)+e^g) \times [-e^g, \delta_z(n)+e^g)$ for the local node n, because we need to keep `NOfPGrid`[][][][] unchanged for the secondary mode particle transfer. Therefore, if we were or will be in secondary mode, the *base* per-grid histogram is built in `NOfPGridTotal`[0][][][], while untouched `NOfPGrid`[0][][][] is used as the base otherwise.

```
  if (Mode_PS(currmode)) {
    reduce_population(MPI_Reduce);  npg = NOfPGridTotal[0];
  } else if (nextmode) {
    npg = NOfPGridTotal[0];
    for (s=0; s<ns; s++) {
      dint *npgs = NOfPGrid[0][s],  *npgt = npg[s];
      For_All_Grid(0, -exti, -exti, -exti, exti, exti, exti)
        npgt[The_Grid()] = npgs[The_Grid()];
    }
  }
```

Now, for each $s \in [0, S)$, we gather sending planes of all $2D$ neighbors to the local node's receiving planes by `oh3_exchange_borders()` giving it the base per-grid histogram `NOfPGrid`[0][s][] or `NOfPGridTotal`[0][s][] and $C-1$ being the entry for per-grid histograms in `BorderExc`[][][][][]. Its second argument for the secondary subdomain's array is `NULL` because we don't broadcast the receiving planes to the helpers as indicated its forth argument `bcast` $= 0$.

Then we add each of $2D$ receiving planes to each boundary plane(s) of d-th dimensional from $d = D - 1$ to 0 to have the complete per-grid histogram. The addition is performed by a series of calls of `add_population()` for each receiving/boundary plane pair. More specifically, the d-th dimensional boundary plane(s) to which we add receiving plane(s) is

specified as $[\beta_0^l, \beta_0^u) \times \cdots \times [\beta_{D-1}^l, \beta_{D-1}^u)$ where $[\beta_k^l, \beta_k^u)$ is specified as follows.

$$[\beta_k^l, \beta_k^u) = \begin{cases} [-2e^g, \delta_k(n) + 2e^g) & k < d \\ [0, e^g) & k = d \text{ and lower} \\ [\delta_k(n) - e^g, \delta_k(n)) & k = d \text{ and upper} \\ [0, \delta_k(n)) & k > d \end{cases}$$

The function is given the arguments for the base per-grid histogram, the lower and upper bound of the boundary plane(s) in each axis shown above, and the offset of the receiving plane(s) from boundary plane(s) namely;

$$gidx(\beta_0, \cdots, \beta_d \pm 2e^g, \cdots, \beta_{D-1}) - gidx(\beta_0, \cdots, \beta_d, \cdots, \beta_{D-1}) = \pm 2e^g \prod_{k=0}^{d-1} (\delta_k^{\max} + 4e^g)$$

where $-/+$ for lower/upper boundary. The method for the addition is same as that we used in the sample code's function `add_boundary_current()` shown in §3.13.

```
  for (s=0; s<ns; s++) {
    oh3_exchange_borders(npg[s], NULL, ct, 0);
    if (OH_DIMENSION>OH_DIM_Z) {
      add_population(npg[s], -exto, x+exto, -exto, y+exto, 0, exti, -dw*exto);
      add_population(npg[s], -exto, x+exto, -exto, y+exto, z-exti, z, dw*exto);
    }
    if (OH_DIMENSION>OH_DIM_Y) {
      add_population(npg[s], -exto, x+exto, 0, exti, 0, z, -w*exto);
      add_population(npg[s], -exto, x+exto, y-exti, y, 0, z, w*exto);
    }
    add_population(npg[s], 0, exti, 0, y, 0, z, -exto);
    add_population(npg[s], x-exti, x, 0, y, 0, z, exto);
  }
}
```

4.10.16 `add_population()`

`add_population()` The function `add_population()`, called solely from `exchange_population()` but $2DS$ times, adds the elements in a receiving plane (set) of per-grid histogram, specified by its argument `npd` being `NOfPGrid`$[0][s][]$ or `NOfPGridTotal`$[0][s][]$, to the boundary plane (set) specified by arguments as $[\texttt{xl}, \texttt{xu}) \times [\texttt{yl}, \texttt{yu}) \times [\texttt{zl}, \texttt{zu})$, whose values are shown in §4.10.15. The location of the receiving plane (set) is specified as the distance between corresponding elements in receiving/boundary plane (sets) by the argument `src` being $\pm 2e^g \prod_{k=0}^{d-1}(\delta_k^{\max} + 4e^g)$ as discussed in §4.10.15. The function simply performs the addition for specified elements by `For_All_Grid_Abs()`.

```
static void
add_population(dint *npd, const int xl, const int xu, const int yl,
               const int yu, const int zl, const int zu,
               const int src) {
  dint *nps=npd+src;
  Decl_For_All_Grid();

  For_All_Grid_Abs(0, xl, yl, zl, xu, yu, zu)
    npd[The_Grid()] += nps[The_Grid()];
}
```

4.10.17 mpi_allreduce_wrapper()

mpi_allreduce_wrapper() The function mpi_allreduce_wrapper(), appearing solely in exchange_particles4p() as the argument of reduce_population(), accepts the same argument set as MPI_Reduce() and calls MPI_Allreduce() passing all arguments except for root in the set of MPI_Reduce() but not in that of MPI_Allreduce(). This function is just for using reduce_population() for all-reduce communication rather than simple non-all type reduction.

```
static int
mpi_allreduce_wrapper(void *sendbuf, void *recvbuf, int count,
                      MPI_Datatype datatype, MPI_Op op, int root,
                      MPI_Comm comm) {
  return(MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm));
}
```

4.10.18 reduce_population()

reduce_population() The function reduce_population(), called from exchange_particles4p() and exchange_population(), receives its sole argument mpired being mpi_allreduce_wrapper() for the former and MPI_Reduce() for the latter, and performs (all-)reduce communication using it in primary ($p = 0$) and secondary ($p = 1$) family members to sum up NOfPGrid[p][][] to have the sum in NOfPGridTotal[0][][] and, if all-reducing, NOfPGridTotal[1][][].

The function is almost equivalent to oh1_all_reduce() and oh1_reduce() but we have to have this variation because the source array NOfPGrid[][][] must be kept unchanged rather than being overwritten it by MPI_IN_PLACE option that level-1 relatives specify. Therefore, if the prime element of MyComm is MPI_COMM_NULL to mean that the local node has no children and thus (all-)reduce operation is not performed, we have to copy NOfPGrid[0][][] into NOfPGridTotal[0][][] explicitly by memcpy(). The base index and the number of elements to be (all-)reduced are specified in FieldDesc[$F-1$].red.base and its element size[p] for the per-grid histogram.

```
static void
reduce_population(int (*mpired)(void*, void*, int, MPI_Datatype, MPI_Op, int,
                                MPI_Comm)) {
  const int ft=nOfFields-1;
  const int base = FieldDesc[ft].red.base;
  const int *size = FieldDesc[ft].red.size;

  if (MyComm->black) {
    if (MyComm->prime!=MPI_COMM_NULL)
      mpired(NOfPGrid[0][0]+base, NOfPGridTotal[0][0]+base, size[0],
             MPI_LONG_LONG_INT, MPI_SUM, MyComm->rank, MyComm->prime);
    if (MyComm->sec!=MPI_COMM_NULL)
      mpired(NOfPGrid[1][0]+base, NOfPGridTotal[1][0]+base, size[1],
             MPI_LONG_LONG_INT, MPI_SUM, MyComm->root, MyComm->sec);
  } else {
    if (MyComm->sec!=MPI_COMM_NULL)
      mpired(NOfPGrid[1][0]+base, NOfPGridTotal[1][0]+base, size[1],
             MPI_LONG_LONG_INT, MPI_SUM, MyComm->root, MyComm->sec);
    if (MyComm->prime!=MPI_COMM_NULL)
      mpired(NOfPGrid[0][0]+base, NOfPGridTotal[0][0]+base, size[0],
```

```
            MPI_LONG_LONG_INT, MPI_SUM, MyComm->rank, MyComm->prime);
  }
  if (MyComm->prime==MPI_COMM_NULL)
    memcpy(NOfPGridTotal[0][0]+base, NOfPGrid[0][0]+base,
           size[0]*sizeof(dint));
}
```

4.10.19 make_recv_list()

make_recv_list()

The function make_recv_list(), called solely from exchange_particles4p(), scans per-grid histogram to build primary receiving block, and then exchanges the block between neighbors to have primary sending block and broadcast them for secondary receiving, secondary sending and alternative secondary receiving blocks for helpers. Its arguments currmode, level, reb and stats are perfectly equivalent to those of the caller exchange_particles4p(), while oldp and newp are parents in the last and next simulation step. The function returns the pointer to the record next to the last block which it builds, or in other words the head of hot-spot sending.

```
static struct S_commlist*
make_recv_list(const int currmode, const int level, const int reb,
               const int oldp, const int newp, const int stats) {
  const int me = myRank, ns=nOfSpecies, nn=nOfNodes, nnns=nn*ns;
  const int nn2 = nn<<1;
  struct S_node *nodes = reb ? NodesNext : Nodes;
  struct S_node *mynode = nodes + me;
  struct S_node *ch;
  struct S_recvsched_context
    context = {0, 0, 0, 0, 0, 0, 0, 0, CommList};
  int rlsize, rlidx;
  const int ft=nOfFields-1;
  const int npgbase = FieldDesc[ft].bc.base;
  const int *npgsize = FieldDesc[ft].bc.size;
  const int lastg =
    Coord_To_Index(GridDesc[0].x-1, GridDesc[0].y-1, GridDesc[0].z-1,
                   GridDesc[0].w, GridDesc[0].dw);
  struct S_commlist *lastrl;
  int i;
```

First, the function builds primary receiving block by calling sched_recv() for the local node's helpers and then the node itself to determine the grid-voxels to be accommodated by them and to find hot-spots. The order of the node scannig, helpers first and then the node itself, is to make the scanning orders of hot-spot senders and receivers coherent. That is, as we will discuss in §4.10.26, since the function scatter_hspot_send() scans the hot-spot senders in helper-first manner, the coherent order of receiver scanning will minimize the amount of hot-spot particles transferred among the family members involved in the hot-spot.

The arguments for the function, besides trivial currmode and reb, are as follows, where *NN* is Nodes[] if reb is false, or NodesNext[] otherwise, for *new* helpands.

- get is $R_n^{\mathrm{get}} = NN[n]$.get.prime for the local node n, or $Q_m^{\mathrm{get}} = NN[m]$.get.sec for its helper m, to specify the baseline number of receiving (if positive) or sending (if negative) primary/secondary particles of n or m.

- `stay` is $Q_n^n = NN[n]$`.stay.prime` for the local node n, or $Q_m^n = NN[m]$`.stay.sec` for its helper m, to specify the number of primary/secondary particles currently accommodated by n or m. The value is not useful for helpers if we have normal accommodation and helpand-helper reconfiguration is not taking place, but `get` has the base line number of particles to be accommodated by the helper in this case.
- `nid` is the node identifier of the local node n or its helper m.
- `tag` is 0 for the local node, or NS for its helpers, to distinguish helpand and helpers and to be set into `S_commlist` record element `tag`.
- `context` is a `S_recvsched_context` structure discussed in §4.9.4, whose elements `x`, `y`, `z`, `g`, `hs`, `nptotal`, `nplimit`, `carryover` are 0 at initial, while `cptr` is initialized to point the head of `CommList`[].

Now we have $\rho = \sigma =$ `context.cptr` $-$ `CommList` records in primary receiving block, but its last record does not necessary has the largest grid-position at $g_{\max} = gidx(\delta_x(n)-1, \delta_y(n)-1, \delta_z(n)-1)$ of the interior of the primary subdomain in its `region` element, because the grid-voxel can be empty. If so, we need to make the last record's `region` have $g_{\max}$ but we cannot simply do it by overwriting the record in two cases. One extreme case is that the subdomain has no particles and thus the primary receiving block is empty. The other is that the last record is for a hot-spot with which the last node cannot have particles any more. In both cases, we add a record to assign all empty grid-voxels up to g_{max} to the local node n for the former or the last node for the latter letting `region` element be $g_{\max}$.

```
for (ch=mynode->child; ch; ch=ch->sibling)
  sched_recv(currmode, reb, ch->get.sec, ch->stay.sec, ch->id, nnns,
             &context);
sched_recv(currmode, reb, mynode->get.prime, mynode->stay.prime, me, 0,
           &context);
rlidx = rlsize = context.cptr - CommList;  lastrl = context.cptr - 1;
if (rlsize==0 || (lastrl->region<lastg && lastrl->count)) {
  struct S_commlist *rl = lastrl + 1;
  rl->rid = rlsize ? lastrl->rid : me;
  rl->tag = 0;  rl->sid = 0;  rl->count = 0;  rl->region = lastg;
  rlidx = ++rlsize;
} else
  lastrl->region = lastg;
```

If we have anywhere accommodation, we have already transferred particles among (possibly) all nodes in the first non-position-aware phase. Therefore, the position-aware particle transfer schedule we need is just for the transfer among the nodes in a helpand-helper family. Thus the local node broadcasts the size of its primary receiving block ρ and then the block itself by `oh1_broadcast()`, while these calls also let it receive its helpand's primary receiving block as its secondary receiving block to `SecRList`[] starting from `CommList`[ρ]. Finally we return to the caller with the pointer to `SecRList`[ρ'] where ρ' is the size of secondary receiving block given by the helpand by the first `oh1_broadcast()`, or $\rho' = 0$ if the local node does not have helpand, to let the next block be hot-spot sending block.

```
if (Mode_Acc(currmode)) {
  SecRList = CommList + rlidx;  rlsize = 0;
  oh1_broadcast(&rlidx, &rlsize, 1, 1, MPI_INT, MPI_INT);
  oh1_broadcast(CommList, SecRList, rlidx, rlsize, T_Commlist, T_Commlist);
```

```
  return(SecRList+rlsize);
}
```

Otherwise, i.e., we have normal accommodation, the local node n exchanges its primary receiving block between its neighbors to have primary sending blocks. We scan all 3^D neighbors in `DstNeighbors`$[k]$ and `SrcNeighbors`$[k]$ to send/receive primary receiving/ primary sending block as follows.

- If `DstNeighbors`$[k] = n$, we skip the exchanging communication but let `RLIndex`$[k]$ be 0 to assume we received primary receiving block itself as the primary sending block with which `make_send_sched()` builds sending schedule of the local node's own primary particles to helpers, and then have secondary receiving block as the secondary sending block for sending secondary particles to the helpand and/or sibling helpers. Note that we check neither `SrcNeighbors`$[k]$ nor the non-first occurrence of n in `DstNeighbors`[] or `SrcNeighbors`[] because it is assured that `SrcNeighbors`$[k]$ is n or $-(n+1)$ if `DstNeighbors`$[k]$ is n or $-(n+1)$ respectively, by the symmetry of neighboring. Also note that though we let `RLIndex`$[k'] =$ `RLIndex`$[k] = 0$ for the non-first occurrence of n at k', the primary receiving block is processed by `make_send_sched()` just once for $k = \lfloor 3^D/2 \rfloor$ because it is assured that if a node itself is its neighbor its corresponding exterior, i.e., sending plane/edge/vertex (set) does not have any particles.
- If `SrcNeighbors`$[k] = m_s \geq 0$ to mean the first occurrence of m_s, we receive some, but at most $2N$, records of m_s's primary receiving block into k-th primary sending block from `CommList`$[\rho]$ and let `RLIndex`$[k] = \rho$. We also send σ records in primary receiving block to `DstNeighbors`$[k] = m_d$ at the same time by `MPI_Sendrecv()` if $m_d \geq 0$, or only perform the reception by `MPI_Recv()`. The size of the received block is obtained by `MPI_Get_count()` and ρ is incremented by the size.
- If `SrcNeighbors`$[k] = m_s < 0$ to mean the second or subsequent occurrence of $-(m_s+1)$, we let `RLIndex`$[k]$ be that of $k' =$ `FirstNeighbor`$[k]$ where `SrcNeighbors`$[k']$ should have m_s. By this operation, `make_send_sched()` can refer to the primary sending block obtained from m_s as the k-th neighbor, for which the exterior is different from that for k'-th neighbor. In addition, if `DstNeighbors`$[k] = m_d \geq 0$, we send σ records in primary receiving block to m_d by `MPI_Send()`.

```
for (i=0; i<OH_NEIGHBORS; i++) {
  const int dst=DstNeighbors[i], src=SrcNeighbors[i];
  int rc;
  MPI_Status st;
  if (dst==me) {
    RLIndex[i] = 0;  continue;
  }
  if (src>=0) {
    RLIndex[i] = rlidx;
    if (dst>=0)
      MPI_Sendrecv(CommList, rlsize, T_Commlist, dst, 0,
                   CommList+rlidx, nn2, T_Commlist, src, 0, MCW, &st);
    else
      MPI_Recv(CommList+rlidx, nn2, T_Commlist, src, 0, MCW, &st);
    MPI_Get_count(&st, T_Commlist, &rc);  rlidx += rc;
  } else {
```

```
    if (dst>=0)
      MPI_Send(CommList, rlsize, T_Commlist, dst, 0, MCW);
    RLIndex[i] = (src<-nn) ? rlidx : RLIndex[FirstNeighbor[i]];
  }
}
```

Now the local node has primary sending block with `RLIndex`$[k]$ for all $k \in [0, 3^D)$. Then we let `RLIndex`$[3^D] = \rho$ so that it has the combined size of primary receiving and primary sending blocks, or in other words the index of `CommList`[] of the head of next block, secondary receiving or alternative secondary receiving block. We also let `SecRList` points the next block head, as well as `AltSecRList` in case we don't have secondary receiving block.

Then if we were in secondary mode, the local node broadcast `RLIndex`[] and then primary receiving and primary sending blocks to its helpers by `oh1_broadcast()`, which also let the node have secondary receiving and secondary sending blocks, whose combined size is `SecRLIndex`$[3^D]$, in `SecRList`[] and indices of them in `SecRLIndex`[]. We let ρ be the combined size of all primary receiving, primary sending, secondary receiving and secondary sending blocks and let `AltSecRList` points `CommList`$[\rho]$. Note that `SecRLIndex`$[3^D]$ remains 0 if the local node does not have helpand.

Then if helpand-helper reconfiguration is taking place, we call `build_new_comm()` to build new communicators for the new primary/secondary families of the local node. Its second argument `−level` tells the function not to call `make_comm_count()` being unnecessary for position-aware particle transfer, and the third argument `nbridx` $= 2$ is to have the neighbors of the new helpand in `Neighbors`[2]. We also call the followings; `update_descriptors()` to update `FieldDesc`[] for the new secondary subdomain which will be referred to by the `oh1_broadcast()` shortly; `set_grid_descriptor()` to let `GridDesc`[2][] has the shape of per-grid histogram for the new secondary subdomain; and `update_real_neighbors()` with the code `URN_TRN` to let `RealDstNeighbors`[0][][] and `RealSrcNeighbors`[0][][] have neighoring information according to the new families while `RealDstNeighbors`[1][][] and `RealSrcNeighbors`[1][][] have that according to transitional state representing new/old helpers of old/new helpand's neighbors. Then the local node broadcasts the size of primary receiving block σ and then the block itself to its helpers by `oh1_broadcast()`, which also let the node have alternative secondary receiving block and its size (being 0 if it does not have helpand) to be added to ρ to know the next hot-spot sending block starts from `CommList`$[\rho]$.

Finally, the local node broadcasts per-grid histogram `NOfPGridTotal`[0][][] to (possibly) new helpers referring to `FieldDesc`$[F-1]$`.bc.{base,size`[]} by `oh1_broadcast()`, which also lets the node have its helpand's per-grid histogram in `NOfPGridTotal`[1][][], before returning to the caller with the pointer to the head of hot-spot sending block namely `CommList`$[\rho]$.

```
RLIndex[OH_NEIGHBORS] = rlidx;  SecRLIndex[OH_NEIGHBORS] = 0;
AltSecRList = SecRList = CommList + rlidx;
if (Mode_PS(currmode)) {
  oh1_broadcast(RLIndex, SecRLIndex, OH_NEIGHBORS+1, OH_NEIGHBORS+1,
                MPI_INT, MPI_INT);
  oh1_broadcast(CommList, SecRList, rlidx,
                SecRLIndex[OH_NEIGHBORS], T_Commlist, T_Commlist);
  AltSecRList = CommList + (rlidx += SecRLIndex[OH_NEIGHBORS]);
}
if (reb) {
  int altrlsize = 0;
```

```
      build_new_comm(currmode, -level, 2, stats);
      update_descriptors(oldp, newp);
      set_grid_descriptor(2, newp);
      update_real_neighbors(URN_TRN, Mode_PS(currmode), oldp, newp);
      oh1_broadcast(&rlsize, &altrlsize, 1, 1, MPI_INT, MPI_INT);
      oh1_broadcast(CommList, AltSecRList, rlsize, altrlsize,
                    T_Commlist, T_Commlist);
      rlidx += altrlsize;
    }
    oh1_broadcast(NOfPGridTotal[0][0]+npgbase, NOfPGridTotal[1][0]+npgbase,
                  npgsize[0], npgsize[1], MPI_LONG_LONG_INT, MPI_LONG_LONG_INT);
    return(CommList+rlidx);
  }
```

4.10.20 `sched_recv()`

Prior to discuss the funcion `sched_recv()`, we show three macros, `Sched_Recv_Check()`, `Sched_Recv_Return()` and `For_All_Grid_From()` used solely in the function.

`Sched_Recv_Check()` The first macro `Sched_Recv_Check`(l, g) is the kernel of the iteration visiting the grid-voxel at g in the loop scanning grid-voxels in `sched_recv()`, if $l \neq 0$. Otherwise ($l = 0$), the macro is used before the function enters the loop to cope with particles carried over from the previous reciever node due to a hot-spot at g.

The macro refers to, besides its arguments, the following elements of `S_recvsched_context` structure given to `sched_recv()` as an argument of the function, or their *cached* version in local variables and with possible modification, for the call of `sched_recv()` with the f-th ($f \in [0, |F(n)|)$ member m_f of the local node n's primary family where $\mathcal{P}_T(g)$ defined as follows.

$$\mathcal{P}_T(g) = \sum_{s=0}^{S-1} \mathcal{P}_T(0, s, g) = \sum_{s=0}^{S-1} \texttt{NOfPGridTotal}[0][s][g]$$

- `nptotal` $= \mathcal{P}_\Sigma(g) = \sum_{i \leq g} \mathcal{P}_T(i)$ is the number of particles in grid-voxels we have already scanned including g itself.
- `nplimit` $= \mathcal{P}_\Lambda(f) = \sum_{i=0}^{f} Q^n_{m_i}$ is the sum of number of particles which the node m_i ($i \in [0, f]$) is expected to accommodate.
- `cptr` is the pointer to a record of primary receiving block in `CommList[]` into which the receiving schedule for m_f is stored.

The macro examines if $\mathcal{P}_\Sigma(g) < \mathcal{P}_\Lambda(f)$ to mean that m_f may accommodate more particles in grid-voxels beyond g. If so, the macro does nothing to let the loop in `sched_recv()` continue to visit the next grid-voxel. Otherwise, i.e., $\mathcal{P}_\Sigma(g) \geq \mathcal{P}_\Lambda(f)$, it lets `region` element of the `S_commlist` record pointed by `ctpr` be g, to mean that we assign grid-voxels up to g to m_f. Then it examines if $\mathcal{P}_\Sigma(g) - \mathcal{P}_\Lambda(f) < 2P_{hot}$ to mean the excess of $\mathcal{P}_\Sigma(g)$ over $\mathcal{P}_\Lambda(f)$ is acceptable. If so, the macro lets the loop iterate once more but lets it stop before visiting the next grid-voxel so that `sched_recv()` returns to its caller telling that the next call will visit the next grid-voxel. In this case, we have assigned whole particles in the grid-voxel at g to m_f so that it will accommodate $\mathcal{Q}^n_f$ particles where

$$\mathcal{Q}^n_f = \mathcal{P}_\Sigma(g) - \sum_{i=0}^{f-1} \mathcal{Q}^n_i = \mathcal{P}_\Sigma(g) - \mathcal{Q}_\Sigma(f-1)$$

and thus

$$\mathcal{P}_\Lambda(f) \le \mathcal{Q}_\Sigma(f) = \mathcal{P}_\Sigma(g) < \mathcal{P}_\Lambda(f) + 2P_{hot}$$

Otherwise, i.e., $\mathcal{P}_\Sigma(g) - \mathcal{P}_\Lambda(f) \ge 2P_{hot}$, we found a hot-spot. In this case, `count` element of the `S_commlist` record is set to $q_{hot}(f) = \lceil (\mathcal{P}_\Lambda(f) - \mathcal{P}_\Sigma(g-1))/P_{hot} \rceil P_{hot}$ to mean we assign hot-spot particles of the smallest multiple of P_{hot} to m_f so that $\mathcal{Q}_\Sigma(f) \ge \mathcal{P}_\Lambda(f)$. Note that since $q_{hot}(f) - P_{hot} < \mathcal{P}_\Lambda(f) - \mathcal{P}_\Sigma(g-1)$ and $\mathcal{P}_\Sigma(g) = \mathcal{P}_\Sigma(g-1) + \mathcal{P}_T(g)$, the carryover amount $q_{co}(f+1)$ to be set into `carryover` element of `context`

$$q_{co}(f+1) = \mathcal{P}_T(g) - q_{hot}(f) > \mathcal{P}_\Sigma(g) - \mathcal{P}_\Lambda(f) - P_{hot} \ge 2P_{hot} - P_{hot} = P_{hot}$$

is sufficiently large for the hot-spot spliting. Also note that since

$$q_{hot}(f) < \mathcal{P}_\Lambda(f) - \mathcal{P}_\Sigma(g-1) + P_{hot}$$

we have

$$\mathcal{Q}_\Sigma(f) = \mathcal{P}_\Sigma(g-1) + q_{hot}(f) < \mathcal{P}_\Lambda(f) + P_{hot}$$

Then the macro invokes `Sched_Recv_Return()` to directly return from `sched_recv()` keeping the coordinate and index of grid-voxel unchanged so that they corresponds to the hot-spot we are now visiting.

Therefore, in both cases we have $\mathcal{P}_\Lambda(f) \le \mathcal{Q}_\Sigma(f) < \mathcal{P}_\Lambda(f) + 2P_{hot}$ and it is satisfied that

$$\mathcal{P}_\Lambda(0) \le \mathcal{Q}_\Sigma(0) = \mathcal{Q}_0^n < \mathcal{P}_\Lambda(0) + 2P_{hot} = Q_{m_0}^n + 2P_{hot}$$

for $f = 0$, a simple induction leads us that

$$\mathcal{Q}_f^n = \mathcal{Q}_\Sigma(f) - \mathcal{Q}_\Sigma(f-1) < \mathcal{P}_\Lambda(f) + 2P_{hot} - \mathcal{P}_\Lambda(f-1) = Q_{m_f}^n + 2P_{hot}$$

to make it sure that the excess of the number of particles assigned to m_f over that expected is less than $2P_{hot}$.

```
#define Sched_Recv_Check(INLOOP, G) {\
  if (nptotal>=nplimit) {\
    cptr->region = G;\
    if (nptotal-nplimit>ovflimit) {\
      const int thresh = ovflimit>>1;\
      const int count = (((nplimit-(nptotal-npt)-1)/thresh) + 1) * thresh;\
      cptr->count = count;  carryover = npt - count;\
      Sched_Recv_Return(INLOOP);\
    } else\
      ret = 1;\
  }\
}
```

`Sched_Recv_Return()` The macro `Sched_Recv_Return`(l) is to return from `sched_recv()`. The macro is invoked in `Sched_Recv_Check()` used in the `sched_recv()`'s loop ($l \neq 0$), or before the loop ($l = 0$). The macro is also invoked directly by `sched_recv()` at its very end after the loop with $l = 1$. What the macro does is *write-back* of cached local variables for `S_recvsched_context` elements, `nptotal`, `carryover` and `cptr`. It also write the elements `x`, `y`, `z` and `g` back to the structure if $l \neq 0$ to mean those elements are modified by the loop.

```
#define Sched_Recv_Return(INLOOP) {\
  if (INLOOP) {\
    context->x = Grid_X();  context->y = Grid_Y();  context->z = Grid_Z();\
    context->g = The_Grid();\
  }\
  context->nptotal = nptotal;  context->carryover = carryover;\
  context->cptr = cptr + 1;\
  return;\
}
```

`For_All_Grid_From()` The macro `For_All_Grid_From`(x_0, y_0, z_0) is for a D-dimensional nested loop similar to that constructed by `For_All_Grid(0,0,0,0,0,0,0)` for primary subdomain interior but the starting grid-voxel is at (x_0, y_0, z_0). The macro is expanded as shown below if $D = 3$ to iterate the loop body for (x_0, y_0, z_0), $(x_0{+}1, y_0, z_0)$, ..., $(x_1{-}1, y_0, z_0)$, $(0, y_0{+}1, z_0)$, ..., $(x_1{-}1, y_0{+}1, z_0)$, ..., $(x_1{-}1, y_1{-}1, z_0)$, $(0, 0, z_0{+}1)$, ..., $(x_1{-}1, y_1{-}1, z_1{-}1)$.

$$
\begin{aligned}
&\texttt{for}(z = z_0,\ x_1 = \delta_x(m) + x_1,\ \ y_1 = \delta_y(m) + y_1,\ z_1 = \delta_z(m) + z_1,\\
&\qquad x' = x_0,\ y' = y_0,\ w = \delta_x^{\max}(m) + 4e^g,\ d = \delta_y^{\max}(m) + 4e^g,\\
&\qquad g_z = z_0 \cdot d \cdot w,\ g_y = g_z + y_0 \cdot w,\ g_x = g_y + x_0\,;\\
&\quad z < z_1\,;\ z{+}{+},\ g_z = g_z + d \cdot w,\ g_y = g_z,\ x' = y' = 0)\\
&\ \ \texttt{for}(y = y'\,;\ y < y_1\,;\ y{+}{+},\ g_y = g_y + w,\ g_x = g_y,\ x' = 0)\\
&\quad \texttt{for}(x = x'\,;\ x < x_1\,;\ x{+}{+},\ g_x{+}{+})
\end{aligned}
$$

```
#define For_All_Grid_From(X0, Y0, Z0)\
  For_Z((fag_zidx=(Z0),\
         fag_x1=GridDesc[0].x, fag_y1=GridDesc[0].y, fag_z1=GridDesc[0].z,\
         fag_x0=(X0), fag_y0=(Y0),\
         fag_w=GridDesc[0].w, fag_dw=GridDesc[0].dw,\
         fag_gz=(Z0)*fag_dw, fag_gy=fag_gz+fag_y0*fag_w,\
         fag_gx=fag_gy+fag_x0),\
        (fag_zidx<fag_z1),\
        (fag_zidx++, fag_gz+=fag_dw, fag_gx=fag_gy=fag_gz, fag_x0=fag_y0=0))\
    For_Y((fag_yidx=fag_y0), (fag_yidx<fag_y1),\
          (fag_yidx++, fag_gy+=fag_w, fag_gx=fag_gy, fag_x0=0))\
      for (fag_xidx=fag_x0; fag_xidx<fag_x1; fag_xidx++,fag_gx++)
```

`sched_recv()` The function `sched_recv()`, called solely from `make_recv_list()`, scans per-grid histogram to determine the set of grid-voxels to be hosted by a node `nid` $= m_f$ being the f-th member of the local node's primary family, whose expected number of accommodating primary (`tag` $= 0$) or secondary (`tag` $= NS$) particles is determined by the argumetns `currmode`, `reb`, `get` and `stay`. The scanning and assignment context is kept in the `S_recvsched_context` structure argument `context` whose elements and their definitions were given in §4.10.19.

```
static void
sched_recv(const int currmode, const int reb, const int get, const int stay,
           const int nid, const int tag, struct S_recvsched_context *context) {
  const int x0=context->x, y0=context->y, z0=context->z, g=context->g;
  const int ovflimit=gridOverflowLimit;
```

```
dint nptotal=context->nptotal;
dint nplimit=context->nplimit;
dint carryover=context->carryover;
dint **npg=NOfPGridTotal[0];
struct S_commlist *cptr=context->cptr;
const int ns=nOfSpecies;
int s, npt=carryover, ret=0;
Decl_For_All_Grid();
int fag_x0, fag_y0, fag_z0;
```

First, the function calculate $\mathcal{P}_\Lambda(f) = \mathcal{P}_\Lambda(f-1) + Q^n_{m_f}$ depending on the `currmode`, `reb` and `tag` arguments which determine how $Q^n_{m_f}$ for the beginning of the next step is calculated from the argument `get`, `stay` and `NOfPrimaries`[][][] as follows.

- If we have normal accommodation and helpand-helper reconfiguration is taking place, `get` $= Q^{\mathrm{get}}_{m_f}$ has the expected number to be accommodated by the helper node m_f, i.e., $Q^n_{m_f}$.
- Otherwise, `get` is calcluated based on `stay` which has $Q^n_{m_f}$ at the end of the last step. Therefore, the next step's $Q^n_{m_f}$ is the sum of `get` and `stay`. Note that `stay` is correctly set by `schedule_particle_exchange()` in non-position-aware particle transfer if we had anywhere accommodation, by `rebalance1()` for the local node n, or by `count_stay()` if we had normal accommodation and helpand-helper configuration is stable.

```
if (!Mode_Acc(currmode) && reb && tag)
  nplimit += get;
else
  nplimit += get + stay;
```

Then after writing $\mathcal{P}_\Lambda(f)$ back to `context`, we examine if $\mathcal{P}_\Sigma(g') - q_{co}(f) \geq \mathcal{P}_\Lambda(f)$ where $g' = g-1$ if $q_{co}(f) = 0$ or $g' = g$ otherwise to let the left-hand side of the inequality mean the number of particles we have already assigned to nodes prceeding m_f. If this ineqaulity holds to mean that we don't have any particles to assign m_f, we simply return from this function without adding `S_commlist` record[70].

Now we have some particles to assign to m_f and thus set `S_commlist` record elements, `rid` to m_f, `tag` to that given as the argument, `count` to 0 to indicate the record is not for hot-spot so far, and `sid` to the hot-spot ordinal kept in `hs` of `context`[71]. Then we invoke `Sched_Recv_Check()` to assign particles carried over to m_f if any. That is, if $q_{co}(f) = 0$, we know $\mathcal{P}_\Sigma(g-1) < \mathcal{P}_\Lambda(f)$ and thus the macro does nothing. Otherwise, some particles are assigned to m_f and it could make the direct return from the macro due to too heavy population to assign particles carried over. If this consecutive carry-over occurs, the macro assigns an amount of particles to m_f knowing that we have already assigned $\mathcal{P}_\Sigma(g) - q_{co}(f)$ to m_i for $i < f$, because we initialize `npt` to have $q_{co}(f)$ in the declarative part.

Otherwise, i.e., all carry-over particles have assigned to m_f, we let `count` element of the `S_commlist` record be the number of them and its `sid` element be -1 to indicate the

[70] If the local node's primary subdomain has no particles, this inequality holds at the first call of `sched_recv()` with $g = 0$ and $f = 0$, because $\mathcal{P}_\Sigma(g-1) = q_{co}(f) = \mathcal{P}_\Lambda(f) = 0$. Therefore, the caller `make_recv_list()` of `sched_recv()` will have no `S_commlist` records in primary receiving block in this case as discussed in §4.10.19.

[71] The element `sid` is meaningful only for hot-spot records but we let it have some specific value to avoid to leave it undefined.

record is the terminal, and increment `hs` element in `context` for the next hot-spot. We also have to take care that the possibility that assigning the carry-over particles to m_f made $\mathcal{P}_\Sigma(g) \geq \mathcal{P}_\Lambda(f)$, i.e., it made m_f unable to have more particles. If not the case, we add a new `S_commlist` record and set its `rid`, `tag`, `count` and `sid` as done above so that m_f has another (likely non-hot-spot) record, after letting the old record's `region` be g because `Sched_Recv_Check()` did not do that.

```
    context->nplimit = nplimit;
    if (nptotal-carryover>=nplimit)  return;
    cptr->rid = nid;  cptr->tag = tag;  cptr->sid = context->hs;
    cptr->count = 0;
    Sched_Recv_Check(0, g);
    if (carryover) {
      cptr->count = carryover;  cptr->sid = -1;  context->hs++;
      if (!ret)  {
        cptr->region = g;  cptr++;
        cptr->rid = nid;  cptr->tag = tag;  cptr->sid = context->hs;
        cptr->count = 0;
      }
    }
```

Now we start scanning grid-voxels by `For_All_Grid_From()` for the subdomain interior (i.e., without exterior). In the loop, at first we check whether $q_{co}(f) > 0$ and if so we skip one iteration so as to visit the grid-voxel next to the hot-spot without visiting it any more. Then, if `Sched_Recv_Check()` for the hot-spot or for the last iteration tells that we have determined the number of particles to be assigned to m_f, we return from the function by `Sched_Recv_Return()`. Otherwise, we calculate $\mathcal{P}_T(g) = \sum_{s=0}^{S-1} \mathcal{P}_T(0, s, g) = \sum_{s=0}^{S-1} \texttt{NOfPGridTotal}[0][s][g]$, let $\mathcal{P}_\Sigma(g) = \mathcal{P}_\Sigma(g-1) + \mathcal{P}_T(g)$, and then invoke `Sched_Recv_Check()` to check the completion of the scanning.

Finally, we invoke `Sched_Recv_Return()` after we finish the last loop iteration for the very last grid-voxel at g visiting which should have made $\mathcal{P}_\Sigma(g) = \mathcal{P}_\Lambda(f)$.

```
    For_All_Grid_From(x0, y0, z0) {
      if (carryover) { carryover = 0;  continue; }
      if (ret) Sched_Recv_Return(1);
      for (s=0,npt=0; s<ns; s++)  npt += npg[s][The_Grid()];
      nptotal += npt;
      Sched_Recv_Check(1, The_Grid());
    }
    Sched_Recv_Return(1);
  }
```

4.10.21 `make_send_sched()`

`make_send_sched()` The function `make_send_sched()`, called solely from `exchange_particles4p()`, scans primary receiving, primary sending, secondary receiving, secondary sending and alternative secondary receiving blocks in `CommList[]` to determine the node to which the local node n send the particles in each grid-voxel and processes all hot-spots after the scan. The function is given arguments `currmode`, helpand-helper reconfiguration indicator `reb`, the parent status code `pcode`, the identifier of the old and (possibly) new helpand `oldp` and `newp`, the pointer to the head of hot-spot sending block `hslist`, an array `nacc[2]` to accummulate the number of primary ($[0] = Q_n^n$) or secondary ($[1] = Q_n^{parent(n)}$) particles to be accommodated

by the local node, and the pointer `nsend` to *return* the number of particles $P_n^{\rm send}$ to be sent from the local node.

```
static void
make_send_sched(const int currmode, const int reb, const int pcode,
                const int oldp, const int newp, struct S_commlist *hslist,
                int *nacc, int *nsend) {
  const int psold = Parent_Old(pcode) ? 1 : 0;
  const int ns2 = nOfSpecies<<1,  nn = nOfNodes;
  int s, ps, n, h, maxhs=-1;
  int nfrom, nto;
  struct S_commlist *rlist[2] = {CommList,SecRList};
  int *rlidx[2] = {RLIndex,SecRLIndex};
```

First, the function performs a few initializations to scan the `CommList[]` records; all elements of `TotalPNext`$[2][S]$ are cleared, `HotSpotTop` is let point the head of `HotSpotList[]`, and it determines the set of neighbor indices whose sub-blocks in primary sending and secondary sending blocks are scanned, $[0, 3^D)$ if we have normal accommodation, or $\{\lfloor 3^D/2 \rfloor\}$ otherwise because only in-family particle transfer will take place.

```
  for (s=0; s<ns2; s++)  TotalPNext[s] = 0;
  HotSpotTop = HotSpotList;
  if (Mode_Acc(currmode)) {
    nfrom = OH_NBR_SELF;  nto = nfrom + 1;
  } else {
    nfrom = 0;  nto = OH_NEIGHBORS;
  }
```

Next, we scan primary sending block in `CommList[]` always and then secondary sending block in `SecRList[]` if the local node has helpand in the last step, i.e., if `Parent_Old(pcode)` is true. For each block, we scan the sub-block for k-th neighbor whose head index is `RLIndex`$[k']$ for primary sending block or `SecRLIndex`$[k']$ for secondary sending block where $k' = (3^D - 1) - k$. Note that we have to *reverse* the neighboring index because `RLIndex`$[k']$ and `SecRLIndex`$[k']$ are corresponds to `SrcNeighbors`$[k']$ and thus to `Neighbors`$[p][(3^D - 1) - k'] =$ `Neighbors`$[p][k]$ for primary sending ($p = 0$) and secondary sending ($p = 1$) blocks. Also note that we visit a neighbor twice or more if it occurred multiple times in `Neighbors`$[p][]$ to examine corresponding exterior, but just once the local node itself or its old helpand with the index $\lfloor 3^D/2 \rfloor$ for the interior of the primary/secondary subdomain without its exterior because it is assured that corresponging exterior does not have any particles. Yet another remark is that we explicitly skip inexistent neighbors such that `Neighbors`$[p][k] < -(N + 1)$ because no records are in `CommList[]` for them while `make_send_sched_body()` needs at least one proper record.

Before scanning each sub-block for neighbor k by `make_send_sched_body()`, we initialize `HotSpot`$[p][k]$ so that it has a dummy `S_hotspot` record obtained from `HotSpotTop` for the queue tail. Then, we call `make_send_sched_body()` to scan the sub-block for the neighbor k and per-grid histogram with the following arguments.

- `ps` $= p$ to indicate the block to be scanned is primary sending block for primary subdomain's neighbor ($p = 0$) or secondary sending block for secondary subdomain's.
- `n` $= k$ being the neighbor index.

- `sdid` = `Neighbors`$[p][k]$ or $-$(`Neighbors`$[p][k]+1$) being the subdomain identifier of the sub-block.
- `self` is true iff $k = \lfloor 3^D/2 \rfloor$ and either $p = 0$ or the local node has same helpand in the last and next step, i.e., `Parent_New_Same(pcode)` is true. This means that `make_send_sched_body()` scans primary receiving or secondary receiving block in which the local node may appear as a receiver.
- `sender` is true to indicate that `make_send_sched_body()` scans primary sending or secondary sending block instead of alternative secondary receiving block.
- `rlist` pointing the first record of the sub-block to be scanned.
- `maxhs` is the pointer to the local variable of the same name to keep the greatest ordinal $h_{\max}$ of the hot-spot encountered in the scan.
- `naccptr` pointing to `nacc`$[p]$, i.e., Q_n^n or $Q_n^{parent(n)}$.
- `nsendptr` is `nsend` argument to point P_n^{send}.

```
for (ps=0; ps<=psold; ps++) {
  const int root = ps ? oldp : myRank;
  for (n=nfrom; n<nto; n++) {
    int nrev = OH_NEIGHBORS - 1 - n;
    int sdid = Neighbors[ps][n];
    const int self = n==OH_NBR_SELF && (ps==0 || Parent_New_Same(pcode));
    struct S_hotspot *hs = HotSpotTop++;
    hs->comm = NULL;  hs->next = NULL;  hs->g = 0;  hs->lev = INT_MAX;
    HotSpot[ps][n].head = HotSpot[ps][n].tail = hs;
    if (sdid<0)  sdid = -(sdid+1);
    if (sdid<nn && (sdid!=root || n==OH_NBR_SELF))
      make_send_sched_body(ps, n, sdid, self, 1, rlist[ps]+rlidx[ps][nrev],
                           &maxhs, nacc+ps, nsend);
  }
}
```

If we have normal accommodation and helpand-helper reconfiguration is taking place to change the local node's helpand, i.e., `Parent_New_Diff(pcode)` is true, we perform one more scan for alternative secondary receiving block by `make_send_sched_body()`, after initializing `HotSpot`$[2][\lfloor 3^D/2 \rfloor]$. The arguments for this call diffeent from the previous ones are as follows.

- `ps` $= 2$ to indicate the block to be scanned is alternative secondary receiving block.
- `n` $= \lfloor 3^D/2 \rfloor$ being the neighbor index for the secondary subdomain.
- `sdid` is the identifier of the newly assigned secondary subdomain.
- `self` is true to mean the local node may appear in alternative secondary receiving block as a receiver.
- `sender` is false to indicate that `make_send_sched_body()` scans alternative secondary receiving block for receiving particles rather than sending.
- `rlist` = `AltSecRList` pointing the head of alternative secondary receiving block.

- `naccptr` pointing to `nacc`[1] to count secondary particles $Q_n^{parent(n)}$.

```
if (!Mode_Acc(currmode) && Parent_New_Diff(pcode)) {
  struct S_hotspot *hs = HotSpotTop++;
  hs->comm = NULL;  hs->next = NULL;  hs->g = 0;  hs->lev = INT_MAX;
  HotSpot[2][OH_NBR_SELF].head = HotSpot[2][OH_NBR_SELF].tail = hs;
  make_send_sched_body(2, OH_NBR_SELF, newp, 1, 0, AltSecRList, &maxhs,
                       nacc+1, nsend);
}
```

Finally, we process hot-spots we encountered in the scan, i.e., those having ordinals not greater than $h_{\max}$. For deadlock-free gather/scatter of hot-spot information, we process hot-spots according to their ordinal from 0 to $h_{\max}$ with implicit synchronization. That is, we call `gather_hspot_recv()` and `scatter_hspot_send()` to gather/scatter information for the h-th hot-spot in the local node's primary subdomain, and `gather_hspot_send()` and `scatter_hspot_recv()` to do that for hot-spots having the ordinal h in other subdomains if any, in a interleaving manner for each ordinal. Therefore, when the local node processes the h-th hot-spot in its primary subdomain, it is assured that other nodes involved in the hot-spot should respond the hot-spot processing in question.

For each ordinal $h \in [0, h_{\max}]$, at first we call `gather_hspot_recv()` to initiate gather operation and to obatain the number of `MPI_Irecv()` requests R_r made in the function, if the head of the hot-spot queue for the primary subdomain `HotSpot`[0][$\lfloor 3^D/2 \rfloor$] has the ordinal h, passing it `currmode` and `reb` argument and the `S_hotspot` structure at the queue head. Then we call `gather_hspot_send()` to respond `gather_hspot_recv()` in other nodes with h, the parent status code `pcode`, R_r to know the first available entry in `Requests`[], the set of neighbor indices to scan, the pointer to `hslist` argument to let it have the next available hot-spot sending block record, and the pointer to the variable to have the number of `MPI_Irecv()` requests R_s made in the function. Then if we called `gather_hspot_recv()`, i.e., if the primary subdomain has the hot-spot of the ordinal h, we call `scatter_hspot_send()` with R_r, `nacc` argument, and the pointer to `hslist` argument. Finally, we call `scatter_hspot_recv()` with h, `pcode`, R_r, R_s, the neighbor index range, and `nacc` and `nsend` arguments, to process the response from other nodes' `scatter_hspot_send()` if any.

```
for (h=0; h<=maxhs; h++) {
  int rreq=0, sreq;
  struct S_hotspot *hs = HotSpot[0][OH_NBR_SELF].head;
  const int self = hs->lev==h;
  if (self)  rreq = gather_hspot_recv(currmode, reb, hs);
  gather_hspot_send(h, pcode, rreq, nfrom, nto, &hslist, &sreq);
  if (self)  scatter_hspot_send(rreq, nacc, &hslist);
  scatter_hspot_recv(h, pcode, rreq, sreq, nfrom, nto, nacc, nsend);
}
}
```

4.10.22 `make_send_sched_body()`

`Grid_Boundary()` Prior to discusing the funcion `make_send_sched_body()`, we show a macro `Grid_Boundary` $(\nu_d, \delta_d(n'), m, d, x_d^l, x_d^u, \Delta_d)$ used solely by the function. The macro gives the d-th dimensional lower (x_d^l) and upper (x_d^u) bound of an exterior or interior of the local node n's primary/secondary subdomain whose d-th dimensional size is $\delta_d(n')$ where $n' = n$ or

$n' = parent(n)$, for the neighbor m whose d-th dimensional process coordinate relative to the subdomain is $\nu_d - 1 \in \{-1, 0, 1\}$. Note that the upper bound x_d^u is relative to the subdomain's upper bound $\delta_d(n')$. The macro also gives the d-th dimensional offset Δ_d from a grid-voxel in the local node's subdomain to that in m's subdomain.

The lower and upper bounds x_d^l and $x_u^l + \delta_d(n')$ are given as follows.

$$(x_d^l,\ x_d^u + \delta_d(n')) = \begin{cases} (-e^g,\ 0) & \nu_d - 1 = -1 \\ (0,\ \delta_d(n')) & \nu_d - 1 = 0 \\ (\delta_d(n'),\ \delta_d(n') + e^g) & \nu_d - 1 = 1 \end{cases}$$

On the other hand, d-th dimensional coordinate value 0 for the subdomain n' corresponds to $\delta_d(m)$ for the lower d-th dimensional neighbor m, and $\delta_d(n')$ for n' to 0 for upper neighbor m. Therefore, the offset (difference) Δ_d from the grid-voxel at $x(n')$ for n' to that $x(m)$ for m is defined as follows to give us $x(m) = x(n') + \Delta_d$, where $\delta_d(m) = \delta_d^u(m) - \delta_d^l(m) =$ `SubDomains`$[m][d][1] -$ `SubDomains`$[m][d][0]$.

$$\Delta_d = \begin{cases} \delta_d(m) & \nu_d - 1 = -1 \\ 0 & \nu_d - 1 = 0 \\ -\delta_d(n') & \nu_d - 1 = 1 \end{cases}$$

```
#define Grid_Boundary(N, GS, SD, DIM, PL, PU, OFF) {\
  const int e = OH_PGRID_EXT;\
  const int *b = SubDomains[SD][DIM];\
  const int off = If_Dim(DIM, b[OH_UPPER]-b[OH_LOWER], 0);\
  if (N==0)      { PL = -e;    PU = -(GS);  OFF = off; }\
  else if (N==1) { PL = 0;     PU = 0;      OFF = 0; }\
  else           { PL = (GS);  PU = e;      OFF = -(GS); }\
}
```

`make_send_sched_body()` The function `make_send_sched_body()`, called solely from `make_send_sched()` but up to $2 \cdot 3^D + 1$ times, scans a sub-block in primary sending or secondary sending block of `CommList[]` and per-grid histogram of local node n's primary or secondary subdomain, to find the node in which particles in each grid-voxel are accommodated and to enqueue the grid-voxel into hot-spot queue. The arguements given to the function were discussed in §4.10.21.

At first, we determine the exterior or interior of the local node's subdomain to be scanned, $\mathcal{S} = [x^l, x^u) \times [y^l, y^u) \times [z^l, z^u)$ for the neighbor whose index $k = \sum_{d=0}^{D-1} \nu_d 3^d$ is given by the argument `n` invoking macro `Grid_Boundary()` for each dimension $d \in [0, D)$. The macro also gives us the d-th dimensional offset Δ_d from a grid-voxel g in the local node's subdomain to that in the neighbor subdomain g', from which we calculate the offset of grid-voxel index $\Delta = gidx(\Delta_x, \Delta_y, \Delta_z)$ by `Coord_To_Index()` to have $g' = g + \Delta$.

```
static void
make_send_sched_body(const int psor2, const int n, const int sdid,
                     const int self, const int sender,
                     struct S_commlist *rlist, int *maxhs, int *naccptr,
                     int *nsendptr) {
  const int me=myRank, nn=nOfNodes, ns=nOfSpecies;
  const int ps = psor2==2 ? 1 : psor2;
  const int nsor0 = ps ? ns : 0;
```

```
const int nx = n % 3, ny = n/3 % 3, nz = n/9;
int xl, xu, yl, yu, zl, zu, xoff, yoff, zoff, ngoff;
int rlg = rlist->region;
int nacc = *naccptr, nsend = *nsendptr;
Decl_For_All_Grid();

Grid_Boundary(nx, GridDesc[psor2].x, sdid, OH_DIM_X, xl, xu, xoff);
Grid_Boundary(ny, GridDesc[psor2].y, sdid, OH_DIM_Y, yl, yu, yoff);
Grid_Boundary(nz, GridDesc[psor2].z, sdid, OH_DIM_Z, zl, zu, zoff);
ngoff = Coord_To_Index(xoff, yoff, zoff, GridDesc[psor2].w,
                       GridDesc[psor2].dw);
```

Now we scan each grid-voxel at g in $\mathcal{S}$ by `For_All_Grid()`. At first we skip `S_commlist` records until we find the record having $g(r)$ in its `region` element where r is its `rid` element such that $g(r) \geq g' = g + \Delta$ to mean the particles in the grid-voxel will be accommodated by the node r. Then if $g(r) = g$ and the `count` element of the record $q_{hot}(r) > 0$ to mean the grid-voxel is a hot-spot and the local node is involved in it, we enqueue the `S_hotspot` record for it at the tail of `HotSpot`$[p][k]$ where p is the `psor2` argument, obtaining a new dummy record from `HotSpotTop` to copy the tail record into it. Then we let the elemets of the old tail record be as follows.

- `g` $= g$ to remember the grid-voxel.
- `n` be the number of the series of hot-spot records in the `CommList`[] for g' and thus the number of receivers of the hot-spot particles.
- `lev` be the hot-spot ordinal h recorded in `sid` in the `S_commlist` record. In addition, if $h > h_{\max}$ where $h_{\max}$ is pointed by `maxhs` argument, we let $h_{\max} = h$.
- `self` be true iff the local node appears as a receiver in the series of hot-spot records and the `self` argument is true to mean we are scanning the interior of primary or secondary subdomain of the local node. Note that the local node can appear as a receiver of a hot-spot in its exterior with transitional helpand-helper reconfiguration. We have to distinguish the occurrences in interior and exterior because the former is to *receive* particles from the latter. In addition, if we are scanning interior but the local nodes does not appear in the hot-spot records as a receiver, we let $\mathcal{P}_O(p', s, g) =$ `NOfPGridOut`$[p'][s][g] = 0$ where $p' = 1$ if $p = 2$ or $p' = p$ otherwise, for all species $s \in [0, S)$ to mean that the local node will not have any particles in the grid-voxel.
- `next` be the pointer to the newly acquired tail record.
- `comm` be the pointer to the `S_commlist` record for the hot-spot. This element is meaningful only for hot-spots in the interior of the local node's primary subdomain and is referred to by `scatter_hspot_send()`. Then afterward, this element will be let point to a `S_commlist` record in hot-spot sending block by `gather_hspot_send_body()` or `scatter_hspot_send()`.

Note that we skip all hot-spot records and visit the record following the tail hot-spot record having `sid` $= -1$ for the next grid-voxel.

```
For_All_Grid(psor2, xl, yl, zl, xu, yu, zu) {
  const int g = The_Grid();
  const int ng = g + ngoff;
```

```
while (rlg<ng)  rlg = (++rlist)->region;
if (rlg==ng && rlist->count) {
  struct S_hotspot *hs = HotSpot[psor2][n].tail;
  struct S_hotspot *hst = HotSpot[psor2][n].tail = HotSpotTop++;
  struct S_commlist *rl = rlist;
  int involved = rlist->rid==me, lev, s;
  *hst = *hs;
  hs->g = g;  hs->next = hst;  lev = hs->lev = rlist->sid;
  hs->comm = rlist;
  for (rlist++; rlist->sid>=0; rlist++)
    involved = involved || rlist->rid==me;
  involved = involved || rlist->rid==me;  rlist++;
                /* involved = involved || (rlist++)->rid==me
                   doesn't work if involved has been true */
  hs->n = rlist - rl;  rlg = rlist->region;
  hs->self = self && involved;
  if (self && !involved)
    for (s=0; s<ns; s++)  NOfPGridOut[ps][s][g] = 0;
  if (lev>*maxhs)  *maxhs = lev;
```

Otherwise, i.e., the `S_commlist` record is not for a hot-spot, we examine if its `rid` element r is equal to n. If so and we are scanning the interior of n's primary/secondary subdomain, i.e., `self` argument is true, we let $\mathcal{P}_O(p', s, g) = \mathcal{P}_T(p', s, g)$, or in other words `NOfPGridOut`$[p'][s][g]$ = `NOfPGridTotal`$[p'][s][g]$, and add $\mathcal{P}_T(p', s, g)$ to Q_n^n or $Q_n^{parent(n)}$[72] and to `TotalPNext`$[p'][s]$, for each species $s \in [0, S)$ because the local node will host the grid-voxel as a whole. The reason why we must check `self` has been discussed above for the hot-spot case. In addition, if we are scanning primary sending or secondary sending block, i.e., `sender` argument is true, we let `NOfPGrid`$[p'][s][g] = 0$ changing its role to mean no particles in the grid-voxel will be sent to other nodes. Note that if `sender` is false to mean we are scanning alternative secondary receiving block, we cannot do it because we have already visited the grid-voxel in the scan of secondary receiving block being a sub-block of secondary sending one to send all particles in it to other nodes in the new family of the old helpand of the local node.

If we are scanning interior but $r \neq n$, on the other hand, we let `NOfPGridOut`$[p'][s][g] = 0$ because the grid-voxel will be empty. On the other hand further, if $r \neq n$ or we are scanning exterior, and `sender` is true as discussed above, we add `NOfPGrid`$[p'][s][g]$ to P_n^{send}[73] and `NOfSend`$[p''][s][r]$, where $p'' = 0$ if the `tag` element t of the record is 0 to mean the particles are primary for r or $p'' = 1$ if $t = NS$ to mean secondary, for each $s \in [0, S)$ because all particles will be sent to r. Note that the one-dimensional index of `NOfSend`$[p''][s][r]$ is obtained by $t + sN + r$, and thus we also let `NOfPGrid`$[p'][s][g] = t + sN + r + 1$ changing its role so that we can revisit `NOfSend`$[p''][s][r]$ easily when we find a primary ($p' = 0$) or secondary ($p' = 1$) particle of species s in the grid-voxel g in `move_to_sendbuf_sec4p()` or `move_and_sort_secondary()`.

```
} else {
  const int rid = rlist->rid;
  int s;
  if (rid==me && self) {
    for (s=0; s<ns; s++) {
      int naccinc = NOfPGridOut[ps][s][g] = NOfPGridTotal[ps][s][g];
```

[72] Not directly to one of them but to a local variable `nacc` *caching* it pointed by `naccptr` argument.
[73] Not directly to it but to a local variable `nsend` *caching* it pointed by `nsendptr` argument.

```
            nacc += naccinc;  TotalPNext[nsor0+s] += naccinc;
            if (sender)  NOfPGrid[ps][s][g] = 0;
          }
        } else {
          if (self)
            for (s=0; s<ns; s++)  NOfPGridOut[ps][s][g] = 0;
          if (sender) {
            int nofsidx = rlist->tag + rid;               /* [ps][0][rid] */
            for (s=0; s<ns; s++,nofsidx+=nn) {
              int nsendinc = NOfPGrid[ps][s][g];
              nsend += nsendinc;  NOfSend[nofsidx] += nsendinc;
              NOfPGrid[ps][s][g] = nofsidx + 1;
            }
          }
        }
      }
    }
```

Finally we return to the caller after writing the cached Q_n^n or $Q_n^{parent(n)}$ and $P_n^{\rm send}$ back to the originals in the grand-caller `exchange_particles4p()` through the pointer arguments `naccptr` and `nsendptr`.

```
    *naccptr = nacc; *nsendptr = nsend;
  }
```

4.10.23 `gather_hspot_recv()`

`Is_Boundary()` Prior to discuss the funcion `gather_hspot_recv()`, we show a macro `Is_Boundary`(x_d, $\delta_d(n)$) to examine if grid-voxel in the interior of the local node n's primary subdomian having d-th dimensinal coordinate x_d is in a d-th dimensional boundary plane of e^g thick. That is, the macro is expanded to the following value b_d.

$$b_d = \begin{cases} -1 & x_d < e^g \\ 0 & e^g \le x_d < \delta_d(n) - e^g \\ 1 & \delta_d(n) - e^g \le x_d \end{cases}$$

Note that we don't check if the grid-voxel in the interior because we know it is definitely so.

```
  #define Is_Boundary(P, B)  (P<OH_PGRID_EXT ? -1 :\
                              (P>=B-OH_PGRID_EXT ? 1 : 0))
```

`gather_hspot_recv()` The function `gather_hspot_recv()`, called solely from `make_send_sched()` but as many times as the number of hot-spots in the local node's primary subdomain, initiates the gather reception by `MPI_Irecv()` for a hot-spot h pointed by the argument `hs` at the head of the queue `HotSpot`$[0][\lfloor 3^D/2 \rfloor]$ for the local node n's primary subdomain. The function posts `MPI_Irecv()` for all nodes which can have the hot-spot in their primary or secondary subdomains. More specifically, the target nodes are n's helpers and, if the hot-spot is in a boundary plane of its primary subdomain, the neighbors sharing the plane as the exterior of their primary subdomain and their helpers.

```
static int
gather_hspot_recv(const int currmode, const int reb,
                  const struct S_hotspot *hs) {
  const int me=myRank, ns=nOfSpecies, nn=nOfNodes;
  const int g = hs->g,  psold = Mode_PS(currmode) || Mode_Acc(currmode);
  int rreq=0, nbx, nby, nbz, nx, ny, nz;
  const struct S_node *nodes = reb ? NodesNext : Nodes;
  MPI_Request *reqs=Requests;
```

At first we find neighbors involved in the hot-spot h at g. If we have normal accommodation, we calculate $(x, y, z) = gidx^{-1}(g)$ by `Index_To_Coord()` and give each coordinate value to `Is_Boundary()` to have $\beta_d \in \{-1, 0, 1\}$ for each $d \in [0, D)$, which means a neighbor having index $k = \sum_{d=0}^{D-1} \nu_d 3^d$ is involved iff $\nu_d - 1 = \beta_d$ or $\nu_d - 1 = 0$ for all $d \in [0, D)$. Therefore, we have $2^3 = 8$ neighbors if h is at a vertex, $2^2 = 4$ if on an edge, $2^1 = 2$ if in a plane, or $2^0 = 1$ if in other inside region, including n itself.

Otherwise, i.e., we have anywhere accommodation, the nodes involved in the hot-spot are only the family member of the local node and thus we make $\beta_d = 0$ for all $d \in [0, D)$.

```
  if (Mode_Acc(currmode))
    nbx = nby = nbz = 0;
  else {
    int x, y, z;
    Index_To_Coord(g, x, y, z, GridDesc[0].w, GridDesc[0].dw);
    nbx = Is_Boundary(x, GridDesc[0].x);
    nby = If_Dim(OH_DIM_Y, Is_Boundary(y, GridDesc[0].y), 0);
    nbz = If_Dim(OH_DIM_Z, Is_Boundary(z, GridDesc[0].z), 0);
  }
```

Then we scan each neighbor $m =$ `DstNeighbors`$[k]$ or $m = -($`DstNeighbors`$[k] + 1)$ whose index k matches the criteria above but excluding n itself to post `MPI_Irecv()` for receiving the number of particles of all S species in h accommodated by the neighbor into `HSRecv`$[k][m][]$. Note that we don't exclude the second or subsequent occurence of a node m, but their report should be recieved individually in `HSRecv`$[][][]$. Though it looks funny that the node m has multiple hot-spots correspongind to a particular hot-spot in the local node's primary subdomain, it may occur because these hot-spots are at different grid-positions in the exterior of m, e.g., in west and east sending planes if m is east and west neighbor of the local node at the same time. Therefore, we gives k to `MPI_Irecv()` as its `tag` argument for distinguishment.

On the other hand, if $k = \lfloor 3^D/2 \rfloor$, we copy `NOfPGrid`$[0][s][g]$ into `HSRecv`$[3^D/2][n][s]$ for each $s \in [0, S)$ to let n *receive* what it would receive if m were not n.

Then for each neighbor m above but including the local node n itself with $k = \lfloor 3^D/2 \rfloor$, we scan its helpers in $H(m)$ listed in $NN[m]$`.child` to post `MPI_Irecv()` for each $m' \in H(m)$ to receive its report into `HSRecv`$[k][m][]$ again, if in the last step we *were* in secondary mode and thus nodes have *old* helpers or we had anywhere accommodation and thus nodes have *new* helpers which are made possible to have hot-spot by the first-phase non-position-aware particle transfer.

Note that we refer to `NodesNext`$[]$ if helpand-helper reconfiguration is taking place indicated by `reb` $\neq 0$, or `Nodes`$[]$ otherwise, for the scanning helpers because the hot-spot is hosted by *old* helpers. Also note that we scan helpers of a neighbor m multiple times if m appears multiple times, but helpers of the local node n itself is scanned only once with

the neighbor index $k = \lfloor 3^D/2 \rfloor$ because they cannot have hot-spots in the exterior of their secondary subdomains. In addition, we gives k to `MPI_Irecv()` as its `tag` argument again to receive the particle amount from a helper of a neighbor which appear multiple times. Since a node m may appear as a neighbor and also as a helper of other neighbor, giving a tag k for a neighbor and its helpers looks to cause some confusion. However, the node m cannot be the k-th neighbor and a helper of the k-th neighbor at the same time, using tag k is sufficient to make the pair (k, m) and thus the receiving buffer `HSRecv`$[k][m][]$ unique in the calls of `MPI_Irecv()`.

Finally, we return to the caller reporting the number of consumed entries in `Requests`[], being $2^{D-1}N$ at most, as the return value.

```
    for (nz=-1; nz<2; nz++) {
      if (nz && nz!=nbz)  continue;
      for (ny=-1; ny<2; ny++) {
        if (ny && ny!=nby)  continue;
        for (nx=-1; nx<2; nx++) {
          const int nbr = (nx+1)+3*((ny+1)+3*(nz+1));
          int nid = DstNeighbors[nbr];
          struct S_node *ch;
          if (nx && nx!=nbx)  continue;
          if (nid<0)  nid = -(nid+1);
          if (nid>=nn)  continue;
          if (nid!=me)
            MPI_Irecv(HSRecv[nbr]+nid*ns, ns, MPI_INT, nid, nbr, MCW,
                      reqs+rreq++);
          if (nbr==OH_NBR_SELF) {
            int s,  *hsr = HSRecv[OH_NBR_SELF] + me*ns;
            for (s=0; s<ns; s++)  hsr[s] = NOfPGrid[0][s][g];
          }
          if (psold && (nid!=me || nbr==OH_NBR_SELF)) {
            for (ch=nodes[nid].child; ch; ch=ch->sibling) {
              const int chid = ch->id;
              MPI_Irecv(HSRecv[nbr]+chid*ns, ns, MPI_INT, chid, nbr, MCW,
                        reqs+rreq++);
            }
          }
        }
      }
    }
    return(rreq);
  }
```

4.10.24 `gather_hspot_send()`

`gather_hspot_send()` The function `gather_hspot_send()`, called solely from `make_send_sched()` but as many times as the maximum ordinal of hot-spots the local node is involved in, scans all hot-spot queued in `HotSpot`[][] having the ordinal h given as the argument `hxidx`. The other arguments it receives are `pcode` for the parent status, `rreq` $= R_r$ being the number of `MPI_Irecv()` posted in `gather_hspot_recv()` and meaning that `Requests`$[R_r]$ is the first available entry for the use in this function, `nfrom` and `nto` to specify the set of neighbor indices $[0, 3^D)$ or $\{\lfloor 3^D/2 \rfloor\}$ to be scanned, `hslist` is the head of hot-spot sending block in which hot-spot sending schedules are built, and `sreqptr` pointing R_s being the number of `MPI_Irecv()` posted in this function.

In this function we call `gather_hspot_send_body()` for each neighbor `Neighbors`$[p][k]$, where index k is in the set specified, of the primary subdomain ($p = 0$) always and secondary subdomain ($p = 1$) if exists, i.e., `Parent_Old()` of `pcode` is true, to send the number of particles for all species in the hot-spot if any and to initiate the reception of the sending schedule in return to it. The function may also initiate another reception to know the number of particles which the local node will accommodate when the neighbor is its helpand. We also call the function once more if helpand-helper reconfiguration is taking place to gives the local node a new secondary subdomain different from the old one, i.e., `Parent_New_Diff()` of `pcode` is true, only for the reception of the number of accommodating particles.

The arguments given to `gather_hspot_send_body()`, other than the arugments of this functin itself, are as follows; `ps` $= p \in [0, 2]$ to refer to `HotSpot`$[p][]$, `n` and `dst` being the index and identifier of the neighbor, and `sender` is false for the new helpand and true for others.

```
static void
gather_hspot_send(const int hsidx, const int pcode, const int rreq,
                  const int nfrom, const int nto, struct S_commlist **hslist,
                  int *sreqptr) {
  const int psold=Parent_Old(pcode) ? 1 : 0;
  MPI_Request *reqs = Requests + rreq;
  int ps, n;

  *sreqptr = 0;
  for (ps=0; ps<=psold; ps++) {
    for (n=nfrom; n<nto; n++)
      gather_hspot_send_body(hsidx, ps, n, Neighbors[ps][n], 1, hslist, reqs,
                             sreqptr);
  }
  if (Parent_New_Diff(pcode))
    gather_hspot_send_body(hsidx, 2, OH_NBR_SELF, RegionId[1], 0, hslist, reqs,
                           sreqptr);
}
```

4.10.25 `gather_hspot_send_body()`

`gather_hspot_send_body()`

The function `gather_hspot_send_body()`, called solely from `gather_hspot_send()` but up to $2 \cdot 3^D + 1$ times, examines the head record $\mathcal{H}$ of the hot-spot queue `HotSpot`$[p][k]$ has the hot-spot whose ordinal is h, where $p =$ `psor2`, $k =$ `n` and $h =$ `hsidx` given by the arguments. Then if the ordinals match, it processes the hot-spot with other arguments; `dst` is the subdomain identifier m or $-(m+1)$ of the neighbor k; `sender` is true iff the local node may send the particles in the hot-spot to m or its helpers; `hslist` is the double pointer to `CommList`$[c]$ from which the sending schedule of the hot-spot is bulit; and `sreqptr` pointing the variable R_s in which we accumulate the number of `MPI_Irecv()` in `gather_hspot_send()`.

At first we examine if h is equal to $\mathcal{H}$`.lev`, and return to the caller without doing anything if not, because it is not the turn for $\mathcal{H}$ or the queue is empty and thus $\mathcal{H}$ has `INT_MAX` ordinal. We also examine if the neighbor subdomain is the local node's primary one, i.e., $p = 0$ and $k = \lfloor 3^D/2 \rfloor$, and return to the caller again if so. Note that the primary subdomain may appear as a neighbor subdomain but the queue for it should always be empty. Also note that the queue should always be empty too for inexsistent neighbors and thus we don't check if $m = -(N + 1)$.

Then if $\mathcal{H}$.self is true to mean m is the local node n's helpand and n is a receiver of the hot-spot particles, we post MPI_Irecv() to receive the amounts of hot-spot particles to accommodate into HSRecvFromParent[s] for all species $s \in [0, S)$ from the helpand m. Note that we give a tag $2 \cdot 3^D$ to MPI_Irecv() to distinguish it from those in gather_hspot_recv() less than 3^D and from that in this function in $[3^D, 2{\cdot}3^D)$ to receive hot-spot sending schedules from a neighbor or that of the helpand.

Then, after initializing $\mathcal{H}$.comm to be NULL, we send the amount of particles in the hot-spot to m if sender is true. That is, we copy $\mathcal{P}_L(p', s, g)$ = NOfPGrid[p'][s][g] to HSSend[s], where $p' = 1$ if $p = 2$ or $p' = p$ otherwise and $g = \mathcal{H}$.g being the grid-position of the hot-spot, for each $s \in [0, S)$, and send the HSSend[] to m by MPI_Send() with a tag $k' = 3^D - 1 - k$ so that m's MPI_Irecv() can distinguish each of multiple occurrence of n in m's neighbors or their helpers. Note that the k-th neighbor m should have the neighbor index $(3^D - 1 - k)$ for n or its helpand.

Then if the hot-spot has one or more particles, we post MPI_Irecv() to receive ρS records of T_Commlist for hot-spot sending schedule into CommList[c] and its successors from m, where ρ is the number of receivers of the hot-spot recorded in $\mathcal{H}$.n. The tag for this MPI_Irecv() is $3^D + k'$ to correspond to k' for the MPI_Send() but also to distinguish it from MPI_Irecv() from m posted by the local node in gather_hspot_recv(). Then the $\mathcal{H}$.comm is let to point CommList[c] and (conceptually) c is incremented by ρS for the next available S_commlist record. Note that the number of receiving records can be smaller than ρS but we simply waste those unused records.

Finally, we return to the caller, after letting hslist arguments have the double pointer to CommList[c] with (possibly) updated c, and reporting the caller the value of R_s which has been incremented by 0, 1 or 2, through sreqptr argument.

```
static void
gather_hspot_send_body(const int hsidx, const int psor2, const int n, int dst,
                       const int sender, struct S_commlist **hslist,
                       MPI_Request *reqs, int *sreqptr) {
  struct S_hotspot *hs = HotSpot[psor2][n].head;
  const int ns = nOfSpecies,  g = hs->g,  nrec = hs->n * ns;
  const int ps = psor2==2 ? 1 : psor2;
  const int nrev = OH_NEIGHBORS - 1 - n;
  struct S_commlist *hsl = *hslist;
  int sreq = *sreqptr;
  int np, s;

  if (hs->lev!=hsidx || (ps==0 && n==OH_NBR_SELF))  return;
  if (dst<0)  dst = -(dst+1);
  if (hs->self)
    MPI_Irecv(HSRecvFromParent, ns, MPI_INT, dst, OH_NEIGHBORS<<1, MCW,
              reqs+sreq++);
  hs->comm = NULL;
  if (sender) {
    for (s=0,np=0; s<ns; s++)  np += (HSSend[s] = NOfPGrid[ps][s][g]);
    MPI_Send(HSSend, ns, MPI_INT, dst, nrev, MCW);
    if (np) {
      MPI_Irecv(hsl, nrec, T_Commlist, dst, OH_NEIGHBORS+nrev, MCW,
                reqs+sreq++);
      hs->comm = hsl;  hsl += nrec;
    }
  }
```

```
  *hslist = hsl;  *sreqptr = sreq;
}
```

4.10.26 scatter_hspot_send()

scatter_hspot_send()

The function `scatter_hspot_send()`, called solely from `make_send_sched()` but as many times as the number of hot-spots in the local node's primary subdomain, completes asyncronous receptions by `MPI_Irecv()` posted by `gather_hspot_recv()` for the hot-spot head record $\mathcal{H}$ of the queue `HotSpot`$[0][\lfloor 3^D/2 \rfloor]$ for the local node n's primary subdomain, and then build the hot-spot receiving and sending schedules to send them to nodes involved in the hot-spot. The function receives the number of posted receptions `nreq` $= R_r$, the pointer `nacc` to Q_n^n, and the double pointer `hslist` to the first available `S_commlist` record at `CommList`$[c] = C_{\rm send}(0)$ in hot-spot sending block as its arguments.

```
static void
scatter_hspot_send(const int rreq, int *nacc, struct S_commlist **hslist) {
  struct S_hotspot *hs = HotSpot[0][OH_NBR_SELF].head;
  const struct S_commlist *rl = hs->comm;
  struct S_commlist *slhead = *hslist,  *sl;
  const int ns=nOfSpecies, nn=nOfNodes, me=myRank, g=hs->g, nr=hs->n;
  int r, ri, s, sinc, *hsr, *nofr;
  dint hst;
```

At first we confirm the completion of all R_r asynchronous receptions recorded in `Requests`[] by `MPI_Waitall()` to obtain their statuses in `Statuses`[], if $R_r > 0$. Then we sum up `NOfPGridTotal`$[0][s][g] = \mathcal{P}_T(0,s,g) = Q_{hot}(s)$ for all $s \in [0,S)$ to have the grand total population Q^Σ_{hot} in the hot-spot at $g = \mathcal{H}$`.g`.

Next we scan hot-spot records $C_{\rm recv}(r)$ in primary receiving block from $\mathcal{H}$`.comm` for each $r \in [0,\rho)$ to build the receiving schedule $q^{\rm recv}_{hot}(r,s)$ for each $r \in [0,\rho)$ and $s \in [0,S)$, where $\rho = \mathcal{H}$`.n`, based on $C_{\rm recv}(r)$`.count` $= q_{hot}(r)$, $Q_{hot}(s)$ and Q^Σ_{hot}. For $r = 0$, we calculate *base* values of $q^{\rm recv}_{hot}(r,s)$ namely $q^{\rm r}_{hot}(r,s) = \lfloor q_{hot}(r) \cdot Q_{hot}(s)/Q^\Sigma_{hot} \rfloor$ so that $q^{\rm recv}_{hot}(r,s)/q_{hot}(r) \approx Q_{hot}(s)/Q^\Sigma_{hot}$ for all r and for each s and thus each m_r accommodates particles of species s with approximately consistent *density*. However, since $q'_{hot}(r) = \sum_{s=0}^{S-1} q^{\rm r}_{hot}(r,s)$ can be less than $q_{hot}(r)$ at most by $S-1$, we need to make adjustment by letting $q^{\rm recv}_{hot}(r,s) = q^{\rm r}_{hot}(r,s) + q^{\Delta{\rm r}}_{hot}(r,s)$ for each s with $q^{\Delta{\rm r}}_{hot}(r,s) \in [0,S)$ such that $\sum_{s=0}^{S-1} q^{\Delta{\rm r}}_{hot}(r,s) = q_{hot}(r) - q'_{hot}(r) = q^{\Delta}_{hot}(r)$.

For the adjustment, first we let $Q_{hot}(s) \leftarrow Q_{hot}(s) - q^{\rm r}_{hot}(r,s)$ and $q^{\Delta{\rm r}}_{hot}(r,s) = 0$ for all $s \in [0,S)$. Then we scan s from 0 to find $Q_{hot}(s) > 0$ and let $q^{\Delta{\rm r}}_{hot}(r,s) \leftarrow q^{\Delta{\rm r}}_{hot}(r,s) + 1$ and $Q_{hot}(s) \leftarrow Q_{hot}(s) - 1$ each time of finding, until we have $\sum_{s=0}^{S-1} q^{\Delta{\rm r}}_{hot}(r,s) = q^{\Delta}_{hot}(r)$. If we don't reach this goal when we have $s = S-1$, we go back to $s = 0$ and repeat the scan cyclicly.

Now we have $q^{\rm recv}_{hot}(r,s)$ for $r = 0$ and for all $s \in [0,S)$ in `NOfRecv`$[r][s]$ and then send them to m_r by `MPI_Send()` giving it the tag $2 \cdot 3^D$ to distinguish it from those for gathering $[0,3^D)$ and those for sending schedule we will send later $[3^D, 2{\cdot}3^D)$, if $m_r \neq n$. Otherwise, i.e., $m_r = n$, we copy them into `NOfPGridOut`$[0][s][g] = \mathcal{P}_O(0,s,g)$ and add it to `TotalPNext`$[s]$ as if n *received* it. In this case we also add $q_{hot}(r)$ to Q^n_n through the pointer argument `nacc`.

Then we do above for $r = 1$ but this time we start the scan of $Q_{hot}(s)$ for the adjustment from the next s of what we visited at last for $r = 0$. We repeat this for succeeding r to have $q^{\rm recv}_{hot}(r,s)$ for all $r \in [0,\rho)$.

```
if (rreq)  MPI_Waitall(rreq, Requests, Statuses);
for (s=0,hst=0; s<ns; s++)  hst += NOfPGridTotal[0][s][g];
for (ri=0,sinc=0,nofr=NOfRecv; ri<nr; ri++,nofr+=ns) {
  const int count = rl[ri].count,  rid = rl[ri].rid;
  int nget = 0;
  for (s=0; s<ns; s++) {
    const int ng = nofr[s] = (NOfPGridTotal[0][s][g]*count) / hst;
    nget+= ng;  NOfPGridTotal[0][s][g] -= ng;
  }
  for (nget=count-nget; nget>0;) {
    if (NOfPGridTotal[0][sinc][g]) {
      nofr[sinc]++;  NOfPGridTotal[0][sinc][g]--;  nget--;
    }
    if (++sinc>=ns)  sinc = 0;
  }
  hst -= count;
  if (rid==me) {
    for (s=0; s<ns; s++) {
      nget = NOfPGridOut[0][s][g] = nofr[s];
      TotalPNext[s] += nget;
    }
    *nacc += count;
  } else {
    MPI_Send(nofr, ns, MPI_INT, rid, OH_NEIGHBORS<<1, MCW);
  }
}
```

Now we have the receiving schedules $q_{hot}^{\mathrm{recv}}(r,s)$ of each m_r of $r \in [0,\rho)$ and $s \in [0,S)$, and then based on them and $\texttt{HSRecv}[k_i][m_i][s] = q_{hot}^{\mathrm{send}}(k_i,m_i,s)$ for each $i \in [0,R_r]$ we build the sending schedule for m_i as the k_i-th neighbor of the local node n. Note that m_i and k_i are obtained from `MPI_SOURCE` and `MPI_TAG` elements of $\texttt{Statuses}[i]$ while $k_{R_r} = \lfloor 3^D/2 \rfloor$ and $m_{R_r} = n$ for n itself which must have the hot-spot in question. Also note that for m and m' in the local node's family and thus has some r and r' such that $m = m_r$, $m' = m_{r'}$ and $r < r'$, it is assured that $i < i'$ if we have $q_{hot}^{\mathrm{send}}(k_i,m_i,s)$ and $q_{hot}^{\mathrm{send}}(k_{i'},m_{i'},s)$ such that $k_i = k_{i'} = \lfloor 3^D/2 \rfloor$, $m_i = m$ and $m_{i'} = m'$ because of the scanning order in `make_recv_list()` and `gather_hspot_recv()` so that the amount of particles transferred among the family members is kept small.

At first we initialize $\texttt{HSReceiver}[s] = r(s)$ to be 0 to mean we start scan from $q_{hot}^{\mathrm{recv}}(0,s)$ for the assignment to m_0 for each $s \in [0,S)$. Then for each $i \in [0,R_r]$ we do the followings for each $s \in [0,S)$ to determine the set of pairs $\mathcal{S}(i,s) = \{(r_s^0,q_s^0),(r_s^1,q_s^1),\ldots\}$ to mean that the node m_i sends its q_s^j particles of species s to the hot-spot receiver node having ordinal r_s^j, i.e., $C_{\mathrm{recv}}(r_s^j)\texttt{.rid}$. For i and s with $q_{hot}^{\mathrm{send}}(k_i,m_i,s) = 0$, $\mathcal{S}(i,s) = \emptyset$. Otherwise, we have $c(i,s)$ pairs, i.e., $|\mathcal{S}(i,s)| = c(i,s)$ such that as follows.

$$
\begin{aligned}
r_s^0 &= \min\{r \,|\, r \geq r(s),\ q_{hot}^{\mathrm{recv}}(r,s) > 0\} \\
r_s^j &= \min\{r \,|\, r > r_s^{j-1},\ q_{hot}^{\mathrm{recv}}(r,s) > 0\} \\
c(i,s) &= \min\{c \,|\, \sum_{j=0}^{c-1} q_{hot}^{\mathrm{recv}}(r_s^j,s) \geq q_{hot}^{\mathrm{send}}(k_i,m_i,s)\}
\end{aligned}
$$

Then we let q_s^j, $q_{hot}^{\mathrm{recv}}(r_s^j, s)$ and $r(s)$ as follows.

$$q_s^j = \begin{cases} q_{hot}^{\mathrm{recv}}(r_s^j, s) & j < c(i,s) - 1 \\ q_{hot}^{\mathrm{send}}(k_i, m_i, s) - \sum_{j=0}^{c(i,s)-2} q_{hot}^{\mathrm{recv}}(r_s^j, s) & j = c(i,s) - 1 \end{cases}$$

$$q_{hot}^{\mathrm{recv}}(r_s^j, s) \leftarrow \begin{cases} 0 & j < c(i,s) - 1 \\ \sum_{j=0}^{c(i,s)-1} q_{hot}^{\mathrm{recv}}(r_s^j, s) - q_{hot}^{\mathrm{send}}(k_i, m_i, s) & j = c(i,s) - 1 \end{cases}$$

$$= q_{hot}^{\mathrm{recv}}(r_s^j, s) - q_s^j$$

$$r(s) \leftarrow r_s^{c(i,s)-1}$$

Then all pairs $\bigcup_{s=0}^{S-1} \mathcal{S}(i,s)$ are listed from $C_{\mathrm{send}}(0)$ to have $\sum_{s=0}^{S-1} c(i,s)$ records of `S_commlist` so that $C_{\mathrm{send}}(l)$ has the following where $l = l(s,j) = \sum_{t=0}^{s-1} c(i,t) + j$.

$$\begin{aligned} C_{\mathrm{send}}(l)\texttt{.sid} &= c(i,s) \\ C_{\mathrm{send}}(l)\texttt{.rid} &= C_{\mathrm{recv}}(r_s^j)\texttt{.rid} \\ C_{\mathrm{send}}(l)\texttt{.region} &= C_{\mathrm{recv}}(r_s^j)\texttt{.region} = \mathcal{H}\texttt{.g} \\ C_{\mathrm{send}}(l)\texttt{.count} &= q_s^j \\ C_{\mathrm{send}}(l)\texttt{.tag} &= C_{\mathrm{recv}}(r_s^j)\texttt{.tag} + sN = s'N \quad (s' \in [0, 2S)) \end{aligned}$$

Note that $C_{\mathrm{send}}(l)\texttt{.sid} = c(i,s)$ is only for $l = l(s,0)$ and those for $l(s,j)$ $(j > 0)$ are left undefined[74]. Also note that $C_{\mathrm{send}}(l)\texttt{.tag} = sN$ if the receiver m is the local node, or $(s+S)N$ if a helper, and thus $C_{\mathrm{send}}(l)\texttt{.tag} + C_{\mathrm{send}}(l)\texttt{.rid}$ gives one-dimensional index of $[p][s][m]$ of an array of $[2][S][N]$.

Then if there are some $\mathcal{S}(i,s) \neq \emptyset$, i.e., there are some $q_{hot}^{\mathrm{send}}(k_i, m_i, s) > 0$, we send the list to $m_i \neq n$ for $i < R_r$ by `MPI_Send()` with tag $3^D + k_i$ to distinguish it from those for gathering $[0, 3^D)$ and that for the receiving schedule sent earlier in this function $2 \cdot 3^D$. For $i = R_r$ and thus $m_i = n$, on the other hand, we simply let $\mathcal{H}$`.comm` point $C_{\mathrm{send}}(0)$ so that the local node n can refer to the records in `scatter_hspot_recv()`, or let it be `NULL` if $\mathcal{S}(i,s) = \emptyset$ for all $s \in [0, S)$. Then we report the caller the next available record in hot-spot sending block being the pointer to $C_{\mathrm{send}}(\sum_{s=0}^{S-1} c(R_r, s))$ through `hslist` argument.

Note that records sent to other nodes or linked from $\mathcal{H}$`.comm` don't have species and thus it looks impossible to judge whether the records for $\mathcal{S}(i,s)$ for a species s exist or not by scanning the records. However, since a receiver node or the local node m_i knows whether $q_{hot}^{\mathrm{send}}(k_i, m_i, s) = 0$ and thus $\mathcal{S}(i,s) = \emptyset$, the records are properly scanned and processed in the function `scatter_hspot_recv_body()`.

```
for (s=0; s<ns; s++)  HSReceiver[s] = 0;
for (r=0; r<=rreq; r++) {
  const int dst = r==rreq ? me          : Statuses[r].MPI_SOURCE;
  const int nbr = r==rreq ? OH_NBR_SELF : Statuses[r].MPI_TAG;
  struct S_commlist *slsave;
  int tag;
  hsr = HSRecv[nbr] + dst*ns;
  for (s=0,sl=slsave=slhead,tag=0; s<ns; s++,tag+=nn) {
```

[74] They have $C_{\mathrm{recv}}(l)$`.sid` for the hot-spot ordinal but meaningless.

```
        int nput = hsr[s],  nget = 0;
        if (nput==0) continue;
        for (ri=HSReceiver[s],nofr=NOfRecv+ri*ns; ; ri++,nofr+=ns) {
          const int ng = nofr[s], ngetsave = nget;
          if (ng) {
            nget += ng;  *sl = rl[ri];  sl->tag += tag;
            if (nput>nget)  {
              nofr[s] = 0;  (sl++)->count = ng;
            } else {
              nofr[s] -= ((sl++)->count = nput-ngetsave);  HSReceiver[s] = ri;
              break;
            }
          }
        }
        slsave->sid = sl - slsave;  slsave = sl;
      }
      if (r==rreq) {
        hs->comm = sl>slhead ? slhead : NULL;  *hslist = sl;
      } else if (sl>slhead) {
        MPI_Send(slhead, sl-slhead, T_Commlist, dst, OH_NEIGHBORS+nbr, MCW);
      }
    }
  }
```

4.10.27 scatter_hspot_recv()

scatter_hspot_recv()

The function scatter_hspot_recv(), called solely from make_send_sched() but as many times as the maximum ordinal of hot-spots the local node is involved in, scans all hot-spot queued in HotSpot[][] having the ordinal h given as the argument hxidx. The other arguments it receives are pcode for the parent status, rreq $= R_r$ and sreq $= R_s$ being the number of MPI_Irecv() posted in gather_hspot_recv() and gather_hspot_send() to mean the requests of latter are in Requests[i] where $i \in [R_r, R_s)$, nfrom and nto to specify the set of neighbor indices $[0, 3^D)$ or $\{\lfloor 3^D/2 \rfloor\}$ to be scanned, nacc[2] $= \{Q_n^n, Q_n^{parent(n)}\}$, and the pointer nsend to $P_n^{\rm send}$.

In this function, at first we confirm the completion of all R_s asynchronous receptions recorded in Requests[i] by MPI_Waitall() to obtain their statuses in Statuses[i] for $i \in [R_r, R_s)$. Then we call scatter_hspot_recv_body() for each neighbor Neighbors[p][k], where index k is in the set specified, of the primary subdomain ($p = 0$) always and secondary subdomain ($p = 1$) if exists, i.e., Parent_Old() of pcode is true, to examine the receiving and sending schedules for a hot-spot sent from a neighbor. We also call the function once more if helpand-helper reconfiguration is taking place to give the local node a new secondary subdomain different from the old one, i.e., Parent_New_Diff() of pcode is true, only for the receiving schedule.

The arguments given to gather_hspot_send_body(), other than the arugments of this functin itself, are ps $= p \in [0, 2]$ to refer to HotSpot[p][], n being the neighbor index to visit, and nacc $+\{0, 1\}$ being $+0$ if $p = 0$ or $+1$ otherwise to specify Q_n^n or $Q_n^{parent(n)}$ respectively.

```
static int
scatter_hspot_recv(const int hsidx, const int pcode, const int rreq,
                   const int sreq, const int nfrom, const int nto, int *nacc,
                   int *nsend) {
```

```
  const int psold = Parent_Old(pcode) ? 1 : 0;
  int ps, n;
  MPI_Status *st = Statuses + rreq;

  if (sreq>0)  MPI_Waitall(sreq, Requests+rreq, st);
  for (ps=0; ps<=psold; ps++) {
    for (n=nfrom; n<nto; n++) {
      scatter_hspot_recv_body(hsidx, ps, n, nacc+ps, nsend);
    }
  }
  if (Parent_New_Diff(pcode)) {
    scatter_hspot_recv_body(hsidx, 2, OH_NBR_SELF, nacc+1, nsend);
  }
}
```

4.10.28 scatter_hspot_recv_body()

scatter_hspot_recv_body()

The function scatter_hspot_recv_body(), called solely from scatter_hspot_recv() but up to $2 \cdot 3^D + 1$ times, examines if the head record $\mathcal{H}$ of the hot-spot queue HotSpot$[p][k]$ has the hot-spot whose ordinal is h, where $p =$ psor2, $k =$ n and $h =$ hsidx given by the arguments. Then if the ordinals match, it processes the hot-spot receiving and sending schedule sent from the k-th neighbor of the local node n's primary ($p = 0$) or secondary($p \neq 0$) subdomain including n or its helpand themselves, updating either Q_n^n or $Q_n^{parent(n)}$ pointed by naccptr argument and/or $P_n^{\rm send}$ pointed by nsendptr argument.

```
static void
scatter_hspot_recv_body(const int hsidx, const int psor2, const int n,
                        int *naccptr, int *nsendptr) {
  const int ns=nOfSpecies, me=myRank;
  const int ps = psor2==2 ? 1 : psor2;
  const struct S_hotspot *hs = HotSpot[psor2][n].head;
  struct S_commlist *sl = hs->comm;
  const int g = hs->g,  self = hs->self;
  int nsend = *nsendptr;
  int slidx, s, si;
```

At first we examine if h is equal to $\mathcal{H}$.lev, and return to the caller without doing anything if not, because it is not the turn for $\mathcal{H}$ or the queue is empty and thus $\mathcal{H}$ has INT_MAX ordinal. Otherwise we dequeue $\mathcal{H}$ to let HotSpot$[p][k]$.head points the successor of $\mathcal{H}$.

Then if $\mathcal{H}$.self is true and $p \neq 0$ to mean that the hot-spot is in the local node n's secondary subdomain and n is one of its hosts, we should have received the receiving schedule in HSRecvFromParent$[S]$. Therefore, we copy its element HSRecvFromParent$[s] = q_{hot}^{\rm recv}(n, s)$ into NOfPGridOut$[1][s][g] = \mathcal{P}_O(1, s, g)$ where $g = \mathcal{H}$.g for each $s \in [0, S)$, to fix each number of particles to accommodate for the hot-spot. We also add $q_{hot}^{\rm recv}(n, s)$ to TotalPNext$[1][s]$, and $\sum_{s=0}^{S-1} q_{hot}^{\rm recv}(n, s)$ to $Q_n^{parent(n)}$ through naccptr argument.

```
  if (hs->lev!=hsidx)  return;
  HotSpot[psor2][n].head = hs->next;
  if (self && ps) {
    int nacc=*naccptr;
```

```
  for (s=0; s<ns; s++) {
    const int nget = NOfPGridOut[1][s][g] = HSRecvFromParent[s];
    nacc += nget;  TotalPNext[ns+s] += nget;
  }
  *naccptr = nacc;
}
```

Then we examine $\mathcal{H}$.comm and return to the caller if it is NULL to mean the local node does not have any particles to send in the hot-spot. Otherwise, i.e., if it points the head of record sequence $C_{\rm send}(0)$ = CommList[b], $C_{\rm send}(1)$ = CommList[$b+1$], ..., we have S_commlist records comprising of $\mathcal{S}(s) = \{(r_s^0, q_s^0), (r_s^1, q_s^1), \ldots\}$ for s such that NOfPGrid[p'][s][g] = $\mathcal{P}_L(p', s, g) > 0$ and $C_{\rm send}(l)$.(rid, count) are $(r_s^j, q_s^j) \in \mathcal{S}(s)$, where $p' = 1$ if $p = 2$ or $p' = p$ otherwise, $l = l(s,j) = \sum_{t=0}^{s-1} c(t) + j$ and $c(s) = C_{\rm send}(l(s,0))$.sid $= |\mathcal{S}(s)|$.

For each s, we do nothing if $\mathcal{P}_L(p', s, g) = 0$ and thus there are no records for s, leaving NOfPGrid[p'][s][g] = $\mathcal{P}_L(p', s, g) = 0$ unchanged[75]. Otherwise, i.e., if $\mathcal{P}_L(p', s, g) > 0$ we let NOfPGrid[p'][s][g] be $-(b + l(s,0) + 1)$ so that we can revisit $C_{\rm send}(l(s,0))$ being the head of $\mathcal{S}(s)$ when we find a particle in the grid-voxel g in move_to_sendbuf_sec4p(), move_to_sendbuf_uw4p(), move_to_sendbuf_dw4p() and/or move_and_sort_secondary(). Then we scan all records in (r_s^j, q_s^j) in $\mathcal{S}(s)$ to add $q_s^j = C_{\rm send}(l(s,j))$.count to NOfSend[$p''$][$s$][$r_s^j$] where the one-dimensional base index for [p''][s][0] is given by $C_{\rm send}(l(s,j))$.tag. We also add q_s^j to $P_n^{\rm send}$ because they will be sent.

We also examine if there exists $r_s^j = n$ when $\mathcal{H}$.self is true, and if so we exchange the record $C_{\rm send}(l(s,j))$ and $C_{\rm send}(l(s,c(s)-1))$ so that the record for the lcoal node is at the tail and thus the functions above let the particles for the local node n stay in n after letting other particles be sent. In addition we let $C_{\rm send}(l(s,c(s)-1))$.tag $= -1$ to indicate it is at the tail and is for the number of particles to be accommodated by the local node rather than that for sending.

Finally we update $P_n^{\rm send}$ in the grand-grand-caller exchange_particles4p() through the argument pointer nsendptr.

```
if (!sl)  return;
slidx = -(sl - CommList + 1);
for (s=0,si=0; s<ns; s++) {
  int mysi = -1,  r;
  const int nr = sl[si].sid;
  if (NOfPGrid[ps][s][g]==0)  continue;
  NOfPGrid[ps][s][g] = slidx - si;
  for (r=0; r<nr; r++,si++) {
    const int rid = sl[si].rid,  count = sl[si].count;
    sl[si].sid = nr;
    if (rid==me && self)  mysi = si;
    else {
      NOfSend[sl[si].tag+rid] += count;  nsend += count;
    }
  }
  if (mysi>=0) {
    struct S_commlist sltmp = sl[mysi];
    sl[mysi] = sl[si-1];  sl[si-1] = sltmp;  sl[si-1].tag = -1;
  }
}
```

[75] NOfPGrid[p'][s][g] will not be referred to until transbound4p() finishes.

```
    *nsendptr = nsend;
  }
```

4.10.29 update_descriptors()

update_descriptors()

The function update_descriptors(), called from exchange_particles4p() when we had anywhere accommodation and from make_recv_list() otherwise, reinitializes BorderExc[][1][][].{send, recv} for the old secondary subdomain given through the argument oldp by clear_border_exchange(), and update FieldDesc[].{bc,red}.size[1] for the new secondary subdomain given through the argument m = newp giving FieldTypes[F][7] and SubDomains[m][D][2]. It also calls adjust_field_descriptor() to modify FieldDesc[$F-1$].{bc,red}.size[1] for per-grid histograms giving it ps = 1.

Note that we do above if the old and new parents are different, and call clear_border_exchange() if the old one exists, while other two functions are called if the new one exists.

```
  static void
  update_descriptors(const int oldp, const int newp) {
    int n;

    if (oldp!=newp) {
      if (oldp>=0)  clear_border_exchange();
      if (newp>=0) {
        set_field_descriptors(FieldTypes, SubDomains[newp], 1);
        adjust_field_descriptor(1);
      }
    }
  }
```

4.10.30 update_neighbors()

Neighbor_Grid_Offset()

Prior to discuss the funcion update_neighbors(), we show a macro Neighbor_Grid_Offset $(p, \nu_d - 1, m, d, c)$ which calculates

$$x_d^0(m,n') = \delta_d^l(m) - \delta_d^l(n') = \begin{cases} \delta_d^l(m) - \delta_d^u(m) = -\delta_d(m) & \nu_d = 0 \\ \delta_d^l(n') - \delta_d^l(n') = 0 & \nu_d = 1 \\ \delta_d^u(n') - \delta_d^l(n') = \delta_d(n') & \nu_d = 2 \end{cases}$$

where $n' = \{n, parent(n)\}[p]$ for the local node n, for GridOffset[p][$\sum_{d=0}^{D-1} \nu_d 3^d$] as discussed in §4.9.5. Note that $\delta_d^\beta(m)$ = SubDomains[m][d][β] and $\delta_d(n')$ = GridDesc[p].c where c = {x, y, z}[d].

```
  #define Neighbor_Grid_Offset(PS, N, SD, D, XYZ)\
    (N==0 ? 0 : (N<0 ? SubDomains[SD][D][OH_LOWER]-SubDomains[SD][D][OH_UPPER] :\
                       GridDesc[ps].XYZ))
```

update_neighbors()

The function update_neighbors() is called from init4p() with p = ps = 0, and from rebalance4p() or exchange_particles4p() with $p = 1$ when we had normal or anywhere

accommodation respectively. The function initializes/updates `AbsNeighbors`$[p][k]$ for all $k \in [0, 3^D)$ to let it have;

$$\texttt{AbsNeighbors}[p][k] = m_k = \begin{cases} \texttt{Neighbors}[p][k] & \texttt{Neighbors}[p][k] \geq 0 \\ -(\texttt{Neighbors}[p][k] + 1) & \texttt{Neighbors}[p][k] < 0 \end{cases}$$

and lets `GridOffset`$[p][k] = gidx(x_0^0(m_k, n'), \ldots)$ where $n' = \{n, parent(n)\}[p]$ and $x_d^0(m_k, n')$ is given by `Neighbor_Grid_Offset()`, as discussed in §4.9.5.

```
static void
update_neighbors(const int ps) {
  int n = 0,  nx,  ny = 0,  nz = 0;
  const int nn = nOfNodes;

  Do_Z(for (nz=-1; nz<2; nz++)) {
    Do_Y(for (ny=-1; ny<2; ny++)) {
      for (nx=-1; nx<2; nx++,n++) {
        int nbr = Neighbors[ps][n];
        nbr = AbsNeighbors[ps][n] = nbr<0 ? -(nbr+1) : nbr;
        if (nbr>=nn)  GridOffset[ps][n] = 0;
        else
          GridOffset[ps][n] =
            Coord_To_Index(Neighbor_Grid_Offset(ps, nx, nbr, OH_DIM_X, x),
                           Neighbor_Grid_Offset(ps, ny, nbr, OH_DIM_Y, y),
                           Neighbor_Grid_Offset(ps, nz, nbr, OH_DIM_Z, z),
                           GridDesc[0].w, GridDesc[0].dw);
      }
    }
  }
}
```

4.10.31 `set_grid_descriptor()`

`set_grid_descriptor()` The function `set_grid_descriptor()` is called from `init4p()` with `idx` $= i = 0$ and `nid` $= m = n$ arguments for the local node n for the initialization, from `rebalance4p()` or `exchange_particles4p()` with $i = 1$ and $m = parent(n)$ when we had normal or anywhere accommodation respectively, and from `make_recv_list()` with $i = 2$ and $m = parent(n)$, when helpand-helper reconfiguration is taking place assigning m to the local node as its secondary subdomain. The function lets `GridDesc`$[i]$ have the shape information of the per-grid histogram for the subdomain m. Note that `GridDesc`[2] is used to have the shape information of *new* $parent(n)$ due to helpand-helper reconfiguration while [1] keeps that of *old* $parent(n)$.

The function lets the elements `w`, `d` and `h` of `GridDesc`$[i]$ be $\delta_d^{\max} + 4e^g =$ `Grid`$[d]$`.size` $+ 4 \cdot$ `OH_PGRID_EXT` with $d = 0$, 1 and 2 respectively for the physical array size, and `dw` be `d` $\times$ `w`, if $D = 3$. The elements `d` and/or `h` are, however, is let be 1 if $D < 2$ or $D < 3$ respectively.

Then the elements `x`, `y` and `z` are let be $\delta_d(m) =$ `SubDomains`$[m][d][1] -$ `SubDomains` $[m][d][0]$ with $d = 0$, 1 and 2 respectively for the upper bound (or the size) of interior of the subdomian m, if $D = 3$ and $m \geq 0$. The elements `y` and/or `z` are, however, is let be 0 if $D < 2$ or $D < 3$ respectively. On the other hand, if $m < 0$ to mean that the local node does not have secondary subdomain, those elements are let be $-4e^g$ so that any

possible coordinate value less than $2e^g$ relative to the upper bound falls out-of-bounds with a coordinate less than $-2e^g$ being the absolute lower bound and thus `For_All_Grid()` with them does nothing.

```
static void
set_grid_descriptor(const int idx, const int nid) {
  const int exto2 = OH_PGRID_EXT<<2;
  const int w = GridDesc[idx].w = Grid[OH_DIM_X].size+(exto2);
  const int d = GridDesc[idx].d =
                If_Dim(OH_DIM_Y, Grid[OH_DIM_Y].size+(exto2), 1);

  GridDesc[idx].h = If_Dim(OH_DIM_Z, Grid[OH_DIM_Z].size+(exto2), 1);
  GridDesc[idx].dw = d * w;
  if (nid>=0) {
    GridDesc[idx].x = SubDomains[nid][OH_DIM_X][OH_UPPER] -
                      SubDomains[nid][OH_DIM_X][OH_LOWER];
    GridDesc[idx].y = If_Dim(OH_DIM_Y,
                             SubDomains[nid][OH_DIM_Y][OH_UPPER] -
                             SubDomains[nid][OH_DIM_Y][OH_LOWER], 0);
    GridDesc[idx].z = If_Dim(OH_DIM_Z,
                             SubDomains[nid][OH_DIM_Z][OH_UPPER] -
                             SubDomains[nid][OH_DIM_Z][OH_LOWER], 0);
  } else {
    GridDesc[idx].x = GridDesc[idx].y = GridDesc[idx].z = -exto2;
    /* to ensure, e.g., x+2*(OH_PGRID_EXT)<=-2*(OH_PGRID_EXT) */
  }
}
```

4.10.32 `adjust_field_descriptor()`

`adjust_field_descriptor()`

The function `adjust_field_descriptor()` is called from `init4p()` with argument `ps` $= p = 0$ for the initialization of the local node n's primary subdomain, and from `update_descriptors()` with $p = 1$ for n's secondary subdomain newly assigned to it by helpand-helper reconfiguration. The function modifies `FieldDesc`$[F-1]$`.{bc,red}.size`$[p]$ for per-grid histogram so that the broadcast/reduction for it are performed on all subarrays for species $s \in [0, S)$ rather than on a subarray for a certain s. That is, with their value σ set by `set_field_descriptors()`, the function updates it as $(S-1)\prod_{d=0}^{D-1} \Phi_d(F-1)+\sigma$ where $\Phi_d(f)$ is `FieldDesc`$[f]$`.size`$[d]$ and thus $\delta_d^{\max} + 4e^g$ for per-grid histogram with $f = F-1$. This means that the origial value σ specifies that the collective communication is peformed on the elements from $[s][b]$ to $[s][b+\sigma-1]$ while the updated one lets the communication be done on the elements from $[0][b]$ to $[S-1][b+\sigma-1]$.

```
static void
adjust_field_descriptor(const int ps) {
  const int f = nOfFields - 1,  ns = nOfSpecies;
  int d, fs;

  for (d=0,fs=1; d<OH_DIMENSION; d++)  fs *= FieldDesc[f].size[d];
  fs *= ns-1;
  FieldDesc[f].bc.size[ps] += fs;    FieldDesc[f].red.size[ps] += fs;
}
```

4.10.33 update_real_neighbors()

update_real_neighbors()

The function update_real_neighbors(), called from init4p(), try_primary4p(), exchange_particles4p() and make_recv_list(), updates RealDstNeighbors[][] and RealSrcNeighbors[][] according to its mode argument to specify the elements to be updated, dosec to specify whether nodes have helpers or not, and oldp $= n^p_{old}$ and newp $= n^p_{new}$ being old and new $parent(n)$ of the local node n on helpand-helper reconfiguration. Note that n^p_{new} is just $parent(n)$ in the stable state of helpand-helper configuration.

As discussed in §4.9.5, these two arrays have the neighbor node idententifiers as follows.

- RealDstNeighbors$[t][p]$ has the set of nodes which will accommodate particles that the local nodes is accommodating as its primary and secondary ones, as their primary ($p = 0$) or secondary ($p = 1$) ones in the stable ($t = 0$) or transitional ($t = 1$) state of helpand-helper configuration. Therefore, each element has the following, where $\mathcal{N}(m)$ is the set of neighbors of a subdomain m including m itself, and $H_{new}(m)$ is the set of m's helpers in the stable helpand-helper configuration or new ones in the transitional configuration.

$$\begin{aligned}
\texttt{RealDstNeighbors}[0][0] &= \mathcal{N}(n) \cup \mathcal{N}(n^p_{new}) \\
\texttt{RealDstNeighbors}[0][1] &= H_{new}(\mathcal{N}(n)) \cup H_{new}(\mathcal{N}(n^p_{new})) \\
\texttt{RealDstNeighbors}[1][0] &= \mathcal{N}(n) \cup \mathcal{N}(n^p_{old}) \\
\texttt{RealDstNeighbors}[1][1] &= H_{new}(\mathcal{N}(n)) \cup H_{new}(\mathcal{N}(n^p_{old}))
\end{aligned}$$

  Note that $\mathcal{N}(n)$ and $\mathcal{N}(n^p_{old})$ are always given by Neighbors[0][] and Neighbors[1][], while $\mathcal{N}(n^p_{new})$ is given by Neighbors[1][] or Neighbors[2][] depending on stable or transitional state of helpand-helper configuration respectively.

- RealSrcNeighbors$[t][p]$ has the set of nodes which is accommodating particles that the local nodes will accommodate as its primary ($p = 0$) or secondary ($p = 1$) ones, as their primary and secondary ones in the stable ($t = 0$) or transitional ($t = 1$) state of helpand-helper configuration. Therefore, each element has the following, where $H_{old}(m)$ is the set of m's old helpers in transitional helpand-helper configuration.

$$\begin{aligned}
\texttt{RealSrcNeighbors}[0][0] &= \mathcal{N}(n) \cup H_{new}(\mathcal{N}(n)) \\
\texttt{RealSrcNeighbors}[0][1] &= \mathcal{N}(n^p_{new}) \cup H_{new}(\mathcal{N}(n^p_{new})) \\
\texttt{RealSrcNeighbors}[1][0] &= \mathcal{N}(n) \cup H_{old}(\mathcal{N}(n)) \\
\texttt{RealSrcNeighbors}[1][1] &= \mathcal{N}(n^p_{new}) \cup H_{old}(\mathcal{N}(n^p_{new}))
\end{aligned}$$

In addition, as discussed in §4.10.5, the mode argument has one of the followings.

- URN_PRI to update [0][0] only and given in the calls from init4p() and try_primary4p().
- URN_SEC to update [0][0] and [0][1] but not [1][], and given in the call from exchange_particles4p() with anywhere accommodation in which we don't need to care about transitional helpand-helper configuration.
- URN_TRN to update all of [0][0], [0][1], [1][0] and [1][1], and given in the call from make_recv_list() with normal accommodation in which we have to care about transitional helpand-helper configuration.

```
static void
update_real_neighbors(const int mode, const int dosec, const int oldp,
                      const int newp) {
  const int me=myRank, nn=nOfNodes, nn4=nn<<2;
  const int dosec0 = mode != URN_PRI;
  int i, nbridx, ps, *doccur[2], *soccur[2];
```

At first we zero-clear all elements of `TempArray`[2][2][N] so that its element [σ][p][m] is true iff m has already been in `RealDstNeighbors`[][p]`.nbor` ($\sigma = 0$) or `RealSrcNeighbors`[][p]`.nbor` ($\sigma = 1$).

```
  for (i=0; i<nn4; i++)  TempArray[i] = 0;
  doccur[0] = TempArray;       doccur[1] = doccur[0] + nn;
  soccur[0] = doccur[1] + nn;  soccur[1] = soccur[0] + nn;
```

Then if `mode` is `URN_TRN`, we exchange [0][0] and [1][0] of `RealSrcNeighbors`[][] so that [1][0] has everything in [0][0] because $H_{old}(\mathcal{N}(n))$ is the current (i.e., old) $H_{new}(\mathcal{N}(n))$.

```
  if (mode==URN_TRN) {
    int *tmp = RealSrcNeighbors[1][0].nbor;
    RealSrcNeighbors[1][0].n = RealSrcNeighbors[0][0].n;
    RealSrcNeighbors[1][0].nbor = RealSrcNeighbors[0][0].nbor;
    RealSrcNeighbors[0][0].nbor = tmp;
  }
```

Now we call `upd_real_nbr()` twice to build the first subsets of `RealDstNeighbors`[0][] and the whole of `RealSrcNeighbors`[0][0], i.e., $\mathcal{N}(n)$ and $H_{new}(\mathcal{N}(n))$, after zero-clearing their `n` elements for cardinality of the sets in thier `nbor` elements. The arguments given to the function are as follows.

- `root` $= n$ to mean we visit $\mathcal{N}(n)$ and their helpers.
- `psp` $= 0$ to mean $\mathcal{N}(n)$ are included in [0][0]`.nbor`.
- `pss` $= 1$ for `RealDstNeighbors`[][] to mean $H_{new}(\mathcal{N}(n))$ are included in [0][1]`.nbor`, while `pss` $= 0$ for `RealSrcNeighbors`[][] to mean $H_{new}(\mathcal{N}(n))$ are included in [0][0]`.nbor`.
- `nbr` $= 0$ to mean $\mathcal{N}(n)$ is given by `Neighbors`[0][].
- `dosec` is true iff `mode` $\neq$ `URN_PRI` to mean we have to visit the elements in $H_{new}(\mathcal{N}(n))$ iff `mode` $\neq$ `URN_PRI`.
- `nodes` is `Nodes`[] to mean $H_{new}(\mathcal{N}(n))$ are obtained by traversing the list from `Nodes`[m]`.child` for $m \in \mathcal{N}(n)$.
- `rnbptr` is `RealDstNeighbors`[0][] or `RealSrcNeighbors`[0][] according to the call.
- `occur`[2] is `TempArray`[0][][] for `RealDstNeighbors`[0][] or `TempArray`[1][][] for `RealSrcNeighbors`[0][].

Then we return to the caller if `mode` = `URN_PRI` because what we need is `RealDstNeighbors`[0][0] = `RealSrcNeighbors`[0][0] = $\mathcal{N}(n)$.

```
RealDstNeighbors[0][0].n = RealDstNeighbors[0][1].n = 0;
RealSrcNeighbors[0][0].n = RealSrcNeighbors[0][1].n = 0;
upd_real_nbr(me, 0, 1, 0, dosec0, Nodes, RealDstNeighbors[0], doccur);
upd_real_nbr(me, 0, 0, 0, dosec0, Nodes, RealSrcNeighbors[0], soccur);
if (mode==URN_PRI)  return;
```

Next we call `upd_real_nbr()` twice again to obtain the second subsets of `RealDst Neighbors`[0][] and the whole of `RealSrcNeighbors`[0][1], i.e., $\mathcal{N}(n^p_{new})$ and $H_{new}(\mathcal{N}(n^p_{new}))$. The arguments given to the function are as follows.

- `root` = n^p_{new} to mean we visit $\mathcal{N}(n^p_{new})$ and their helpers.
- `psp` = 0 for `RealDstNeighbors`[][] to mean $\mathcal{N}(n^p_{new})$ are included in [0][0]`.nbor`, whiele `psp` = 1 for `RealSrcNeighbors`[][] to mean $\mathcal{N}(n^p_{new})$ are included in [0][1]`.nbor`.
- `pss` = 1 to mean $H_{new}(\mathcal{N}(n^p_{new}))$ are included in [0][1]`.nbor`.
- `nbr` = 2 or 1 to mean $\mathcal{N}(n^p_{new})$ is given by `Neighbors`[2][] if `mode` = `URN_TRN`, or `Neighbors`[1][] otherwise, i.e., `mode` $\neq$ `URN_TRN` respectively. That is, $\mathcal{N}(n^p_{new})$ has neighbors of the stable helpand or newly assigned helpand.
- `dosec` is unconditionally true to mean always we have to visit the elements in $H_{new}(\mathcal{N}(n^p_{new}))$.
- `nodes` is `Nodes`[] to mean $H_{new}(\mathcal{N}(n^p_{new}))$ are obtained by traversing the list from `Nodes`[m]`.child` for $m \in \mathcal{N}(n^p_{new})$.
- `rnbptr` is `RealDstNeighbors`[0][] or `RealSrcNeighbors`[0][] according to the call.
- `occur`[2] is `TempArray`[0][][] for `RealDstNeighbors`[0][] or `TempArray`[1][][] for `RealSrcNeighbors`[0][].

Then if `mode` $\neq$ `URN_TRN`, we return to caller with;

$$\texttt{RealDstNeighbors}[0][] = \{\mathcal{N}(n) \cup \mathcal{N}(n^p_{new}),\ H_{new}(\mathcal{N}(n)) \cup H_{new}(\mathcal{N}(n^p_{new}))\}$$
$$\texttt{RealSrcNeighbors}[0][] = \{\mathcal{N}(n) \cup H_{new}(\mathcal{N}(n)),\ \mathcal{N}(n^p_{new}) \cup H_{new}(\mathcal{N}(n^p_{new}))\}$$

```
nbridx = mode==URN_TRN ? 2 : 1;
upd_real_nbr(newp, 0, 1, nbridx, 1, Nodes, RealDstNeighbors[0], doccur);
upd_real_nbr(newp, 1, 1, nbridx, 1, Nodes, RealSrcNeighbors[0], soccur);
if (mode!=URN_TRN)  return;
```

Finally, we call `upd_real_nbr()` thrice, twice for `RealDstNeighbors`[1][] and once for `RealSrcNeighbors`[1][1], after zero-clearing `TempArray`[σ][p][m] for all m such that $m \in$ `RealDstNeighbors`[0][p]`.nbor` ($\sigma = 0$) and $m \in$ `RealSrcNeighbors`[0][p]`.nbor` ($\sigma = 1$).

The first two calls for `RealDstNeighbors`[1][] are to obtain;

$$\texttt{RealDstNeighbors}[1][0] = \mathcal{N}(n) \cup \mathcal{N}(n^p_{old})$$
$$\texttt{RealDstNeighbors}[1][1] = H_{new}(\mathcal{N}(n)) \cup H_{new}(\mathcal{N}(n^p_{old}))$$

for which the first and second calls build the first and second half of each element respectively. To these two calls, we give the following arguments.

- `root` $= n$ for the first call and n^p_{old} for the second to visit $\mathcal{N}(n)$ or $\mathcal{N}(n^p_{old})$ and their helpers respectively.
- `psp` $= 0$ to mean $\mathcal{N}(n)$ and $\mathcal{N}(n^p_{old})$ are included in `RealDstNeighbors`[1][0]`.nbor`, commonly.
- `pss` $= 1$ to mean $H_{new}(\mathcal{N}(n))$ and $H_{new}(\mathcal{N}(n^p_{old}))$ are included in `RealDstNeighbors`[1][1]`.nbor`, commonly.
- `nbr` $= 0$ for the first call and 1 for the second to mean $\mathcal{N}(n)$ and $\mathcal{N}(n^p_{old})$ are given by `Neighbors`[0][] and `Neighbors`[1][] respectively.
- `dosec` is true to visit $H_{new}(\mathcal{N}(n))$ and $H_{new}(\mathcal{N}(n^p_{old}))$ unconditionally (as far as n^p_{old} exists.
- `nodes` is commonly `Nodes`[] to mean $H_{new}(\mathcal{N}(m))$ are obtained by traversing the list from `Nodes`$[m']$`.child` for $m' \in \mathcal{N}(m)$ where $m \in \{n, n^p_{old}\}$.
- `rnbptr` is commonly `RealDstNeighbors`[1][].
- `occur`[2] is commonly `TempArray`[0][][].

On the other hand, the third and last call to obtain;

$$\texttt{RealSrcNeighbors}[1][1] = \mathcal{N}(n^p_{new}) \cup H_{old}(\mathcal{N}(n^p_{new}))$$

gives the following arguments to `upd_real_nbr()`.

- `root` $= n^p_{new}$ to visit $\mathcal{N}(n^p_{new})$ and their helpers.
- `psp` $=$ `pss` $= 1$ to mean $\mathcal{N}(n^p_{new})$ and $H_{old}(\mathcal{N}(n^p_{new}))$ are included in `RealSrcNeighbors`[1][1]`.nbor`, respectively.
- `nbr` $= 2$ to mean $\mathcal{N}(n^p_{new})$ is given by `Neighbors`[2][].
- `dosec` is that in this function's argument to mean we visit $H_{old}(\mathcal{N}(n^p_{new}))$ if we were in secondary mode.
- `nodes` is `NodesNext`[] to mean $H_{old}(\mathcal{N}(n^p_{new}))$ is obtained by traversing the list from `NodesNext`$[m]$`.child` for $m \in \mathcal{N}(n^p_{new})$, becuse `NodesNext`[] is `Nodes`[] in the last step when `update_real_neighbors()` is called.
- `rnbptr` is `RealSrcNeighbors`[1][].
- `occur`[2] is `TempArray`[1][][].

```
  for (ps=0; ps<2; ps++) {
    const int nd = RealDstNeighbors[0][ps].n;
    const int ns = RealSrcNeighbors[0][ps].n;
    for (i=0; i<nd; i++)  doccur[ps][RealDstNeighbors[0][ps].nbor[i]] = 0;
    for (i=0; i<ns; i++)  soccur[ps][RealSrcNeighbors[0][ps].nbor[i]] = 0;
  }
  RealDstNeighbors[1][0].n = RealDstNeighbors[1][1].n = 0;
  RealSrcNeighbors[1][1].n = 0;
  upd_real_nbr(me,   0, 1, 0, 1,     Nodes,     RealDstNeighbors[1], doccur);
  upd_real_nbr(oldp, 0, 1, 1, 1,     Nodes,     RealDstNeighbors[1], doccur);
  upd_real_nbr(newp, 1, 1, 2, dosec, NodesNext, RealSrcNeighbors[1], soccur);
}
```

4.10.34 upd_real_nbr()

upd_real_nbr() The function upd_real_nbr(), called solely from update_real_neighbors() but up to seven times, to add members of $\mathcal{N}(r)$ and, if dosec argument is ture, $H(\mathcal{N}(r))$ to $RN[p_p]$ and $RN[p_s]$ respectively where r = root $\in \{n, n^p_{new}, n^p_{old}\}$, $\mathcal{N}(r)$ is given by Neighbors[nbr], $H(m)$ is given by $NN[m]$ = nodes[m] being either of Nodes[] or NodesNext[], $RN[2]$ = rnbrptr[2] is either of RealDstNeighbors[t][] or RealSrcNeighbors[t][] with $t \in \{0, 1\}$, and p_p = psp and p_s = pss. Another argument occur[2][N] being TempArray[σ][][] with $\sigma \in \{0, 1\}$ to indicate that a node m has already been in $RN[p]$ iff occur[p][m] is true.

```
static void
upd_real_nbr(const int root, const int psp, const int pss,
             const int nbr, const int dosec, struct S_node *nodes,
             struct S_realneighbor rnbrptr[2], int *occur[2]) {
  const int me=myRank;
  struct S_realneighbor *pnbr = rnbrptr+psp,  *snbr = rnbrptr+pss;
  int *poccur = occur[psp],  *soccur = occur[pss];
  int i;
```

At first we check if $r < 0$ to mean n^p_{new} or n^p_{old} does not exist, and return to the caller doing nothing if so. Otherwise, we add r to $RN[p_p]$.nbor[] if $r \neq n$ and r is not in the set. Then, if dosec is true, we traverse the list $NN[r]$.child to add each of its member $m \in H(r)$ to $RN[p_s]$.nbor[] if $m \neq n$ and m is not in the set. Note that we exclude n itself from both sets because n does not communicate with itself in the communication in its primary/secondary families. Also note that we visit $H(r)$ even if occur[p_p][r] is true because it could have been visited as a helper and $p_p = p_s$.

```
  if (root<0)  return;
  if (root!=me && !poccur[root]) {
    pnbr->nbor[pnbr->n++] = root;  poccur[root] = 1;
  }
  if (dosec) {
    struct S_node *ch;
    for (ch=nodes[root].child; ch; ch=ch->sibling) {
      const int nid = ch->id;
      if (nid!=me && !soccur[nid]) {
        snbr->nbor[snbr->n++] = nid;  soccur[nid] = 1;
      }
    }
  }
```

Next we traverse Neighbors[nbr][k] for all $k \in [0, 3^D)$ to have $\mathcal{N}(r)$ and add each $m \in \mathcal{N}(r)$ to $RN[p_p]$.nbor[] if $m \neq r$ and m is not in the set. If so and dosec is true, we also traverse the list $NN[m]$.child to add each of its member m' to $RN[p_s]$.nbor[] if m' is not in the set. Note that we exclude neither of n nor r from both sets because they can be a helper of their (or n's helpand's) neighbors and, for n, it must perform self communication for particle movement crossing the boundary of its primary/secondary subdomains. Also note that we visit $H(m)$ even if occur[p_p][m] is true because it could have been visited as a helper and $p_p = p_s$.

```
  for (i=0; i<OH_NEIGHBORS; i++) {
```

```
      const int nid = Neighbors[nbr][i];
      struct S_node *ch;
      if (nid<0 || nid==root)  continue;
      if (!poccur[nid]) {
        pnbr->nbor[pnbr->n++] = nid;  poccur[nid] = 1;
      }
      if (dosec) {
        for (ch=nodes[nid].child; ch; ch=ch->sibling) {
          const int cid = ch->id;
          if (!soccur[cid]) {
            snbr->nbor[snbr->n++] = cid;  soccur[cid] = 1;
          }
        }
      }
    }
  }
```

4.10.35 exchange_xfer_amount()

exchange_xfer_amount()

The function exchange_xfer_amount(), called solely from exchange_particles4p(), exchanges NOfSend[][][] in the nodes responsible of a subdomain and its neighbors as the nodes' primary/secondary subdomain to have NOfRecv[][][] for position-aware particle transfer when we will be in secondary mode in the next step. The function is given two arguments; trans $= t \in \{0, 1\}$ to mean we have stable ($t = 0$) or transitional ($t = 1$) state of helpand-helper configuration and to be used to refer to RealSrcNeighbors$[t][][]$ and RealDstNeighbors$[t][][]$ to find the senders/receivers of NOfSend[][][], respectively; and psnew $= p_n \in \{0, 1\}$ to mean the local node will have a secondary subdomain ($p_n = 1$) and thus will receive some particles, or not.

```
  static void
  exchange_xfer_amount(const int trans, const int psnew) {
    const struct S_realneighbor *snbr = RealSrcNeighbors[trans];
    const struct S_realneighbor *dnbr = RealDstNeighbors[trans];
    const int nnns = nOfNodes * nOfSpecies;
    int ps, tag, req;
```

First we post MPI_Irecv() to receive NOfRecv$[p][s][m]$ from all nodes $m \in$ RealSrcNeighbors$[t][p]$.nbor[] which is m's NOfSend$[p][s][n]$ for the local node n, for $p \in \{0, p_n\}$. The receiving data has the MPI's data-type T_Hgramhalf for a slice $[*][N]$ of an integer array $[S][N]$. Since a node m can appear in both of RealSrcNeighbors$[t][0]$ and RealSrcNeighbors$[t][1]$, we give a tag 0 and NS for $p = 0$ and $p = 1$ respectively to MPI_Irecv() to distinguish them.

```
    for (ps=0,tag=0,req=0; ps<=psnew; ps++,tag+=nnns) {
      const int n = snbr[ps].n;
      const int *nbor = snbr[ps].nbor;
      int i,  *nrbase = NOfRecv + tag;
      for (i=0; i<n; i++,req++) {
        const int nid = nbor[i];
        MPI_Irecv(nrbase+nid, 1, T_Hgramhalf, nid, tag, MCW, Requests+req);
      }
    }
```

Next, we send local node n's `NOfSend`$[p][s][m]$ to all nodes $m \in$ `RealDstNeighbors`$[t][p]$`.nbor[]` so as to be received into m's `NOfRecv`$[p][s][n]$, for $p \in \{0, 1\}$ by `MPI_Isend()`. The data-type and tag are same as the receiving counterpart `MPI_Irecv()`, i.e., `T_Hgramhalf` and $\{0, NS\}[p]$.

```
  for (ps=0,tag=0; ps<2; ps++,tag+=nnns) {
    const int n = dnbr[ps].n;
    const int *nbor = dnbr[ps].nbor;
    int i,  *nsbase = NOfSend + tag;
    for (i=0; i<n; i++,req++) {
      const int nid = nbor[i];
      MPI_Isend(nsbase+nid, 1, T_Hgramhalf, nid, tag, MCW, Requests+req);
    }
  }
```

Finally, we confirm the completion of all `MPI_Irecv()` and `MPI_Isend()` calls recorded in `Requests[]` by `MPI_Waitall()` to have their completion status in `Statuses[]` (but not referring to).

```
  MPI_Waitall(req, Requests, Statuses);
}
```

4.10.36 `count_population()`

`count_population()` The function `count_population()`, called from `try_primary4p()` and `exchange_particles4p()` when they find we have anywhere accmmodation, counts the particle population in each grid-voxel to have the per-grid histogram in `NOfPGrid[][][]` after we perform non-position-aware particle transfer. The function is given three arguments; `nextmode` being 0 or 1 for the call from `try_primary4p()` or `exchange_particles4p()` respectively to mean we will be in primary/secondary mode respectively; $p_n =$ `psnew` $= 1$ iff the local node will have a secondary subdomain; and `stats` $= 0$ for the call from `exchange_particles4p()` because `count_population()` is included in the previous region for the timing measurement, while it is `stats` argument of the caller `try_primary4p()` because `count_population()` starts a new region.

```
static void
count_population(const int nextmode, const int psnew, const int stats) {
  int ps, s, t, i, j, tp;
  const int ns=nOfSpecies, exti=OH_PGRID_EXT;
  Decl_For_All_Grid();
  Decl_Grid_Info();
```

After starting the new timing measurement with the key `STATS_TB_SORT` and `nextmode` if required by `stats` $\neq 0$, we do the followings for all $p \in [0, p_n]$ and $s \in [0, S)$. First we zero-clear the per-grid histogram `NOfPGrid`$[ps][s][g]$ for all $g = gidx(x, y, z)$ for $(x, y, z) \in [0, \delta_x(n)) \times [0, \delta_y(n)) \times [0, \delta_y(n))$ by `For_All_Grid()` and `The_Grid()` to have g.

Then we scan all particles `Particles`$[i]$ in $pbuf(p, s)$, whose size is `TotalPNext`$[p][s]$ which has already been set by `move_to_sendbuf_primary()` called from `try_primary4p()`, or `move_to_sendbuf_secondary()` called from `exchange_particles()` called from `exchange_particles4p()`. The size is also set into `TotalP`$[p][s]$ for the reference

in `sort_particles()`, `move_to_sendbuf_sec4p()` and its callees, or `move_and_sort_secondary()`, together with the sum of `TotalPNext`$[0][s]$ for all $s \in [0, S)$ set into `primaryParts` and that of `TotalPNext`$[p][s]$ for all $p \in \{0, 1\}$ set into `totalParts`.

For each `Particles`$[i]$, we extract its grid-position g by `Grid_Position()` and increment `NOfPGrid`$[p][s][g]$ to let it have per-grid histogram finally. We also make `nid` of each particle have $\lfloor 3^D/2 \rfloor 2^\Gamma + g$ so that all particles look like staying in the local node's primary/secondary subdomain. This operation is necessary because a particle has traveled from a neighbor subdomain can have a neighbor index $k \neq \lfloor 3^D/2 \rfloor$ with which other functions should confuse that it should go out to a neighbor subdomain.

Finally, we let `nOfInjections` be 0 because all injected particles have been processed by the first-phase non-position-aware particle transfer.

```
  if (stats) oh1_stats_time(STATS_TB_SORT, nextmode);
  for (ps=0,t=0,j=0,tp=0; ps<=psnew; ps++) {
    for (s=0; s<ns; s++,t++) {
      dint *npgs = NOfPGrid[ps][s];
      const int tpn = TotalP[t] = TotalPNext[t];
      tp += tpn;
      For_All_Grid(ps, -exti, -exti, -exti, exti, exti, exti)
        npgs[The_Grid()]=0;
      for (i=0; i<tpn; i++,j++) {
        const int g = Grid_Position(Particles[j].nid);
        npgs[g]++;
        Particles[j].nid = Combine_Subdom_Pos(OH_NBR_SELF, g);
      }
    }
    if (ps==0)  primaryParts = tp;
  }
  totalParts = tp;  nOfInjections = 0;
}
```

4.10.37 `sort_particles()`

`sort_particles()` The function `sort_particles()` is called from the following functions to move particles in `Particles[]` to `SendBuf[]` with sorting.

- `try_primary4p()` when it finds we have anywhere accommodation with which we have to gather primary particles at first and then sort them. It gives `npg` = `NOfPGrid[][][]` as the complete per-grid histogram built by `count_population()`, `nextmode` = `psnew` = 0 because we will be in primary mode and thus we have only primary particles to be sorted, and `stats` = 0 because the timing measurement with `STATS_TB_SORT` has already been started by `count_population()` preceding the call.

- `try_primary4p()` when it finds we have normal accommodation but $P_n + P_n^{\text{send}} > P_{lim}$ to mean we have to transfer particles among neighbors at first and then sort all primary particles. It gives `npg` = `NOfPGrid[][][]` if we were in primary mode in which the per-grid histogram was built by the neighbor communication by `exchange_population()`, or `npg` = `NOfPGridTotal[][][]` if we were in secondary mode in which we needed in-family reduction in `exchange_population()` to have the per-grid histogram. It also gives `nextmode` = `psnew` = 0 because we will be in primary mode and thus we have only primary particles to be sorted, and its own `stats` argument to `stats` to start the timing measurement with `STATS_TB_SORT` if required by `stats` $\neq 0$.

- `exchange_particles4p()` when it finds $P_n + P_n^{\rm send} > P_{lim}$ to mean we have to transfer particles among neighbors at first and then sort all primary and secondary particles. Since the per-grid histogram is built in `NOfPGridOut[][][]` by `make_send_sched()` and its element is `int` instead of `dint` of `NOfPGrid[][][]` and `NOfPGridTotal[][][]`, the per-grid histogram is not given through the argument `npg`, which has `NULL` but its use is specified by `nextmode` $\neq 0$. The argument `nextmode` also specifies the index of `GridDesc[]` used for secondary particles, i.e., [1] or [2] for stable or transitional state of helpand-helper configuration respectively. The argument `psnew` is 1 iff the local node will have a secondary subdomain, i.e., it is not the root of the family tree. The argument `stats` is that of the caller to start the timing measurement with `STATS_TB_SORT` if required by `stats` $\neq 0$.

```
static void
sort_particles(dint ***npg, const int nextmode, const int psnew,
               const int stats) {
  const int ns=nOfSpecies;
  struct S_particle *p = Particles;
  int ps, s, t, i, npt;
  Decl_For_All_Grid();
  Decl_Grid_Info();
```

After starting the new timing measurement with the key `STATS_TB_SORT` if required by `stats` $\neq 0$, we do the followings for all $p \in [0, p_n]$ and $s \in [0, S)$ where $p_n =$ `psnew`. First, we build the index array for bucket sort in `NOfPGridTotal[][][]` as follows.

$$G_p = \{g \,|\, g = gidx(x,y,z),\ (x,y,z) \in \prod_{d=0}^{D-1} [0, \delta_d(m)),\ m = \{n, parent(n)\}[p]\}$$

$$\texttt{NOfPGridTotal}[p][s][g] = \sum_{q=0}^{p-1}\sum_{t=0}^{S-1}\sum_{h\in G_0} \mathcal{P}(q,t,h) + \sum_{t=0}^{s-1}\sum_{h\in G_p} \mathcal{P}(p,t,h) + \sum_{h<g} \mathcal{P}(p,s,h)$$

where $\mathcal{P}(p,s,h)$ is `NOfPGridOut`$[p][s][h] = \mathcal{P}_O(p,s,h)$ if `nextmode` $\neq 0$, or `npg`$[p][s][h] =$ `NOfPGrid`$[p][s][g] = \mathcal{P}_L(p,s,h)$ or `NOfPGridTotal`$[p][s][g] = \mathcal{P}_T(p,s,h)$ otherwise. In the latter case, we let `NOfPGridOut`$[p][s][h] = \mathcal{P}_O(p,s,h)$ have the value `npg`$[p][s][h]$ as the number of particles the local node accommodates in the grid-voxel at g. The array `NOfPGridTotal[][][]` is built scanning elements by `For_All_Grid()` whose first argument is 0 when $p = 0$, or `nextmode` $\in \{1, 2\}$ when $p = 1$ to specify the index of `GridDesc[]`.

Then we scan all particles `Particles`$[i]$ in $pbuf(p,s)$, whose size is `TotalPNext`$[p][s]$ which has already been set by `move_to_sendbuf_primary()` called from `try_primary4p()`, or `move_to_sendbuf_sec4p()` called from `exchange_particles4p()`. For each `Particles`$[i]$, we extract its grid-position g by `Grid_Position()` and move it to `SendBuf`[`NOfPGridTotal`$[p][s][g]$] and then increment `NOfPGridTotal`$[p][s][g]$ for the next particle in the grid-voxel at g.

```
  if (stats) oh1_stats_time(STATS_TB_SORT, nextmode?1:0);
  for (ps=0,t=0,npt=0; ps<=psnew; ps++) {
    for (s=0; s<ns; s++,t++) {
      int *npgo = NOfPGridOut[ps][s];
      dint *npgt = NOfPGridTotal[ps][s];
      const int tpn = TotalPNext[t];
```

```
      if (nextmode) {
        const int gdidx = ps ? nextmode : 0;
        For_All_Grid(gdidx, 0, 0, 0, 0, 0, 0) {
          const int np = npgo[The_Grid()];
          npgt[The_Grid()] = npt;  npt += np;
        }
      } else {
        dint *npgs = npg[ps][s];
        For_All_Grid(0, 0, 0, 0, 0, 0, 0) {
          const int np = npgo[The_Grid()] = npgs[The_Grid()];
          npgt[The_Grid()] = npt;  npt += np;
        }
      }
      for (i=0; i<tpn; i++,p++)
        SendBuf[npgt[Grid_Position(p->nid)]++] = *p;
    }
  }
}
```

4.10.38 `move_and_sort_primary()`

`move_and_sort_primary()` The function `move_and_sort_primary()` is called solely from `try_primary4p()` when it finds we have normal accommodation and $P_n + P_n^{\text{send}} \leq P_{lim}$ to mean we can move particles staying in and leaving from the local node toghther from `Particles[]` to `SendBuf[]` with sorting. It receives the following three arguments; `npg` = `NOfPGrid[][][]` if we were in primary mode or `npg` = `NOfPGridTotal[][][]` otherwise to show the complete per-grid histogram; `psold` $= p_c$ is 1 iff we were in secondary mode and the local node had a secondary subdomain; and `stats` to mean we have to start new timing measurement if required by `stats` $\neq 0$.

```
static void
move_and_sort_primary(dint ***npg, const int psold, const int stats) {
  const int nn=nOfNodes, ns=nOfSpecies, nnns=nn*ns, me=myRank;
  const int ninj=nOfInjections,  sbase=specBase;
  struct S_particle *rbb, *p, *sbuf;
  int ps, s, t, i, nacc, mysd, *sbd;
  Decl_For_All_Grid();
  Decl_Grid_Info();
```

After starting the new timing measurement with the key `STATS_TB_MOVE` if required by `stats` $\neq 0$, we do the followings for all $s \in [0, S)$. For the local node n, at first we let

$$\texttt{TotalPNext}[0][s] = \sum_{m=0}^{N-1} \sum_{p\in\{0,1\}} q(m)[p][s][n] = \sum_{m=0}^{N-1} \sum_{p\in\{0,1\}} \texttt{NOfPrimaries}[p][s][m]$$

and `TotalPNext`$[1][s] = 0$. We also let `RecvBufBases`$[0][s]$ point `Particles`$[r(s)]$ where;

$$\begin{aligned} r(s) &= \sum_{t=0}^{s-1} \left(\left(\sum_{m=0}^{N-1} \sum_{p\in\{0,1\}} q(m)[p][t][n] \right) - q(n)[0][t][n] \right) \\ &= \sum_{t=0}^{s-1} (\texttt{TotalPNext}[0][t] - \texttt{NOfPLocal}[0][t][n]) \end{aligned}$$

to mean $rbuf(0,s)$ are continually ranked from `Particles`[0] and its size is the number of particles of species s in n's primary subdomain excluding those the local node has already accommodated as its primary particles. In addition, we let `NOfPLocal`[0][s][n] = 0 and `InjectedParticles`[0][p][s] = $q^{\rm inj}(n)[p][s] = 0$ for $p \in \{0,1\}$ so that `set_sendbuf_disps()` excludes particles staying in n and ignores injected particles when it builds `SendBufDisps`[s][].

Then we scan `NOfPGrid`[0][s][g] or `NOfPGridTotal`[0][s][g] given through the argument `npg` by `For_All_Grid()` to copy it into `NOfPGridOut`[0][s][g] and build the index array for sorting in `NOfPGridTotal`[0][s][g] as discussed in §4.10.37.

```
if (stats) oh1_stats_time(STATS_TB_MOVE, 0);
for (s=0,t=0,nacc=0,rbb=Particles; s<ns; s++,t+=nn) {
  int n, tpn, *npgo=NOfPGridOut[0][s], *nprime=NOfPrimaries+t;
  dint *npgs=npg[0][s], *npgt=NOfPGridTotal[0][s];
  for (n=0,tpn=0; n<nn; n++)  tpn += nprime[n] + nprime[n+nnns];
  TotalPNext[s] = tpn;  TotalPNext[ns+s] = 0;
  RecvBufBases[s] = rbb;  rbb += tpn - NOfPLocal[t+me];
  NOfPLocal[t+me] = 0;
  InjectedParticles[s] = InjectedParticles[ns+s] = 0;
  For_All_Grid(0, 0, 0, 0, 0, 0, 0) {
    const int np = npgo[The_Grid()] = npgs[The_Grid()];
    npgt[The_Grid()] = nacc;  nacc += np;
  }
}
```

Then we let `RecvBufBases`[0][S] (or [1][0]) have the value for S defined above point the entry next to the tail of $pbuf(0,S-1)$. Then we call `set_sendbuf_disps()`, giving it `secondary` = p_c to let it take care of secondary particles if exist and `parent` = -1 to mean the local node will not have helpand, to build `SendBufDisps`[s][m] for $sbuf(s,m)$.

Next we perform the core part of the sorting by scanning $pbuf(p,s)$ for all $p \in \{0,p_c\}$ and $s \in [0,S)$ whose size is `TotalP`[p][s]. For each `Particles`[i] having non-negative `nid` element, we extract its subdomain identifer m and grid-position g by `Neighbor_Subdomain_Id()` and `Grid_Position()` respectively. Then if $p = 0$ and $m = n$, we move it to `SendBuf`[`NOfPGridTotal`[0][s][g]] and then increment `NOfPGridTotal`[0][s][g] for the next particle in the grid-voxel at g. Otherwise, we move it to `SendBuf`[P_n + `SendBufDisps`[s][m]] and then increment `SendBufDisps`[s][m] for the next particle to be sent to m. Note that we move secondary particles of species s in the local node's primary subdomain accidentally to $sbuf(s,n)$ rather than $pbuf(0,s)$ because `exchange_primary_particles()` treats them as the target of all-to-all communication.

```
RecvBufBases[s] = rbb;
sbuf = SendBuf + nacc;
set_sendbuf_disps(psold, -1);
for (ps=0,t=0,p=Particles,mysd=me; ps<=psold; ps++,mysd=-1) {
  for (s=0,sbd=SendBufDisps; s<ns; s++,t++,sbd+=nn) {
    dint *npgt = NOfPGridTotal[0][s];
    const int itail = TotalP[t];
    for (i=0; i<itail; i++,p++) {
      const OH_nid_t nid = p->nid;
      if (nid>=0) {
        const int sdid = Neighbor_Subdomain_Id(nid, ps);
        if (sdid==mysd)  SendBuf[npgt[Grid_Position(nid)]++] = *p;
        else             sbuf[sbd[sdid]++] = *p;
```

```
        }
      }
    }
  }
```

Then we scan injected particles whose amount is `nOfInjections` $= Q_n^{\rm inj}$ residing beyond the last $pbuf(p_c, S-1)$. For each `Particles`$[i]$ having non-negative `nid` element, we extract its subdomain identifer m, grid-position g and species s by `Subdomain_Id()`, `Grid_Position()`, `Particle_Spec()` respectively. Then if the `nid` of the particle is non-negative and $m = n$, we move it to `SendBuf`[`NOfPGridTotal`$[0][s][g]$] and then increment `NOfPGridTotal`$[0][s][g]$ for the next particle in the grid-voxel at g. Note that the second argument of `Subdomain_Id()` is 0 to let it to refer to `AbsNeighbors`$[0]$[] if necessary. This is correct for primary injected particles, and also for secondary ones because the subdomain code is not less than 3^D and thus the macro should give us m not less than $N + 3^D$ and thus N without looking up `AbsNeighbors`[][].

Otherwise, m can be greater than $N-1$ to mean the particle was injected into or around the local node's old secondary subdomain. If so, we perform `Primarize_Id()` to let the particle has $\sigma' = \sigma - (N + 3^D)$ in its subdomain code and to have real m. This operation is subtle because `Primarize_Id()` may refers to `AbsNeighbors`$[1][k]$ for the k-th neighbor of the local node's old secondary subdomain through the macro `Subdomain_Id()` invoked in it. However, the array keeps correct values and thus the resulting subdomain identifier m is also correct.

Then we move it to `SendBuf`$[P_n$+`SendBufDisps`$[s][m]]$ and then increment `SendBufDisps` $[s][m]$ for the next particle to be sent to m. Note that we move particles of $m = n$ to $sbuf(s, m)$ again.

Finally we call `set_sendbuf_disps()` again so that `SendBufDisps`[][] regains the displacements of $sbuf(s, m)$ for the reference in `exchange_primary_particles()`.

```
    for (i=0; i<ninj; i++,p++) {
      const OH_nid_t nid = p->nid;
      const int s = Particle_Spec(p->spec-sbase);
      int sdid;
      if (nid<0) continue;
      sdid = Subdomain_Id(nid, 0);
      if (sdid==me) SendBuf[NOfPGridTotal[0][s][Grid_Position(nid)]++] = *p;
      else {
        if (sdid>=nn) Primarize_Id(p, sdid);
        sbuf[SendBufDisps[s*nn+sdid]++] = *p;
      }
    }
    set_sendbuf_disps(psold, -1);
  }
```

4.10.39 `sort_received_particles()`

`sort_received_particles()`

The function `sort_received_particles()` is called from `try_primary4p()` with the argument `nextmode` $= 0$ and `exchange_particles4p()` with `nextmode` $= 1$, to sort particles received from other nodes when we have normal accommodation and $Q_n + P_n^{\rm send} \le P_{lim}$ and the local node will have secondary subdomain (`psnew` $= p_n = 1$) or not ($p_n = 0$) in the next step. The other argument `stats` means we have to start new timing measurement if required by `stats` $\neq 0$.

After starting the new timing measurement with the key `STATS_TB_SORT` and `nextmode` if required by `stats` $\neq 0$, the function scans $rbuf(p,s)$ for all $p \in \{0, p_n\}$ and $s \in [0, S)$, which are contiguously ranked from `Particles`[0] and are pointed by `RecvBufBases`$[p][s]$. For each `Particles`$[i]$ in $rbuf(p,s)$, we extract its grid-position g by `Grid_Position()` to move it to `SendBuf`[`NOfPGridTotal`$[p][s][g]$] and then increment `NOfPGridTotal`$[p][s][g]$ for the next received particle in the grid-voxel at g. Note that all $rbuf(p,s)$ are contiguously ranked and thus `RecvBufBases`$[pS+s+1]$ points the entry next to the tail of $rbuf(p,s)$ including the case of $p=1$ and $s=S-1$.

```
static void
sort_received_particles(const int nextmode, const int psnew, const int stats) {
  const int ns=nOfSpecies;
  int ps, s;
  struct S_particle *p = Particles, **rbb = RecvBufBases+1;
  Decl_Grid_Info();

  if (stats) oh1_stats_time(STATS_TB_SORT, nextmode);
  for (ps=0; ps<=psnew; ps++) {
    for (s=0; s<ns; s++,rbb++) {
      dint *npgt = NOfPGridTotal[ps][s];
      const struct S_particle *rbtail = *rbb;
      for (; p<rbtail; p++)  SendBuf[npgt[Grid_Position(p->nid)]++] = *p;
    }
  }
}
```

4.10.40 Macros `Local_Grid_Position()` and `Move_Or_Do()`

`Local_Grid_Position()` The macro `Local_Grid_Position`$(g, k{\cdot}2^\Gamma{+}g, p)$, used in the macro `Move_Or_Do()` and the function `oh4p_remove_mapped_particle()` directly, transforms a particle's grid-position g in the k-th neighbor of the local node n's primary ($p=0$) or secondary ($p=1$) subdomain into its corresponding index g' in the n's subdomain. Note that it is assured that the second argument `nid` of the particle should have $k \cdot 2^\Gamma + g$ because `Move_Or_Do()` is used in the functions called in normal accommodation or after the subdomain codes in `nid` of all particles are let be $\lfloor 3^D/2 \rfloor$ by `count_population()` with anywhere accommodation, and `oh4p_remove_mapped_particle()` uses this macro if we are in normal accommodation.

Therefore, g' is obtained by $g +$ `GridOffset`$[p][k]$ as discussed in §4.9.5.

```
#define Local_Grid_Position(G, NID, PS)  ((G) + GridOffset[PS][NID>>loggrid])
```

`Move_Or_Do()` The macro `Move_Or_Do`(π, p, n', μ, a), used in the particle scanning loop in `move_to_sendbuf_sec4p()`, `move_to_sendbuf_uw4p()`, `move_to_sendbuf_dw4p()` and `move_and_sort_secondary()`, examines a primary ($p=0$) or secondary ($p=1$) particle π of species s in `Particles`[]. If `nid` of the particle is negative to mean the particle was eliminated, we skip the iteration of the loop by `continue`. Otherwise we obtain its subdomain identifier m by `Neighbor_Subdomain_Id()` and grid-position g by `Grid_Position()` if $m = n'$, or by transforming what the macro gives into the local coordinate g by `Local_Grid_Position()` otherwise, where n' is the primary/secondary subdomain of the local node.

```
#define Move_Or_Do(P, PS, MYSD, MOVEIF, ACT) {\
  const OH_nid_t nid = P->nid;\
  int g = Grid_Position(nid);\
  int sdid, dst;\
  if (nid<0)  continue;\
  sdid = Neighbor_Subdomain_Id(nid, PS);\
  if (sdid!=(MYSD)) g = Local_Grid_Position(g, nid, PS);\
  dst = npg[g];\
```

Then if $c = \texttt{npg}[g] = \texttt{NOfPGrid}[p][s][g] = 0$ to mean that the particle should stay in the local node, we perform the operation a specified by the macro invoker. Note that `npg` is an implicit argument given to the macro and has the pointer to $\texttt{NOfPGrid}[p][s][0]$.

Otherwise and if $\mu \neq 0$, we do the followings. If $c > 0$ to mean that $c = ((p'S + s)N + m')+1$ for the one-dimensional index of $[p'][s][m']$ for sending π to the node m', we move it to $\texttt{SendBuf}[\beta + \texttt{NOfSend}[p'][s][m']]$ and then increment $\texttt{NOfSend}[p'][s][m']$ for the next particle to be sent to m', where $\beta = Q_n$ if the grand-invoker is `move_and_sort_secondary()`, or $\beta = 0$ otherwise. Note that the pointer `sb` to $\texttt{SendBuf}[\beta]$ is another implicit argument.

Otherwise, c is the negative index to the `S_commlist` record in hot-spot sending block for a hot spot at g, namely $C = \texttt{CommList}[-(c+1)]$. If $t = C\texttt{.tag} < 0$ to mean the record is for the local node itself, we let $\texttt{NOfPGrid}[p][s][g] = 0$ and perform the action a for staying particles.

Otherwise, we move the particle to $\texttt{SendBuf}[\beta + \texttt{NOfSend}[p'][s][m']]$ to send it to $m' = C\texttt{.rid}$ and increment $\texttt{NOfSend}[p'][s][m']$ for the next particle to be sent to m'. Note that the one-dimensional index of $\texttt{NOfSend}[p'][s][m']$ is obtained by $t+m'$ because $t = (p'S+s)N$. Then $C\texttt{.count}$ is decremented and if it becomes 0, $c = \texttt{NOfPGrid}[p][s][g]$ is decremented to let it have the negative index of the next record. We also let the `nid` element of the moved particle -1 if $\mu < 0$[76], so that it will be skipped when it is revisited in `move_to_sendbuf_uw4p()` rather than mistakingly recognized as a staying particle because c can be 0 or be pointing a record with $t < 0$.

```
  if (dst==0)  { ACT; }\
  else if (MOVEIF) {\
    if (dst>0)  sb[NOfSend[dst-1]++] = *P;\
    else {\
      struct S_commlist *hs = CommList - (dst + 1);\
      if (hs->tag<0) {\
        npg[g] = 0;  ACT;\
      } else {\
        sb[NOfSend[hs->tag+hs->rid]++] = *P;\
        if (MOVEIF<0)  P->nid = -1;\
        if (--(hs->count)==0)  npg[g]--;\
      }\
    }\
  }\
}
```

[76] We can do this operation always but do it only when it is really necessary for the sake of comprehensiveness. This restriction gives us a small performance benefit avoiding unnecessary memory write, while the conditional operation should not cause any performance degradation because the condition `(MOVEIF<0)` is replaced with `(1<0)` or `(-1<0)` which reasonably smart compilers must eliminate (together with the assignment).

4.10.41 move_to_sendbuf_sec4p()

move_to_sendbuf_sec4p()

The function move_to_sendbuf_sec4p(), called solely from exchange_particles4p() when it finds $Q_n + P_n^{\rm send} > P_{lim}$ to mean we have to transfer particles among neighbors before sorting, is the position-aware counterpart of move_to_sendbuf_secondary() to move particles to be sent to SendBuf[] and pack those to stay in the local nodes in Particles[]. It is given arguments psold $= p_c = 1$ if the local node had a secondary subdomain or 0 otherwise, trans $= t \neq 0$ iff we have transitional state of helpand-helper configuration, oldp $= n^p_{old}$ being the local node n's helpand in the last step, nacc[2] $= \{Q^n_n, Q^{n^p_{new}}_n\}$, nsend $= P_n^{\rm send}$, and stats $\neq 0$ if we have to start new timing measurement.

Note that having n^p_{old} instead of n^p_{new} as an argument is essential for this function and its callees move_to_sendbuf_uw4p() and move_to_sendbuf_dw4p(). They refer to NOfPGrid[1][s][g] through the macro Move_Or_Do() to determine the fate of each secondary particles, i.e., staying in the local node n or being sent to another node. Since this map is corresponding to the secondary subdomain n^p_{old} in the last step, g has to be calculated based on n^p_{old}. Therefore, Move_Or_Do() recognizes that a particle in the subdomain n^p_{old} is possibly to stay in the local node even if $n^p_{old} \neq n^p_{new}$ due to helpand-helper reconfiguration, but the transfer schedule in NOfPGrid[1][][] definitely tells us all the secondary particles should go out from the local node. This also means that a particle traveling to the subdomain n^p_{new} being a helper of n^p_{old} and thus being accommodated possibly by the local node can be *sent* to the local node n itself. This subtle situation, however, should not cause any problems because in this situation RealDstNeighbors[2][1] should have n as a new helper of a neighbor of n's old helpand n^p_{old} and RealSrcNeighbors[2][1] should also have n as a old helper of a neighbor of n's new helpand n^p_{new}.

```
static void
move_to_sendbuf_sec4p(const int psold, const int trans, const int oldp,
                      const int *nacc, const int nsend, const int stats) {
  const int me=myRank, ns=nOfSpecies, sbase=specBase;
  const int ninj=nOfInjections, nplim=nOfLocalPLimit;
  int ninjp=0, ninjs=nplim, i;
  struct S_particle *sb = SendBuf,  *p;
  Decl_Grid_Info();
```

After starting the new timing measurement with the key STATS_TB_MOVE if required by stats $\neq 0$, we call set_sendbuf_disps4p() to build the index array of $sbuf(p, s, m)$ in NOfSend[p][s][m] for $m \in$ RealDstNeighbors[t][p] giving it t as its argument.

Then we scan injected particles whose amount is nOfInjections $= Q_n^{\rm inj}$ residing beyond the last $pbuf(p_c, S-1)$, i.e., from Particles[totalParts]. For each Particles[i] having non-negative nid, we extract its species s by Particle_Spec(). Then if Secondary_Injected() tells us that the particle was injected into or around the local node's secondary subdomain in the last step, we perform Primarize_Id_Only() to let the particle has $\sigma' = \sigma - (N + 3^D)$ in its subdomain code. Then we move it in bottom-up manner to the region from SendBuf[$P_n^{\rm send}$] to SendBuf[$P_{lim} - 1$], i.e., the empty region follwing $sbuf(1, S-1, N-1)$, if the injected subdomain m is n^p_{old} and it was scheduled to stay, or to $sbuf(p', s, m)$ otherwise by Move_Or_Do() which invokes Neighbor_Subdomain_Id() with $p = 1$ and refers to NOfPGrid[1][s][g].

Otherwise, i.e., if Secondary_Injected() is false to mean the particle was injected into or around the local node n's primary subomain, we do the same as the secondary case, but

we move the particles to the top half of the region from top if $m = n$ and scheduled to stay, and let `Move_Or_Do()` acts for a primary particle.

Note that in each case we count the number of particles injected into primary/secondary subdomain and staying in the local node to have $q_{\rm pri}^{\rm inj}$ and $q_{\rm sec}^{\rm inj}$ respectively. Therefore, injected and staying primary particles are moved to the region `SendBuf`$[P_n^{\rm send} + i]$ where $i \in [0, q_{\rm pri}^{\rm inj})$, while secondary particles are moved to `SendBuf`$[P_n^{\rm send} + P_{lim} - i]$ where $i \in [1, q_{\rm sec}^{\rm inj}]$. Also note that the region for the injected particles should be large enough because $P_{lim} \geq Q_n \geq P_n^{\rm send} + Q_n^{\rm inj} \geq P_n^{\rm send} + (q_{\rm pri}^{\rm inj} + q_{\rm sec}^{\rm inj})$.

```
if (stats) oh1_stats_time(STATS_TB_MOVE, 1);
set_sendbuf_disps4p(trans);

for (i=0,p=Particles+totalParts; i<ninj; i++,p++) {
  const int s = Particle_Spec(p->spec-sbase);
  const OH_nid_t nid = p->nid;
  const int ps = Secondary_Injected(nid) ? 1 : 0;
  dint *npg = NOfPGrid[ps][s];
  if (nid<0) continue;
  if (ps) {
    Primarize_Id_Only(p);
    Move_Or_Do(p, ps, oldp, 1, (sb[--ninjs]=*p));
  } else
    Move_Or_Do(p, ps, me  , 1, (sb[nsend+ninjp++]=*p));
}
```

Next we call `move_to_sendbuf_uw4p()` to move primary particles to be sent to `SendBuf`[] and to pack those to stay *upward* giving it arguments `ps` $= 0$ for primary particles, `mysd` $= n$ for the primary subdomain identifier, `cbase` $= 0$ to start the scan from $pbuf(0, 0)$, and `nbase` $= 0$ to mean the packed $pbuf(0, 0)$ will be also at `Particles`[0].

Then if $p_c \neq 0$ to mean we have secondary particles, we call `move_to_sendbuf_uw4p()` again but this time the arguments to be given are `ps` $= 1$ for secondary particles, `mysd` $= n_{old}^p$ for the secondary subdomain identifier, `cbase` $=$ `primaryParts` being Q_n^n in the last step to start the scan from $pbuf(1, 0)$, and `nbase` $=$ `nacc`[0] being Q_n^n in the next step to give the location of the packed $pbuf(1, 0)$. We also call its *downward* counterpart `move_to_sendbuf_dw4p()` with `ps` $= 1$ and `mysd` $= n_{old}^p$ too, and `ctail` $=$ `totalParts` being Q_n in the last step to show the tail of $pbuf(1, S-1)$, and `ntail` $=$ `nacc`[0] $+$ `nacc`[1] being Q_n in the next step and thus corresponding to the tail of the packed $pbuf(1, S-1)$.

Otherwise, i.e., if $p_c = 0$, we let;

$$\texttt{RecvBufBases}[1][s] = \texttt{Particles} + Q_n^n + \sum_{t=0}^{s-1} \texttt{TotalPNext}[1][t]$$

so that $rbuf(1, s) = pbuf(1, s)$.

Finally, we call `move_to_sendbuf_dw4p()` (again) for the primary particles, with `ps` $= 0$ and `mysd` $= n$ same as the upward counterpart, and `ctail` $=$ `primaryParts` and `ntail` $=$ `nacc`[0] being the tail of $pbuf(0, S-1)$ before and after packing respectively.

```
move_to_sendbuf_uw4p(0, me, 0, 0);
if (psold) {
  move_to_sendbuf_uw4p(1, oldp, primaryParts, nacc[0]);
  move_to_sendbuf_dw4p(1, oldp, totalParts, nacc[0]+nacc[1]);
} else {
```

```
    struct S_particle *rbb=Particles+nacc[0];
    int s;
    for (s=0; s<ns; s++) {
      RecvBufBases[ns+s] = rbb;  rbb += TotalPNext[ns+s];
    }
  }
  move_to_sendbuf_dw4p(0, me, primaryParts, nacc[0]);
```

Then we move back injected and staying primary ($p = 0$) and secondary ($p = 1$) particles whose amount is $q^{\rm inj}_{\rm pri}$ and $q^{\rm inj}_{\rm sec}$ respectively, from `SendBuf[]` to `Particles[]`. For each particle of species s, obtained by `Particle_Spec()`, we move them to the location pointed by `RecvBufBases`$[p][s]$ and increment `RecvBufBases`$[p][s]$ so that they are moved into $rbuf(p, s)$.

Finally, we let `primaryParts` and its shadow pointed by `secondaryBase` be Q^n_n in the next step.

```
  for (i=0,p=SendBuf+nsend; i<ninjp; i++,p++)
    *(RecvBufBases[Particle_Spec(p->spec-sbase)]++) = *p;
  for (i=ninjs,p=SendBuf+ninjs; i<nplim; i++,p++)
    *(RecvBufBases[Particle_Spec(p->spec-sbase)+ns]++) = *p;

  primaryParts = *secondaryBase = nacc[0];
}
```

4.10.42 `move_to_sendbuf_uw4p()`

`move_to_sendbuf_uw4p()` The function `move_to_sendbuf_uw4p()`, called solely from `move_to_sendbuf_sec4p()` once or twice, scans particles in the local node n's primary (`ps` $= p = 0$) or secondary ($p = 1$) subdomain `mysd` $= m$ from `Particles`$[b^-_0]$ where b^-_0 = `cbase` argument for $pbuf(p, 0)$ in the last step. It moves scanned particles to be sent to other nodes to `SendBuf[]`, and packs those to stay in the local node *upward*, i.e., to the direction of smaller indices of `Particles[]` and to the region beginning `Particles`$[b^+_0]$ where b^+_0 = `nbase` argument for the packed $pbuf(p, 0)$.

```
static void
move_to_sendbuf_uw4p(const int ps, const int mysd, const int cbase,
                     const int nbase) {
  const int ns=nOfSpecies;
  const int nsor0 = ps ? ns : 0;
  const int *ctp = TotalP + nsor0,  *ntp = TotalPNext + nsor0;
  struct S_particle *p,  **rbb = RecvBufBases + nsor0,  *sb = SendBuf;
  int s, c, d, cn, dn;
  Decl_Grid_Info();
```

We scan particles in each $pbuf(p, s)$ for all $s \in [0, S)$ and determine the direction of packing particles to stay in the local node as follows. Let us define b^-_s and b^+_s recursively as $b^-_{s+1} = b^-_s$ + `TotalP`$[p][s]$ and $b^+_{s+1} = b^+_s$ + `TotalPNext`$[p][s]$, where `TotalPNext`$[p][s] = |pbuf(p, s)|$ was accummulated by functions called by `make_send_sched()`, namely `make_send_sched_body()`, `scatter_hspot_send()` and `scatter_hspot_recv_body()`.

If $b^+_s \le b^-_s$, it is assured that the packing direction is upward because, for the particle being i-th and j-th in $pbuf(p, s)$ in the last and next step respectively, it is assured $j \le i$

and thus their indices $b_s^+ + j \leq b_s^- + i$. Therefore, we move all particles in $pbuf(p,s)$ from its top to bottom by `Move_Or_Do()` to let it move `Particles`$[b_s^- + i]$ to `Particles`$[b_s^+ + j]$ for particles to stay. After that, we let `RecvBufBases`$[p][s]$ point `Particles`$[b_s^+ + l]$ where l is the number of particles staying in $pbuf(p,s)$ so that $rbuf(p,s)$ is placed at the bottom of $pbuf(p,s)$ for the next step.

If $b_{s+1}^+ > b_{s+1}^-$, on the other hand, we can pack staying particles in $pbuf(p,s)$ by moving downward, because it is assured $b_{s+1}^+ - j > b_{s+1}^- - i$ for $j \leq i$. However we have to postpone it after packing $pbuf(p,s+1)$ and its successors. Therefore, we leave them to `move_to_sendbuf_dw4p()` but just let `RecvBufBases`$[p][s]$ point `Particles`$[b_s^+]$, i.e., the top of $pbuf(p,s)$ for the next step.

Otherwise, i.e., if $b_s^+ > b_s^-$ but $b_{s+1}^+ \leq b_{s+1}^-$, we have to pack the second half upward and then the first half downward. First we find the first staying particle being i_m-th and j_m-th in $pbuf(p,s)$ such that $b_s^+ + j_m \leq b_s^- + i_m$, by `Move_Or_Do()` letting it to move all particles to be sent to `SendBuf`[]. Note that this scan may make a hot-spot only have particles staying in the local node if any letting `NOfPGrid`$[p][s][g]$ in question be 0, so that those staying particles is correctly reconginzed as staying in this scan and the backward scan done afterward. At the same time, all scanned particles to be sent in the hot-spot are eliminated by letting their `nid` be -1 because the fourth argument of the macro is -1, in order to keep them from mistakingly recognized as staying particles in the backward scan.

Then we continue the scan from i_m and j_m to move all remaining particles in $pbuf(p,s)$ by `Move_Or_Do()` to let it move `Particles`$[b_s^- + i]$ to `Particles`$[b_s^+ + j]$ for particles to stay. After that, we let `RecvBufBases`$[p][s]$ point `Particles`$[b_s^+ + l]$ where l is the number of particles staying in $pbuf(p,s)$ so that $rbuf(p,s)$ is placed at the bottom of $pbuf(p,s)$ for the next step. Then finally, we scan the first half again from the bottom $i_m - 1$ and $j_m - 1$ to the top by `Move_Or_Do()` to let it move particles to stay downward, but to let it do nothing for particles with `NOfPGrid`$[p][s][g] \neq 0$ because they have already moved to `SendBuf`[] and `NOfPGrid`$[p][s][g] = 0$ for all staying particles including those in hot-spots.

```
    for (s=0,c=cbase,d=nbase; s<ns; s++,c=cn,d=dn) {
      dint *npg = NOfPGrid[ps][s];
      cn = c + ctp[s];  dn  = d + ntp[s];
      if (d<=c) {
        for (p=Particles+c; c<cn; c++,p++)
          Move_Or_Do(p, ps, mysd, 1, (Particles[d++]=*p));
        rbb[s] = Particles + d;
      } else if (dn>cn) {
        rbb[s] = Particles + d;
      } else {
        const int cb = c;
        int cm, dm;
        for (p=Particles+c; c<d; c++,p++)  Move_Or_Do(p, ps, mysd, -1, (d++));
        cm = c - 1;  dm = d - 1;
        for (p=Particles+c; c<cn; c++,p++)
          Move_Or_Do(p, ps, mysd, 1, (Particles[d++]=*p));
        rbb[s] = Particles + d;
        for (c=dm,d=dm,p=Particles+c; c>=cb; c--,p--)
          Move_Or_Do(p, ps, mysd, 0, (Particles[d--]=*p));
      }
    }
  }
```

4.10.43 move_to_sendbuf_dw4p()

move_to_sendbuf_dw4p() The function move_to_sendbuf_dw4p(), called solely from move_to_sendbuf_sec4p() once or twice, scans particles in the local node n's primary (ps $= p = 0$) or secondary ($p = 1$) subdomain mysd $= m$ from Particles[$b_S^- - 1$] where b_S^- = ctail argument for the element following $pbuf(p, S-1)$ in the last step. It moves scanned particles to be sent to other nodes to SendBuf[], and packs those to stay in the local node *downward*, i.e., to the direction of greater indices of Particles[] and to the region ending Particles[$b_S^+ - 1$] where b_S^+ = ntail argument for the element following the packed $pbuf(p, S-1)$.

```
static void
move_to_sendbuf_dw4p(const int ps, const int mysd, const int ctail,
                     const int ntail) {
  const int ns=nOfSpecies;
  const int nsor0 = ps ? ns : 0;
  const int *ctp = TotalP + nsor0,  *ntp = TotalPNext + nsor0;
  struct S_particle *sb = SendBuf,  *p;
  int s, c, d, cn, dn;
  Decl_Grid_Info();
```

We scan particles in $pbuf(p, s)$ such that $b_{s+1}^+ > b_{s+1}^-$ where b_s^- and b_s^+ are defined recursively as $b_{s+1}^- = b_s^- +$ TotalP[p][s] and $b_{s+1}^+ = b_s^+ +$ TotalPNext[p][s], or in other words $b_s^- = b_{s+1}^-$ − TotalP[p][s] and $b_s^+ = b_{s+1}^+$ − TotalPNext[p][s]. For such $pbuf(p, s)$, we can pack staying particles in it by moving downward, because it is assured $b_{s+1}^+ - j > b_{s+1}^- - i$ for a particle being i-th and j-th in $pbuf(p, s)$ from its tail in the last and next step respectively and thus $j \le i$. Therefore, we move all particles in $pbuf(p, s)$ from its bottom to top by Move_Or_Do() to let it move Particles[$b_{s+1}^- - i$] to Particles[$b_{s+1}^+ - j$] for particles to stay, i.e. those in grid-voxels at g such that NOfPGrid[p][s][g] = 0.

```
  cn = ctail;  dn = ntail;
  for (s=ns-1,c=cn-1,d=dn-1; s>=0; s--,c=cn-1,d=dn-1) {
    dint *npg = NOfPGrid[ps][s];
    cn -= ctp[s];  dn -= ntp[s];
    if (c>=d || cn>=dn)  continue;
    for (p=Particles+c; c>=cn; c--,p--)
      Move_Or_Do(p, ps, mysd, 1, (Particles[d--]=*p));
  }
}
```

4.10.44 move_and_sort_secondary()

move_and_sort_secondary() The function move_and_sort_secondary(), called solely from exchange_particles4p() when it finds $Q_n + P_n^{\text{send}} \le P_{lim}$ to mean we can move particles staying in and leaving from the local node together from Particles[] to SendBuf[] with sorting. It is given the following arguments; psold $= p_c = 1$ iff the local node had secondary particles; psnew $= p_n = 1$ iff it will have secondary particles; trans $= t \in \{0, 1\}$ being 1 iff we have transitional state of helpand-helper configuration; oldp $= n_{old}^p$ being the local node n's helpand in the last step; nacc[2] $= \{Q_n^n, Q_n^{n_{new}^p}\}$; and stats $\neq 0$ if we have to start new timing measurement. The reason why this function needs to have n_{old}^p instead of n_{new}^p is same as what we discussed in §4.10.41.

```
static void
move_and_sort_secondary(const int psold, const int psnew, const int trans,
                        const int oldp, const int *nacc, const int stats) {
  const int me=myRank, ns=nOfSpecies, nn=nOfNodes, sbase=specBase;
  const int mysubdom[2] = {me, oldp},  ninj = nOfInjections;
  struct S_particle *p,  *rbb,  *sb = SendBuf + nacc[0] + nacc[1];
  int *nofr;
  int ps, s, t, npt, i;
  Decl_For_All_Grid();
  Decl_Grid_Info();
```

After starting the new timing measurement with the key `STATS_TB_MOVE` if required by `stats` $\neq 0$, we call `set_sendbuf_disps4p()` to build the index array of $sbuf(p,s,m)$ in `NOfSend`$[p][s][m]$ for $m \in$ `RealDstNeighbors`$[t][p]$ giving it t as its argument.

Then we do the followings for all $p \in \{0, p_n\}$ (not $\{0, p_c\}$) and $s \in [0, S)$. First we let `RecvBufBases`$[p][s]$ be as follows.

$$\begin{aligned}
\mathcal{N}_S(n) &= \texttt{RealSrcNeighbors}[t][0].\texttt{nbor}[] \\
\mathcal{N}_S(n^p_{new}) &= \texttt{RealSrcNeighbors}[t][1].\texttt{nbor}[] \\
\texttt{RecvBufBases}[0][s] &= \texttt{Particles} + \sum_{s'=0}^{s-1} \sum_{m \in \mathcal{N}_S(n)} \texttt{NOfRecv}[0][s'][m] \\
\texttt{RecvBufBases}[1][s] &= \texttt{Particles} + \\
&\qquad \sum_{s'=0}^{S-1} \sum_{m \in \mathcal{N}_S(n)} \texttt{NOfRecv}[0][s'][m] + \sum_{s'=0}^{s-1} \sum_{m \in \mathcal{N}_S(n^p_{new})} \texttt{NOfRecv}[1][s'][m]
\end{aligned}$$

That is, we let $|rbuf(p,s)| = \sum_{m \in \mathcal{N}_s(n')} \texttt{NOfRecv}[p][s][m]$ where $n' = \{n, n^p_{new}\}[p]$, rank them contiguously from `Particles`$[0]$, and let `RecvBufBases`$[p][s]$ point the head of $rbuf(p,s)$. Note that we also let `RecvBufBases`$[1][S]$ be the value defined above so that `sort_received_particles()` refers to it pointing the entry following the tail of $rbuf(1, S-1)$.

Then we scan `NOfPGridOut`$[p][s][g]$ to build the index array for sorting in `NOfPGridTotal` $[p][s][g]$ as discussed in §4.10.37, by `For_All_Grid()` giving 0 to its first argument if $p = 0$, or $t+1$ if $p = 1$ so that it refers to appropriate element of `GridDesc`[].

```
  if (stats) oh1_stats_time(STATS_TB_MOVE, 1);
  set_sendbuf_disps4p(trans);
  for (ps=0,t=0,nofr=NOfRecv,rbb=Particles,npt=0; ps<=psnew; ps++) {
    const int nnbr = RealSrcNeighbors[trans][ps].n;
    const int *rnbr = RealSrcNeighbors[trans][ps].nbor;
    const int gdidx = ps ? trans+1 : 0;
    for (s=0; s<ns; s++,t++,nofr+=nn) {
      int n, nrec;
      dint *npgt = NOfPGridTotal[ps][s];
      const int *npgo = NOfPGridOut[ps][s];
      for (n=0,nrec=0; n<nnbr; n++)  nrec += nofr[rnbr[n]];
      RecvBufBases[t] = rbb;  rbb += nrec;
      For_All_Grid(gdidx, 0, 0, 0, 0, 0, 0) {
        const int np = npgo[The_Grid()];
        npgt[The_Grid()] = npt;  npt += np;
```

```
      }
    }
  }
  RecvBufBases[t] = rbb;
```

Then for all $p \in \{0, p_c\}$ and $s \in [0, S)$, we scan all particles in $pbuf(p, s)$ whose size is `TotalP`$[p][s]$ invoking `Move_Or_Do()` to move each of them in grid-voxel at g to `SendBuf`$[$`NOfPGridTotal`$[p][s][g]]$ and increment the index if it stays in the local node, or to `SendBuf`$[Q_n+$`NOfSend`$[p'][s]]$ and increment the index too where $p' = 0$ if the particle becomes primary of m or $p' = 1$ otherwise, according to `NOfPGrid`$[p][s][g]$.

```
  for (ps=0,p=Particles,t=0; ps<=psold; ps++) {
    const int mysd = mysubdom[ps];
    for (s=0; s<ns; s++,t++) {
      dint *npg = NOfPGrid[ps][s],  *npgt = NOfPGridTotal[ps][s];
      const int itail = TotalP[t];
      for (i=0; i<itail; i++,p++)
        Move_Or_Do(p, ps, mysd, 1, (SendBuf[npgt[g]++]=*p));
    }
  }
```

We continue the scan for injected particles whose amount is `nOfInjections` $= Q_n^{\rm inj}$ residing beyond the last $pbuf(p_c, S-1)$. For each `Particles`$[i]$ having non-negative `nid` element, we extract its species s by `Particle_Spec()` and examines if it was injected into or around the local node's primary ($p = 0$) or secondary ($p = 1$) subdomain in the last step by `Secondary_Injected()`, performing `Primarize_Id_Only()` to let the particle has $\sigma' = \sigma - (N + 3^D)$ in its subdomain code in the secondary case. Then we move the particle to `SendBuf`$[]$ by `Move_Or_Do()` giving it n ($p = 0$) or n^p_{old} ($p = 1$) for the subdomain identifier and letting it refer to `NOfPGrid`$[p][s][g]$ and `NOfPGridTotal`$[p][s][g]$ where g is the grid-position of the particle.

Finally, we let `primaryParts` and its shadow pointed by `secondaryBase` be Q_n^n in the next step.

```
  for (i=0; i<ninj; i++,p++) {
    const int s = Particle_Spec(p->spec-sbase);
    const OH_nid_t nid = p->nid;
    const int ps = Secondary_Injected(nid) ? 1 : 0;
    const int mysd = mysubdom[ps];
    dint *npg = NOfPGrid[ps][s],  *npgt = NOfPGridTotal[ps][s];
    if (nid<0) continue;
    if (ps)  Primarize_Id_Only(p);
    Move_Or_Do(p, ps, mysd, 1, (SendBuf[npgt[g]++]=*p));
  }
  primaryParts = *secondaryBase = nacc[0];
}
```

4.10.45 `set_sendbuf_disps4p()`

`set_sendbuf_disps4p()` The function `set_sendbuf_disps4p()`, called from `move_to_sendbuf_sec4p()` and `move_and_sort_secondary()` prior to their particle scan, is the position-aware counterpart of `set_sendbuf_disps()` to build the index array for `SendBuf`$[]$ in `NOfSend`$[][][]$ based on the sending counts in itself. The function is given an argument `trans` $= t \in \{0, 1\}$ being 1

iff we have transitional state of helpand-helper configuration and thus we need to refer to `RealDstNeighbors`[1][].

The function lets `NOfSend`[p][s][m] as follows so that $sbuf(p,s,m)$ are ranked in `SendBuf`[] contiguously and $|sbuf(p,s,m)|$ is the original value of `NOfSend`[p][s][m] which we refer to as $q^{\mathrm{send}}(p,s,m)$;

$$\texttt{NOfSend}[p][s][m_i] = \sum_{q=0}^{p-1}\sum_{t=0}^{S-1}\sum_{j=0}^{c(q)-1} q^{\mathrm{send}}(q,t,m_j(q)) + \sum_{t=0}^{s-1}\sum_{j=0}^{c(p)-1} q^{\mathrm{send}}(p,t,m_j(p)) + \sum_{j=0}^{i-1} q^{\mathrm{send}}(p,s,m_j(p))$$

where;

$$\begin{aligned} m \in \mathcal{N}_D(n(p)) &= \{m_0(p),\ldots,m_{c(p)-1}(p)\} = \texttt{RealDstNeighbors}[t][p]\texttt{.nbor}[] \\ n(p) &= \{n, n^p_{old}\}[p] \\ c(p) &= |\mathcal{N}_D(n(p))| = \texttt{RealDstNeighbors}[t][p]\texttt{.n} \end{aligned}$$

Roughly speaking, the equation above is calculated recursively by;

$$\texttt{NOfSend}[p][s][m_i] = \texttt{NOfSend}[p][s][m_{i-1}] + q^{\mathrm{send}}(p,s,m_{i-1})$$

if we may consider that $\texttt{NOfSend}[p][s][m_{-1}(p)] = \texttt{NOfSend}[p][s-1][m_{c(p)-1}(p)]$ and so on.

```
static void
set_sendbuf_disps4p(const int trans) {
  const int nn=nOfNodes, ns=nOfSpecies;
  int ps, s, i, np, *sbd;

  for (ps=0,sbd=NOfSend,np=0; ps<2; ps++) {
    const int n = RealDstNeighbors[trans][ps].n;
    const int *nbor = RealDstNeighbors[trans][ps].nbor;
    for (s=0; s<ns; s++,sbd+=nn) {
      for (i=0; i<n; i++) {
        const int nid = nbor[i];
        const int nsend = sbd[nid];
        sbd[nid] = np;  np += nsend;
      }
    }
  }
}
```

4.10.46 `xfer_particles()`

`xfer_particles()` The function `xfer_particles()`, called solely from `exchange_particles4p()` in two cases with $Q_n + P_n^{\mathrm{send}} \leq P_{lim}$ or not, sends particles in the local node to other nodes in `RealDstNeighbors`[t][] and receives particles from other nodes in `RealSrcNeighbors`[t][], where $t = \texttt{trans} \in \{0,1\}$ argument being 1 iff we have transitional state of helpand-helper configuration. The other arguments are $\texttt{psnew} = p_n \in \{0,1\}$ being 1 iff the local node will have secondary subdomain in the next step and thus may have some secondary particles to receive, and `sbuf` is the pointer to `SendBuf`[0] or `SendBuf`[Q_n] to specify the location of $sbuf(0,0,0)$.

```
static void
xfer_particles(const int trans, const int psnew, struct S_particle *sbuf) {
  const int nn=nOfNodes, ns=nOfSpecies;
  int ps, s, t, i, req, sdisp, *nofr, *nofs;
```

First, we post `MPI_Irecv()` to receive particles, whose amount is $\texttt{NOfRecv}[p][s][m_i] > 0$ and data-type is `T_Particle`, from the node $m_i = \texttt{RealSrcNeighbors}[t][p]\texttt{.nbor}[i]$ for all $p \in \{0, p_n\}$, $s \in [0, S)$ and $i \in [0, c(p))$ where $c(p) = \texttt{RealSrcNeighbors}[t][p]\texttt{.n}$. The location of the receiving buffer is in $rbuf(p, s)$ in which particles received from all sender nodes are ranked contiguously, and thus it is from $\texttt{RecvBufBases}[p][s] + \sum_{j=0}^{i-1} \texttt{NOfRecv}[p][s][j]$. The tag of the communication is $pS + s$ so that coupling it with m_i makes each communication unique.

```
  for (ps=0,t=0,nofr=NOfRecv,req=0; ps<=psnew; ps++) {
    const int n = RealSrcNeighbors[trans][ps].n;
    const int *nbor = RealSrcNeighbors[trans][ps].nbor;
    for (s=0; s<ns; s++,t++,nofr+=nn) {
      struct S_particle *rbuf = RecvBufBases[t];
      for (i=0; i<n; i++) {
        const int nid = nbor[i];
        const int nrecv = nofr[nid];
        if (nrecv) {
          MPI_Irecv(rbuf, nrecv, T_Particle, nid, t, MCW, Requests+req++);
          rbuf += nrecv;
        }
      }
    }
  }
```

Next, we post `MPI_Isend()` to send particles of `T_Particle` to the nodes $m_i = \texttt{RealDstNeighbors}[t][p]\texttt{.nbor}[i]$ for all $p \in \{0, 1\}$, $s \in [0, S)$ and $i \in [0, c(p))$ where $c(p) = \texttt{RealDstNeighbors}[t][p]\texttt{.n}$. The sending buffer is $sbuf(p, s, m_i)$ whose offset from $sbuf(0, 0, 0)$ is $\texttt{NOfSend}[p][s][m_{i-1}]$ if we consider $[p][s][m_{-1}]$ is $[p][s-1][m_{c(p)-1}]$ etc., and since $sbuf(p, s, m_i)$ are ranked contiguously, the amount of sending particles is $\texttt{NOfSend}[p][s][m_i] - \texttt{NOfSend}[p][s][m_{i-1}]$. The tag of the communication is $pS + s$ again to ensure the uniqueness of the communication. Note that we let $\texttt{NOfSend}[p][s][m_i] = 0$ for the use of the next step.

```
  for (ps=0,t=0,sdisp=0,nofs=NOfSend; ps<2; ps++) {
    const int n = RealDstNeighbors[trans][ps].n;
    const int *nbor = RealDstNeighbors[trans][ps].nbor;
    for (s=0; s<ns; s++,t++,nofs+=nn) {
      for (i=0; i<n; i++) {
        const int nid = nbor[i];
        const int sdnxt = nofs[nid];
        const int nsend = sdnxt - sdisp;
        nofs[nid] = 0;
        if (nsend) {
          MPI_Isend(sbuf+sdisp, nsend, T_Particle, nid, t, MCW,
                    Requests+req++);
        }
        sdisp = sdnxt;
```

```
      }
    }
  }
```

Finally, we confirm the completions of all `MPI_Irecv()` and `MPI_Isend()` by `MPI_Waitall()` recorded in `Requests[]` to obtain their completion status in `Statuses[]` (but not referring to).

```
  MPI_Waitall(req, Requests, Statuses);
}
```

4.10.47 Macro `Check_Particle_Location()`

`Check_Particle_Location()` The macro `Check_Particle_Location`(π, p, s, S, i) checks the consistency of the arguments π, p and s given to its invoker `oh4p_map_particle_to_neighbor()`, `oh4p_map_particle_to_subdomain()` or `oh4p_remove_mapped_particle()`, if `OH_NO_CHECK` is not `#define`d.

It always checks if $p \in \{0, 1\}$ and $s \in [0, S)$. Then if `PbufIndex` $\neq$ `NULL` to mean `transbound4p()` has already been called to give us meaningful values in it and `totalParts`, we further check the following consistencies. For injected particles ($i \neq 0$), we checks that $\mathit{parent}(n) =$ `RegionId`$[1] \geq 0$ if $p = 1$ and that π points a location not beyond `Particles`$[Q_n + Q_n^{\mathrm{inj}}]$, i.e., $\pi <$ `Particles` + `totalParts` + `nOfInjections`. For ordinary particles ($i = 0$) on the other hand, it checks π points a location in $\mathit{pbuf}(p, s)$, i.e.

$$\texttt{Particles} + \texttt{PbufIndex}[p][s] \leq \pi < \texttt{Particles} + \texttt{PbufIndex}[p][s{+}1]$$

where we consider `PbufIndex`$[p][S] =$ `PbufIndex`$[p{+}1][0]$. Then if one or more examinations fail, we abort the execution by `local_errstop()` showing the particle index $\pi -$ `Particles` and values of p and s.

If `OH_NO_CHECK` is `#define`d, on the other hand, the macro replacement gives us nothing.

```
#ifndef OH_NO_CHECK
#define Check_Particle_Location(P, PS, S, NS, INJ) {\
  const int t = (PS) ? (S)+(NS) : (S);\
  const int pidx = (P) - Particles;\
  if ((PS)<0 || (PS)>1 || (S)<0 || (S)>=(NS) ||\
      (PbufIndex && ((INJ) ?\
                     (((PS)&&RegionId[1]<0) ||\
                         pidx>=totalParts+nOfInjections) :\
                     (pidx<PbufIndex[t] || pidx>=PbufIndex[t+1])))\
    local_errstop("'part' argument pointing %c%d%c of the particle buffer is "\
                  "inconsistent with 'ps'=%d and 's'=%d",\
                  specBase?'(':'[', pidx+specBase,\
                  specBase?')':']', PS, (S)+specBase);\
}
#else
#define Check_Particle_Location(P, PS, S, NS, INJ)
#endif
```

4.10.48 Macros `Map_Particle_To_Neighbor()` and `Adjust_Neighbor_Grid()`

`Map_Particle_To_Neighbor()` The macro `Map_Particle_To_Neighbor`$(\pi, X_d, d, m, k, 3^d, \delta_d(m), x_d, g)$, used solely in `oh4p_map_particle_to_neighbor()` D-times, examines the d-th dimensional coordinate X_d of the position of particle π in the subdomain m or its exterior to find the subdomain in which π resides. It updates the neighbor index k by $\pm 3^d$ if π is in d-th dimensional exterior. It also calculates d-th dimensional integer coordinate x_d local to m for X_d, and updates grid-voxel's *partial* index g by adding x_d, and then convert x_d to the local coordinate of the subdomain in whch π resides if it is in the d-th dimensional upper exterior of m.

First, we calculate $x'_d = (X_d - \Delta^l_d \cdot \gamma_d)/\gamma_d + \Delta^l_d$, where $\Delta^l_d \cdot \gamma_d =$ `Grid`$[d]$`.fcoord`$[0]$, $1/\gamma_d =$ `Grid`$[d]$`.rgsize`, and $\Delta^l_d =$ `Grid`$[d]$`.coord`$[0]$, to have the global integer coordinate x'_d for X_d. Then we make the following correction for the floating point calculation error so that $x'_d\gamma_d \le X_d < (x'_d + 1)\gamma_d$, where $\gamma_d =$ `Grid`$[d]$`.gsize`.

$$x'_d \leftarrow \begin{cases} x'_d - 1 & X_d < x'_d\gamma_d \\ x'_d & x'_d\gamma_d \le X_d < (x'_d+1)\gamma_d \\ x'_d + 1 & (x'_d+1)\gamma_d \le X_d \end{cases}$$

Then we have the local coordinate x_d for m by subtracting $\delta^l_n(m) =$ `SubDomains`$[m][d][0]$ from x'_d, i.e. $x_d = x'_d - \delta^l_d(m)$, and update the partial index $g = gidx(0, x_{d+1}, \ldots)$ by adding x_d to it, i.e., $g \leftarrow g + x_d = gidx(x_d, x_{d+1}, \ldots)$.

Next if $x_d < 0$ to mean π is in the d-th dimensional lower exterior of m, we update $k = \sum_{e=0}^{d} 3^e + \sum_{e=d+1}^{D-1} \nu_e 3^e$ by subtracting 3^d from it to have $k \leftarrow k - 3^d = \sum_{e=0}^{d-1} 3^e + \sum_{e=d}^{D-1} \nu_e 3^d$ where $\nu_d = 0$ for d-th dimensional lower neighbors. We also check if $X_d < \Delta^l_d \cdot \gamma_d$ and, if so, confirm that the d-th dimensional lower system boundary condition is periodic, i.e., `Boundaries`$[m][d][0] = 0$, to let $X_d \leftarrow X_d + (\Delta^u_d - \Delta^l_d)\gamma_d$ where $\Delta^u_d \cdot \gamma_d =$ `Grid`$[d]$`.fcoord`$[1]$. If this examination fails to mean π is crossing a non-periodic system boundary, we make π eliminated by letting its `nid` be -1 and force `oh4p_map_particle_to_neighbor()` to return to its caller. We also confirm $x_d \ge -e^g$, and if unsatisfied we let $k = -3^D$ to let `oh4p_map_particle_to_neighbor()` to consult `oh4p_map_particle_to_subdomain()` to have the subdomain to which π warped. Note that letting $k = -3^D$ makes it sure that the series of the invocations of this macro results $k < 0$ because other invocations add at most $3^D - 2$ to k.

If $x_d \ge \delta_d(m)$, on the other hand, to mean π is in the d-th dimensional upper exterior of m, we let $k \leftarrow k + 3^d = \sum_{e=0}^{d-1} 3^e + \sum_{e=d}^{D-1} \nu_e 3^d$ where $\nu_d = 2$ for d-th dimensional upper neighbors. We also check if $X_d \ge \Delta^u_d \cdot \gamma_d$ and, if so, confirm that the d-th dimensional upper system boundary condition is periodic, i.e., `Boundaries`$[m][d][1] = 0$, to let $X_d \leftarrow X_d - (\Delta^u_d - \Delta^l_d)\gamma_d$. If this examination fails to mean π is crossing a non-periodic system boundary, we let π`.nid` $= -1$ to mean π has gone away and force `oh4p_map_particle_to_neighbor()` return to its caller with -1. Then, unlike the lower boundary case, we let $x_d \leftarrow x_d - \delta_d(m)$ to make x_d local to the neighbor because this conversion can be done without knowing the neighbor subdomain's shape. We also confirm, like lower boundary case, $x_d < e^g$, and if unsatisfied we let $k = -3^D$ for consulting `oh4p_map_particle_to_subdomain()` too.

```
#define Map_Particle_To_Neighbor(P, XYZ, DIM, MYSD, K, INC, UB, G, IDX) {\
  const double xyz = XYZ;\
  const double gsize = Grid[DIM].gsize;\
  const double lb = Grid[DIM].fcoord[OH_LOWER];\
  const double gf =\
    (G = (xyz-lb)*Grid[DIM].rgsize + Grid[DIM].coord[OH_LOWER]) * gsize;\
```

```
    if (gf>xyz) G--;\
    else if (gf+gsize<=xyz) G++;\
    G -= SubDomains[MYSD][DIM][OH_LOWER];  IDX += G;\
    if (G<0) {\
      K -= INC;\
      if (xyz<lb) {\
        if (Boundaries[MYSD][DIM][OH_LOWER]) { P->nid = -1;  return(-1); }\
        XYZ += Grid[DIM].fcoord[OH_UPPER] - lb;\
      }\
      if (G<-OH_PGRID_EXT)  K = -OH_NEIGHBORS;\
    } else if (G>=UB) {\
      double ub = Grid[DIM].fcoord[OH_UPPER];\
      K += INC;\
      if (xyz>=ub) {\
        if (Boundaries[MYSD][DIM][OH_UPPER]) { P->nid = -1;  return(-1); }\
        XYZ -= ub - lb;\
      }\
      G-=UB;\
      if (G>=OH_PGRID_EXT)  K = -OH_NEIGHBORS;\
    }\
  }
```

Adjust_Neighbor_Grid()

The macro `Adjust_Neighbor_Grid`(x_d, m, d), used solely in `oh4p_map_particle_to_neighbor()` D-times, transform the d-th dimensional local coordinate $x_d < 0$ of the local node's primary/secondary subdomain into that of its neighbor m. That is, if $x_d < 0$ we let $x_d \leftarrow x_d + \delta_d(m) = x_d + (\delta_d^u(m) - \delta_d^l(m))$ where $\delta_d^\beta(m) =$ `SubDomains`$[m][d][\beta]$.

```
#define Adjust_Neighbor_Grid(G, N, DIM)\
  if (G<0) G += SubDomains[N][DIM][OH_UPPER]-SubDomains[N][DIM][OH_LOWER];
```

4.10.49 oh4p_map_particle_to_neighbor()

oh4p_map_particle_to_neighbor_()
oh4p_map_particle_to_neighbor()

The API functions `oh4p_map_particle_to_neighbor_()` for Fortran and `oh4p_map_particle_to_neighbor()` for C find the subdomain m, which is primary (`ps` $= p = 0$) or secondary ($p = 1$) subdomain of the local node n or its neighbor, should accommodate the particle of species `s` $= s$ in `Particles`[] and pointed by the argument pointer `part` $= \pi$, and return m or -1 if particle went out-of-bounds. The function for C is also called from `oh4p_inject_particle()` and `oh4p_remap_particle_to_subdomain()` to find the subdomain for a particle injected and mapped again respectively.

They have a number of differences from their level-3 couterparts `oh3_map_particle_to_neighbor[_]()`. First, they receive the `S_particle` structure pointer π rather than their position coordinate values. Second, they need species s to refer to an element of `NOfPLocal`[][][][], mantaining it by the functions rather than the simulator body to bring another difference, and `NOfPGrid`[][][][] for which we add calculations of integer coordinate values and the grid-position of the particle. Third, they accept particles causing anywhere accommodation by calling `oh4p_map_particle_to_subdomain()` if the particles travel to outside of the exterior of the primary/secondary subdomain.

The function `oh4p_map_particle_to_neighbor_()` simply calls its counterpart `oh4p_map_particle_to_neighbor()`, but decrementing `s` by 1 to make it zero-origined.

```
int
oh4p_map_particle_to_neighbor_(struct S_particle *part, const int *ps,
                               const int *s) {
  return(oh4p_map_particle_to_neighbor(part, *ps, *s-1));
}
int
oh4p_map_particle_to_neighbor(struct S_particle *part, const int ps,
                              const int s) {
  const int ns = nOfSpecies,  inj = part>=Particles+totalParts;
  int x, y, z, w, d, dw, mysd;
  const int psnn = ps ? (s+nOfSpecies)*nOfNodes : s*nOfNodes;
  int k = OH_NBR_SELF,  idx = 0;
  int gz, gy, gx;
  int sd;
  Decl_Grid_Info();
```

First, we invoke `Check_Particle_Location()` to check the consistency of arguments, before referring to elements in `GridDesc`[] and `RegionId`[]. The fifth argument of the macro i is determined by whether π is beyond `Particles`+`nOfInjections`, i.e., π is for an injected particle. Then, we invoke `Map_Particle_To_Neighbor()` D-times for $d = D-1$ down to $d = 0$, with `Do_Z()` and `Do_Y()` to skip invocations if $D < 3$ or $D < 2$ respectively. To the macro, we give the following arguments.

- the particle pointer π.
- d-th dimensional coordinate X_d of π's position.
- the dimension d.
- local node's subdomain identifier $n' = \{n, parent(n)\}[p] = \texttt{RegionId}[p]$.
- neighbor index k initialized to be $\lfloor 3^D/2 \rfloor$ for the subdomain n' itself.
- 3^d to update k to let it have the neighbor index for the target subdomain m at last.
- d-th dimensional size of n' being $\delta_d(n') = \texttt{GridDesc}[p].\{\texttt{x},\texttt{y},\texttt{z}\}[d]$.
- the variable x_d to have the local coordinate value.
- grid-position g intitalized to be 0 and multiplied by $\delta^{\max}_{d-1} + 4e^g = \texttt{GridDesc}[p].\{\texttt{w},\texttt{d}\}[d-1]$ after the invocation to have $g = gidx(x_0, \ldots)$ for n' at last.

```
  Check_Particle_Location(part, ps, s, ns, inj);
  x = GridDesc[ps].x;  y = GridDesc[ps].y;  z = GridDesc[ps].z;
  w = GridDesc[ps].w;  d = GridDesc[ps].d;  dw = GridDesc[ps].dw;
  mysd = RegionId[ps];
  Do_Z(Map_Particle_To_Neighbor(part, part->z, OH_DIM_Z, mysd, k, 9, z, gz,
                                idx));
  Do_Z(idx *= d);
  Do_Y(Map_Particle_To_Neighbor(part, part->y, OH_DIM_Y, mysd, k, 3, y, gy,
                                idx));
  Do_Y(idx *= w);
  Map_Particle_To_Neighbor(part, part->x, OH_DIM_X, mysd, k, 1, x, gx, idx);
```

Now we have the neighbor index k, being most likely $\lfloor 3^D/2 \rfloor$ to mean π stays in the subodomain n' it has resided. If so, g can be used both as the index of `NOfPGrid`[][][] and as the grid-position part of π's `nid`. Moreover, we know the subdomain is n' for `NOfPLocal`[][][] and for the return value without looking up `AbsNeighbors`[][]. Therefore, we quickly perform necessary operations before returning to the caller with n'; incrementing `NOfPGrid`$[p][s][g]$ and `NOfPLocal`$[p][s][n']$; letting `nid` be $k \cdot 2^\Gamma + g$ by `Combine_Subdom_Pos()`. The other operations we have to do before returning is to increment `InjectedParticles`$[0][p][s]$ if the particle is injected, because we need it maintained for non-position-aware particle transfer. In addition we let the subdomain code of `nid` be $k+N+3^D$ if $p = 1$ by `Secondarize_Id()` to tell its secondariness to the functions called from `transbound4p()`.

On the other hand, one of `Map_Particle_To_Neighbor()` may let $k = -3^D$ to make final k be negative to mean the particle warped outside the exterior of the local node's primary/secondary subdomain. If so, we consult `oh4p_map_particle_to_subdomain()` to have the subdomain of the particle.

```
if (k==OH_NBR_SELF) {
  NOfPGrid[ps][s][idx]++;
  NOfPLocal[psnn+mysd]++;
  part->nid = Combine_Subdom_Pos(k, idx);
  if (inj) {
    if (ps) {
      InjectedParticles[ns+s]++;  Secondarize_Id(part);
    } else {
      InjectedParticles[s]++;
    }
  }
  return(mysd);
} else if (k<0)
  return(oh4p_map_particle_to_subdomain(part, ps, s));
```

Otherwise, i.e., if $k \geq 0$ but $k \neq \lfloor 3^D/2 \rfloor$, we have to consult `AbsNeighbors`[][] to have $m =$ `AbsNeighbors`$[p][k]$. Then if $m \geq N$ to indicate the neighbor does not exist, we make π eliminated and return to the simulator body with -1. Otherwise we continue to let $x_d \leftarrow x_d + \delta_d(m)$ for all $d \in [0, D)$ by `Adjust_Neighbor_Grid()` if $x_d < 0$, to have the local coordinates in m. We also maintain the per-subdomain population histogram by incrementing `NOfPLocal`$[p][s][m]$.

```
sd = AbsNeighbors[ps][k];
if (sd>=nOfNodes) {
  part->nid = -1;  return(-1);
}
Adjust_Neighbor_Grid(gx, sd, OH_DIM_X);
Do_Y(Adjust_Neighbor_Grid(gy, sd, OH_DIM_Y));
Do_Z(Adjust_Neighbor_Grid(gz, sd, OH_DIM_Z));
NOfPLocal[psnn+sd]++;
```

Finally, with g being the grid-voxel index in the exterior of n' and x_d for $d \in [0, D)$ being the grid-voxel coordinate in m, we increment `NOfPGrid`$[p][s][g]$ and let π's `nid` element be $k \cdot 2\Gamma + gidx(x_0, \ldots)$ by `Coord_To_Index()` and `Combine_Subdom_Pos()`. However, if $m = n'$ meaning n' has itself as its neighbor and π travels to the neighbor subdomain, π must be counted as a member in the grid-voxel at $gidx(x_0, \ldots)$ rather than at g and π must have

$\lfloor 3^D/2 \rfloor$ in the subdomain code of its `nid`, to avoid complication due to $g \neq gidx(x_0, \ldots)$ in n'. This also solves relatively minor problems by assuring that we will not make self-communication for particles staying a subdomain eventually and that we will not have any particles in the exterior of n' if the corresponding neighbor is n' itself. The other operation we have to do for $m = n'$ case is to increment `InjectedParticles`$[p][s]$ if π is an injected particle as discussed before.

Then really finally, we return to the caller with m, after performing `Secondarize_Id()` if the particle is injected and $p = 1$ as discussed before.

```
    if (sd==mysd) {
      idx = Coord_To_Index(gx, gy, gz, w, dw);
      NOfPGrid[ps][s][idx]++;
      part->nid = Combine_Subdom_Pos(OH_NBR_SELF, idx);
      if (inj)  InjectedParticles[ps ? ns+s : s]++;
    } else {
      NOfPGrid[ps][s][idx]++;
      part->nid = Combine_Subdom_Pos(k, Coord_To_Index(gx, gy, gz, w, dw));
    }
    if (inj && ps)  Secondarize_Id(part);
    return(sd);
  }
```

4.10.50 Macros `Map_To_Grid`, `Map_Particle_To_Subdomain()` and `Local_Coordinate()`

`Map_To_Grid()` The macro `Map_To_Grid`$(\pi, X^*_d, X_d, d, x_d, x'_d)$, used solely in `oh4p_map_particle_to_subdomain()` D-times, examines the d-th dimensional coordinate X^*_d of the position of particle π currently accommodated by the local node and copy it to a local variable X_d. It calculates d-th dimensional global coordinate x_d of the grid-voxel in which π resides, and its *raw* value x'_d without taking care of periodic system boundary if any.

First, we examine if $X_d < \Delta^l_d \cdot \gamma_d =$ `Grid`$[d]$`.fcoord`$[0]$ or $X_d \geq \Delta^u_d \cdot \gamma_d =$ `Grid`$[d]$`.fcoord`$[1]$ to mean π has crossed a system boundary. If so, we confirm the d-th dimensional system boundary condition is periodic, i.e., `BoundaryCondition`$[d][\beta] = 0$ for corresponding $\beta \in \{0, 1\}$, or make π eliminated and force `oh4p_map_particle_to_subdomain()` to return its caller with -1. Then we let X_d and X^*_d be $X_d \pm (\Delta^u_d - \Delta^l_d)\gamma_d$, and let $b_d = \mp(\Delta^u_d - \Delta^l_d)$ where $\Delta^\beta_d =$ `Grid`$[d]$`.coord`$[\beta]$ to regain the *raw* value of x_d by $x'_d = x_d + b_d$ after we calculate x_d.

Next, we calculate $x_d = (X_d - \Delta^l_d \cdot \gamma_d)/\gamma_d + \Delta^l_d$, where $1/\gamma_d =$ `Grid`$[d]$`.rgsize`, and make the corection on it as discussed in §4.10.48 so that $x_d\gamma_d \leq X_d < (x_d + 1)\gamma_d$, where $\gamma_d =$ `Grid`$[d]$`.gsize`. Finally, we let $x'_d = x_d + b_d$ where $b_d = \mp(\Delta^u_d - \Delta^l_d)$ if π has crossed a system boundary as discussed above, or $b_d = 0$ otherwise.

```
#define Map_To_Grid(P, PXYZ, XYZ, DIM, GG, LG) {\
  const double gsize = Grid[DIM].gsize;\
  const double lb = Grid[DIM].fcoord[OH_LOWER];\
  const double ub = Grid[DIM].fcoord[OH_UPPER];\
  double gf;\
  XYZ = PXYZ;\
  LG = 0;\
  if (XYZ<lb) {\
    if (BoundaryCondition[DIM][OH_LOWER]) { P->nid = -1;  return(-1); }\
```

```
    XYZ += (ub - lb);  PXYZ = XYZ;\
    LG = Grid[DIM].coord[OH_LOWER] - Grid[DIM].coord[OH_UPPER];\
  }\
  else if (XYZ>=ub) {\
    if (BoundaryCondition[DIM][OH_UPPER]) { P->nid = -1;  return(-1); }\
    XYZ -= (ub - lb);  PXYZ = XYZ;\
    LG = Grid[DIM].coord[OH_UPPER] - Grid[DIM].coord[OH_LOWER];\
  }\
  GG = (XYZ-lb)*Grid[DIM].rgsize + Grid[DIM].coord[OH_LOWER];\
  gf = GG * gsize;\
  if (gf>XYZ) GG--;\
  else if (gf+gsize<=XYZ) GG++;\
  LG += GG;\
}
```

`Map_Particle_To_Subdomain()` The macro `Map_Particle_To_Subdomain`(x_d, d, π_d), used solely in `oh4p_map_particle_to_subdomain()` D-times if we have regular process coordinate, calculates d-th dimensional process (subdomain) coordinate value π_d in which a particle at d-th dimensional integer coordinate x_d resides as;

$$\pi_d \leftarrow \begin{cases} \lfloor (x_d - \Delta_d^l)/\delta_d^{\min} \rfloor & x < \Delta_d^- \\ \Pi_d^- + \lfloor (x_d - \Delta_d^-)/(\delta_d^{\min} + 1) \rfloor & x \geq \Delta_d^- \end{cases}$$

referring to;

$$\texttt{Grid}[d]\texttt{.coord}[0] = \Delta_d^l$$
$$\texttt{Grid}[d]\texttt{.light.}\{\texttt{size,thresh,n}\} = \{\delta_d^{\min},\ \Delta_d^-,\ \Pi_d^-\}$$

in a similar way we discussed in §4.7.19 but with integer coordinate and parameters. Since this integer arithmetic is definately accurate, we don't need any corrections which level-3 counterparts does with `Adjust_Subdomain()`.

```
#define Map_Particle_To_Subdomain(XYZ, DIM, SDOM) {\
  double thresh = Grid[DIM].light.thresh;\
  if (XYZ<thresh)\
    SDOM = (XYZ - Grid[DIM].coord[OH_LOWER]) / Grid[DIM].light.size;\
  else\
    SDOM = (XYZ - thresh)/ (Grid[DIM].light.size + 1) + Grid[DIM].light.n;\
}
```

`Local_Coordinate()` The macro `Local_Coordinate`$(m, n', x_d, x'_d, d, k, 3^d, a)$, used solely in `oh4p_map_particle_to_subdomain()` D-times, converts d-th dimensional global coordinate value x_d and its *raw* counterpart x'_d into the correspoinding local coordinate in m and in local node's own n' respectively. It also updates the neighbor index k by $\pm 3^d$ if x_d is in d-th dimensional exteriorof n'. The conversion is basically done by $x_d \leftarrow x_d - \delta_d^l(m)$ and $x'_d \leftarrow x'_d - \delta_d^l(n')$ where $\delta_d^l(m) = \texttt{SubDomains}[m][d][0]$, but we have to take care of the case $m = n'$ and $x_d \neq x'_d$. This surprising combination can happen when a particle has crossed a periodic system boundary and $m = n'$ is the sole subdomain between exiting and entering boundaries, i.e., both of d-th dimensional boundary planes of n' are also system boundary planes. Therefore if $m = n'$, we let $x'_d = x_d$ for referring `NOfPGrid`$[][][]$ by $gidx(\ldots, x'_d, \ldots)$ because of the reason we discussed in §4.10.49. Otherwise, i.e., if $m \neq n'$ and thus the particle

is not in the interior of n', we confirm $-e^g \le x'_d < \delta_d(n') + e^g =$ `SubDomains`$[n'][d][1] -$ `SubDomains`$[n'][d][0] + e^g$ and decrement/increment k by 3^d if $x'_d < 0$ or $x'_d \ge \delta_d(n')$ respectively, or we let $a = 1$ to mean we have anywhere accommodation.

```
#define Local_Coordinate(N, MYSD, GG, LG, DIM, K, INC, AA) {\
  GG -= SubDomains[N][DIM][OH_LOWER];\
  if (N==MYSD)  LG = GG;\
  else {\
    const int ub = SubDomains[MYSD][DIM][OH_UPPER];\
    if (LG>=ub+OH_PGRID_EXT)  AA = 1;\
    else {\
      const int inc = LG<ub ? 0 : INC;\
      LG -= SubDomains[MYSD][DIM][OH_LOWER];\
      if (LG<-OH_PGRID_EXT)  AA = 1;\
      k += LG<0 ? -INC : inc;\
    }\
  }\
}
```

4.10.51 oh4p_map_particle_to_subdomain()

oh4p_map_particle_to_subdomain_()
oh4p_map_particle_to_subdomain()

The API functions `oh4p_map_particle_to_subdomain_()` for Fortran and `oh4p_map_particle_to_subdomain()` for C find the subdomain m in which the local node's primary (`ps` $= p = 0$) or secondary particle of species `s` $= s$ pointed by `part` $= \pi$ resides, and return m or -1 if particle went out-of-bounds. The function for C is also called from `oh4p_map_particle_to_neighbor()` and `oh4p_remap_particle_to_subdomain()` to find the subdomain for a particle warping and being remapped respectively.

They have a number of difrences from their level-3 couterparts `oh3_map_particle_to_subdomain[_]()`. First, they receive the `S_particle` structure pointer π rather than their position coordinate values. Second, they need species s to refer to an element of `NOfPLocal`[][][], mantaining it by the functions rather than the simulator body to bring another difference, and `NOfPGrid`[][][] for which we need calculations of integer coordinate values and the grid-position of the particle. Third, they take care of the system periodic boundary if π has been crossing it.

The function `oh4p_map_particle_to_subdomain_()` simply calls its counterpart `oh4p_map_particle_to_subdomain()`, but decrementing `s` by 1 to make it zero-origined.

```
int
oh4p_map_particle_to_subdomain_(struct S_particle *part, const int *ps,
                                const int *s) {
  return(oh4p_map_particle_to_subdomain(part, *ps, *s-1));
}
int
oh4p_map_particle_to_subdomain(struct S_particle *part, const int ps,
                               const int s) {
  const int ns = nOfSpecies,  inj = part>=Particles+totalParts;
  const int nx  = Grid[OH_DIM_X].n;
  const int nxy = If_Dim(OH_DIM_Y, nx*Grid[OH_DIM_Y].n, 0);
  const int t = ps ? ns + s : s;
  int w, dw, mysd;
  int sd;
```

```
double x, y, z;
int px, py, pz;
int gx, gy, gz;
int lx, ly, lz;
int k = OH_NBR_SELF,  aacc = 0;
Decl_Grid_Info();
```

First, we invoke `Check_Particle_Location()` to check the consistency of arguments, before referring to elements in `GridDesc`[] and `RegionId`[]. The fifth argument of the macro i is determined by whether π is beyond `Particles`+`nOfInjections`, i.e., π is for an injected particle. Then, we invoke `Map_To_Grid()` D-times to calculate the d-th dimensional global integer coordinate x_d and x'_d of the grid-voxel in which π resides with/without taking care of system periodic boundary crossing respectively, for all $d \in [0, D)$ supressing invocations for $d \geq D$ by `Do_Y()` and/or `Do_Z()`.

Then if `SubDomainDesc` $\neq$ `NULL` to mean we have irregular process coordinate, we call `map_irregular_subdomain()` giving it floating point coordinates of π to obtain the subdomain identifier m in which π resides, and confirm $m \geq 0$ or make π elimintated due to out-of-bounds letting its `nid` and the function's return value be -1.

Otherwise, i.e., if we have regular process coordinate, we invoke `Map_Particle_To_Subdomain()` D-times to have d-th dimensional process coordinate π_d, from which we calculate m in which π resides by `Coord_To_Index()` to which we also give $\mathit{\Pi}_0$ and $\mathit{\Pi}_0 \cdot \mathit{\Pi}_1$ where $\mathit{\Pi}_d =$ `Grid`$[d].n$.

```
Check_Particle_Location(part, ps, s, ns, inj);
w = GridDesc[ps].w;  dw = GridDesc[ps].dw;  mysd = RegionId[ps];
Map_To_Grid(part, part->x, x, OH_DIM_X, gx, lx);
Do_Y(Map_To_Grid(part, part->y, y, OH_DIM_Y, gy, ly));
Do_Z(Map_To_Grid(part, part->z, z, OH_DIM_Z, gz, lz));
if (SubDomainDesc) {
  sd = map_irregular_subdomain(x, If_Dim(OH_DIM_Y ,y, 0),
                                  If_Dim(OH_DIM_Z, z, 0));
  if (sd<0) { part->nid = -1;  return(-1); }
} else {
  Map_Particle_To_Subdomain(gx, OH_DIM_X, px);
  Do_Y(Map_Particle_To_Subdomain(gy, OH_DIM_Y, py));
  Do_Z(Map_Particle_To_Subdomain(gz, OH_DIM_Z, pz));
  sd = Coord_To_Index(px, py, pz, nx, nxy);
}
```

Now we can convert global coordinate values x_d and x'_d to their counterparts local to m and $n' = \{n, \mathit{parent}(n)\}[p] =$ `RegionId`$[p]$ for the local node n by invoking `Local_Coordinate()` D-times. By these invocations, we also know π is at somewhere outside of the exterior of n' and thus we have anywhere accommodation. If so, we set the bit for accommodation mode in `currMode` by `Mode_Set_Any()` and let `nid` of π be $(m+3^D) \cdot 2^\Gamma + \mathit{gidx}(x_0, \ldots)$.

Otherwise, we increment `NOfPGrid`$[p][s][\mathit{gidx}(x'_0, \ldots)]$ using `Coord_To_Index()` again but this time giving it $\delta_0^{\max} + 4e^g =$ `GridDesc`$[p]$`.w` and the product of it and $(\delta_1^{\max} + 4e^g)$ being `GridDesc`$[p]$`.dw`. Then, we let `nid` of π be $k \cdot 2^\Gamma + \mathit{gidx}(x_0, \ldots)$ by `Coord_To_Index()` and `Combine_Subdom_Pos()` where k is the neighbor index calculated by `Local_Coordinate()`.

Then if the partilce π is injected one, we increment `InjectedParticles`$[0][p][s]$ if $m = n'$ because we need it maintained for non-position-aware particle transfer, and let the subdo-

main code of `nid` be $\{k, m+3^D\} + N + 3^D$ if $p = 1$ by `Secondarize_Id()` to tell its secondariness to the functions called from `transbound4p()`.

Finally we return to the the caller giving m as the return value.

```
  Local_Coordinate(sd, mysd, gx, lx, OH_DIM_X, k, 1, aacc);
  Do_Y(Local_Coordinate(sd, mysd, gy, ly, OH_DIM_Y, k, 3, aacc));
  Do_Z(Local_Coordinate(sd, mysd, gz, lz, OH_DIM_Z, k, 9, aacc));
  NOfPLocal[t*nOfNodes+sd]++;
  if (aacc) {
    currMode = Mode_Set_Any(currMode);
    part->nid = Combine_Subdom_Pos(sd+OH_NEIGHBORS,
                                   Coord_To_Index(gx, gy, gz, w, dw));
  } else {
    NOfPGrid[ps][s][Coord_To_Index(lx, ly, lz, w, dw)]++;
    part->nid = Combine_Subdom_Pos(k, Coord_To_Index(gx, gy, gz, w, dw));
  }
  if (inj) {
    if (sd==mysd)  InjectedParticles[t]++;
    if (ps)  Secondarize_Id(part);
  }
  return(sd);
}
```

4.10.52 `oh4p_inject_particle()`

`oh4p_inject_particle_()` `oh4p_inject_particle()`

The API functions `oh4p_inject_particle_()` for Fortran and `oh4p_inject_particle()` for C inject a particle pointed by `part` $= \pi$ as a primary (`ps` $= p = 0$) or secondary ($p = 1$) one for the local node expecting (or knowing) it resides in the local node n's primary/secondary subdomain.

The differences of them from their counterparts `oh2_inject_particle[_]()` are as follows. First they have `ps` $= p$ argument to specify the subdomain in which the particle likely resides, while the level-2 counterparts *guess* that from its `nid` element. Second, they determine the subdomain m in which the particle resides by themselves calling `oh4p_map_particle_to_neighbor()` which also maintains `NOfPLocal[][][]` and `NOfPGrid[][][]` for the particle, rather than expecting `nid` has the subdomain identifier. Finally, they have a return value m so that the caller is aware that the particle is out-of-bounds if so.

The function `oh4p_inject_particle_()` simply calls its counterpart `oh4p_inject_particle()` which does everything.

```
int
oh4p_inject_particle_(const struct S_particle *part, const int *ps) {
  return(oh4p_inject_particle(part, *ps));
}
int
oh4p_inject_particle(const struct S_particle *part, const int ps) {
  const int ns = nOfSpecies;
  int inj = totalParts + nOfInjections++;
  struct S_particle *p = Particles + inj;
  int s = Particle_Spec(part->spec - specBase);
  int sd;
```

In the declaration part, we determine the location of the particle is stored, i.e., `Particles`[$Q_n + Q_n^{\rm inj}$] where Q_n = `totalParts` and $Q_n^{\rm inj}$ = `nOfInjections`, increment $Q_n^{\rm inj}$ to keep track the total number of injected particles, and have the species s of the particle by `Particle_Spec()` taking its origin `specBase` into account.

Then we confirm `S_particle` structure has `spec` element, i.e., `OH_HAS_SPEC` is true, or $S = 1$, or abort the execution by `local_errstop()`. The abortion also takes place if the total number of the accommodating particles including that just now injected exceeds the absolute limit P_{lim} = `nOfLocalPLimit`.

Then we store the particle into the location above and call `oh4p_map_particle_to_neighbor()` to obtain the subdomain identifier m, which could be less than 0 to cause canceling the injection. Note that `oh4p_map_particle_to_neighbor()` recognizes the particle is injected because we give it the location beyond `Particles`[`totalParts`] and thus maintains `InjectedParticles`[][][] and secondarize the particle if necessary.

Finally we return to the the caller giving m as the return value.

```
#ifndef OH_HAS_SPEC
  if (ns!=1)
    local_errstop("particles cannot be injected when S_particle does not "
                  "have 'spec' element and you have two or more species");
#endif
  if (inj>=nOfLocalPLimit)
    local_errstop("injection causes local particle buffer overflow");
  *p = *part;
  sd = oh4p_map_particle_to_neighbor(p, ps, s);
  if (sd<0)  nOfInjections--;
  return(sd);
}
```

4.10.53 `oh4p_remove_mapped_particle()`

`oh4p_remove_mapped_particle_()`
`oh4p_remove_mapped_particle()`

The API functions `oh4p_remove_mapped_particle_()` for Fortran and `oh4p_remove_mapped_particle()` for C eliminate a primary (`ps` = $p = 0$) or secondary ($p = 1$) particle pointed by `part` = π of species `s` = s, which has already been *mapped* on to a subdomain by `oh4p_map_particle_to_neighbor()`, `oh4p_map_particle_to_subdomain()` or `oh4p_inject_particle()`. This explicit elmination is required to maintain `NOfPLocal`[][][] and `NOfPGrid`[][][] in order to reflect the elimination to them.

First, we invoke `Check_Particle_Location()` to check the consistency of arguments giving the fifth argument i determined by whether π is beyond `Particles`+`nOfInjections`, i.e., π is for an injected particle. Then we examine if the π's `nid` is negative and return to the caller doing nothing if so. Next we obtain the subdomain identifier m of π by `Subdomain_Id()` and examine if $m \geq N$. If so to mean the particle is an injected secondary one, we get real m by `Primarize_Id()` and force $p = 1$.

Then we mark π eliminated by letting its `nid` be -1 and decrement `NOfPLocal`[p][s][m]. Then if the particle is injected into the local node n's primary/secondary subdomain, i.e., $m = n' = \{n, parent(n)\}[p]$ = `RegionId`[p], we decrement `InjectedParticles`[0][p][s] to cancel the increment on the injection. Finally, if we have normal accommodation so far, i.e., `Mode_Acc()` of `currMode` is false, we also decrement `NOfPGrid`[p][s][g'], where $g' = g$ if $m = n'$ or otherwise g' is what `Local_Grid_Position()` gives us with g, `nid` of (primarized) π and p, to cancel the increment done when π was mapped.

```
void
oh4p_remove_mapped_particle_(struct S_particle *part, const int *ps,
                             const int *s) {
  oh4p_remove_mapped_particle(part, *ps, *s-1);
}
void
oh4p_remove_mapped_particle(struct S_particle *part, const int ps,
                            const int s) {
  const int nn = nOfNodes, ns = nOfSpecies, inj = part>=Particles+totalParts;
  OH_nid_t nid = part->nid;
  int sd, g, psreal=ps, mysd, t;
  Decl_Grid_Info();

  Check_Particle_Location(part, psreal, s, ns, inj);
  if (nid<0)  return;
  sd = Subdomain_Id(nid, psreal);
  g = Grid_Position(nid);
  if (sd>=nn) {
    psreal = 1;  Primarize_Id(part, sd);  nid = part->nid;
  }
  mysd = RegionId[psreal];
  part->nid = -1;
  t = psreal ? ns+s : s;
  NOfPLocal[t*nn+sd]--;
  if (inj && sd==mysd)  InjectedParticles[t]--;
  if (Mode_Acc(currMode))  return;
  if (sd!=mysd)  g = Local_Grid_Position(g, nid, psreal);
  NOfPGrid[psreal][s][g]--;
}
```

4.10.54 oh4p_remap_particle_to_neighbor()

oh4p_remap_particle_to_neighbor_()
oh4p_remap_particle_to_neighbor()

The API functions `oh4p_remap_particle_to_neighbor_()` for Fortran and `oh4p_remap_particle_to_neighbor()` do what the functions `oh4p_remove_mapped_particle()` and `oh4p_map_particle_to_neighbor()` do in series, giving arguments `part`, `ps` and `s` to both of them.

```
int
oh4p_remap_particle_to_neighbor_(struct S_particle *part, const int *ps,
                                 const int *s) {
  return(oh4p_remap_particle_to_neighbor(part, *ps, *s-1));
}
int
oh4p_remap_particle_to_neighbor(struct S_particle *part, const int ps,
                                const int s) {
  oh4p_remove_mapped_particle(part, ps, s);
  return(oh4p_map_particle_to_neighbor(part, ps, s));
}
```

4.10.55 oh4p_remap_particle_to_subdomain()

oh4p_remap_particle_to_subdomain_()
oh4p_remap_particle_to_subdomain()

The API functions oh4p_remap_particle_to_subdomain_() for Fortran and oh4p_remap_particle_to_subdomain() do what the functions oh4p_remove_mapped_particle() and oh4p_map_particle_to_subdomain() do in series, giving arguments part, ps and s to both of them.

```
int
oh4p_remap_particle_to_subdomain_(struct S_particle *part, const int *ps,
                                  const int *s) {
  return(oh4p_remap_particle_to_subdomain(part, *ps, *s-1));
}
int
oh4p_remap_particle_to_subdomain(struct S_particle *part, const int ps,
                                 const int s) {
  oh4p_remove_mapped_particle(part, ps, s);
  return(oh4p_map_particle_to_subdomain(part, ps, s));
}
```

4.11 Level-4s Library Overview

The level-4s library is an extension of OhHelp for yet another position-aware particle management for SPH (Smoothed Particle Hydrodynamics) method. In the SPH method, the computations on a particle require attributes of other particles surrounding it. This means that the computations on particles in a voxel may refer to particles in voxels surrounding it, and thus a node responsible of a set of voxels must have particles in voxels surrounding the voxel set.

This requirement makes it tough to implement the level-4s library as an extension of the level-4p counterpart due to the followings. First, we cannot split the particle set in a hot-spot because the whole set must be accommodated by the node for the voxel and possibly by other nodes for the voxels surrounding it. Second, in the level-4p the shape of the voxel set for a node is not always a cuboid to bring a severe complication into the communication to let the node have particles in voxels surrounding the set.

For the level-4s library, these difficulties are eased by introducing the consept of the *maximum density* being the maximum number $\mathcal{D}$ of particles in a voxel. Since SPH simulations of incompressible flow obviously has a certain upper bound of the particle density and we can expect some upper bound even in those of compressible one, we exploit the maximum density to make us free from the hot-spot problem. Moreover, the maximum density allows us to make the unit of subdomain splitting larger than a voxel because we have a certain upper bound number of particles in a unit set of voxels. More specifically, the unit in level-4s library is a xy-plane of a subdomain so that the set of voxels for a node is always a cuboid, which we refer to as (primary/secondary) *subcuboid*. Though this causes that the excess of the particle population for a node from what the OhHelp balancer suggests is significantly large, up to $\mathcal{D} \cdot \delta_x(n) \cdot \delta_y(n)$ for a node having a subdomain n, the inter-process communication of *halo* particles in the voxels at interior/exterior surface of the subcuboid, or in other words in *halo planes* consisting *interior halo planes* and *exterior halo planes* having particles to be sent and received respectively, can be implemented relatively easily.

Despite the level-4s's own features shown above, its implementation shares many aspects with the level-4p counterparts. Its outline is shown below emphasizing its similarity to and difference from the level-4p library.

1. The functions to map particles to subdomains such as `oh4s_map_particle_to_neighbor()` and particle injection/removal functions such as `oh4s_inject_particle()` are perfectly equivalent their level-4p counterparts, `oh4p_map_particle_to_neighbor()`, `oh4p_inject_particle()` and so forth.

2. The function `transbound4s()` and its direct callee functions `try_stable4s()` and `rebalance4s()` are very similar to their level-4s counterparts, but `try_primary4s()` is completely different from `try_primary4p()` because it calls `exchange_particles4s()` for primary particle transfer and sorting instead of having its own mechanism. This difference comes from the functionality to exchange halo particles between nodes for subdomains neighboring to the local node's primary (and secondary in general) subdomains.

3. When the complete per-grid histogram $\mathcal{P}_T(p, s, g) = \texttt{NOfPGridTotal}[p][s][g]$ is built in `exchange_population()`, the function also builds the *per-plane histogram* $\mathcal{P}_Z(z) = \sum_s \sum_{x,y} \mathcal{P}_T(0, s, gidx(x, y, z))$ for each xy-plane at z in the primary subdomain of the local node. This histogram is scanned by `make_recv_list()` when we will be in secondary mode in the next step so that each member node of the local node n's primary family is assigned particles in a subcuboid and the population in it is

approximately equal to that the OhHelp load balancing mechanism requires. That is, `make_recv_list()` determines $\zeta_p^\beta(m)$ being the local z-coordinate of the lower ($\beta = 0$) or upper ($\beta = 1$) surface of the primary ($p = 0$) or secondary ($p = 1$) subcuboid for all $m \in F(n)$ where $p = 0$ for $m = n$ and $p = 1$ for others, and record them in primary receiving block.

4. Based on the assignment above and that received from the helpand, or the obvious assignment for primary mode execution, `make_send_sched()` decides the destination of each particle in the local node in a manner similar to (but simpler than because of no hot-spots) that of its level-4p counterpart. In addition, the function also performs the level-4s's own procedure to build the sending/receiving schedule of particles in *horizontal halo planes* for nodes whose subcuboids are below/above the local node's subcuboids.

5. After exchanging the sending/receiving particle amount by `exchange_xfer_amount()` as done in level-4p, non-halo particles are transferred and sorted by `move_and_sort()`, `xfer_particles()` and `sort_received_particles()` if `SendBuf[]` can accommodate particles both to be sent and to reside in the next step and helpand-helper reconfiguration does not take place. Otherwise, `move_to_sendbuf_4s()`, `xfer_particles()` and `sort_particles()` perform those operations. A differences from level-4p is that `move_and_sort()` and `move_to_sendbuf_4s()` are commonly used for both primary and secondary mode cases because their caller `exchange_particles4s()` is commonly used too. The other and more important difference is that a level-4s's own function `make_bxfer_sched()` is called just before `move_and_sort()` or `sort_particles()` to build the sending/receiving schedule of particles in *vertical halo planes* for neighbor nodes having *contacting subcuboids* which share a vertical surface parallel to z-axis with the local node's primary/secondary subcuboid. The sending schedule is referred to by `move_and_sort()`, `sort_particles()` and `sort_received_particles()` to move particles in *vertical interior halo planes* parallel to z-axis, to the newly introduced sending buffer `BoundarySendBuf[]` being the sequence of $hbuf_v^s(d, p, \beta, m)$ for primary($p = 0$) or secondary ($p = 1$) particles to be sent to the node m responsible of the d-th dimensional ($d \in \{0, 1\}$) lower ($\beta = 0$, west or south) or upper ($\beta = 1$, east or north) neighbor subdomain of the local node's primary/secondary one.

6. After the particle transfer communication taken by `xfer_particles()` to have all non-halo particles to be accommodated by the local node in `SendBuf[]` as done in level-4p, `xfer_boundary_particles_v()` sends particles in vertical interior halo planes to neighbors via $hbuf_v^s(d, p, \beta, m)$ in `BoundarySendBuf[]` and then receives halo particles into *vertical exterior halo planes* parallel to z-axis and just *outside* of thier interior counterparts via $hbuf_v^r(d, p, \beta, m)$ in `Particles[]`. The function works twice with $d = 0$ for west/east-bound communication and then with $d = 1$ for south/north-bound, the latter of which carries halo particles in neighbor subdomains which contact with the local one by edges to make the direct communications with nodes for the subdomains unnecessary. Then `xfer_boundary_particles_h()` performs halo particle transfer dierectly from *horizontal interior halo planes* in `SendBuf[]` namely $hbuf_h^s(p, \beta, s)$ being the bottommost ($\beta = 0$) and topmost ($\beta = 1$) xy-planes of the local node's subcuboid, and directly to *horizontal exterior halo planes* in `SendBuf[]` namely $hbuf_h^r(p, \beta, s)$ just below ($\beta = 0$) and above ($\beta = 1$) the interior counterparts. Since this carries halo particles in neighbor subdomains contacting with the local one by vertices, it is unnecessary to communicate the nodes for those subdomains directly too.

4.12 Header File ohhelp4s.h

The header file of level-4s library, ohhelp4s.h, is very similar to the level-4p counterpart ohhelp4p.h but has a few additions, deletions and modifications for introducing subcuboid-based particle assignment and halo particle transfer, and for eliminating hot-spot-related functions.

4.12.1 Constants

At first we define the following constants as done in ohhelp4p.h (§4.9.1).

OH_PGRID_EXT
- The constant OH_PGRID_EXT $= e^g = 1$ is exactly equivalent to that in the level-4p library, but the level-4s library cannot cope with the case of $e^g > 1$ and thus init4s() aborts the execution if $e^g > 1$.

OH_NBR_SELF
- The constant OH_NBR_SELF $= \sum_{d=0}^{D-1} 3^d = \lfloor 3^D/2 \rfloor$ is exactly equivalent to that in the level-4p library.

Then we define the following level-4s's own constants for neighbor indices.

OH_NBR_BCC
OH_NBR_TCC
- The constants OH_NBR_BCC $= 1 \cdot 3^0 + 1 \cdot 3^1 + 0 \cdot 3^2 = 2$ and OH_NBR_TCC $= 1 \cdot 3^0 + 1 \cdot 3^1 + 2 \cdot 3^2 = 20$ are the neighbor indices of the subdomains contacting with the local node's subdomain at its bottom and top surfaces. They are used in make_send_sched() to know neighbor nodes whose subcuboids contact with the local subcuboid at its bottom and top surfaces.

Here we revisit a few other constants/switches related to level-4s extension.

OH_LIB_LEVEL_4S
OH_LIB_LEVEL_4PS
- The switch OH_LIB_LEVEL_4S can be defined in oh_config.h to declare that the OhHelp library should be configured with level-4s extension, as discussed in §3.3. The switch OH_LIB_LEVEL_4PS is defined iff OH_LIB_LEVEL_4S or OH_LIB_LEVEL_4P is defined.

OH_POS_AWARE
STATS_TB_SORT
- The switch OH_POS_AWARE and the constant STATS_TB_SORT can be defined in ohhelp1.h and oh_stats.h respectively, for position-awareness and sorting time measurement, as discussed in §4.9.1.

- With other library levels, D = OH_DIMENSION can be less than 3 for 1- or 2-dimensional simulations. However, the level-4s library is applicable only to 3-dimensional simulations and thus init4s() aborts the execution if $D \neq 3$.

```
#define OH_PGRID_EXT    1
#define OH_NBR_SELF     (OH_NEIGHBORS>>1)
#define OH_NBR_BCC      (1+1*3+0*3*3)
#define OH_NBR_TCC      (1+1*3+2*3*3)
```

4.12.2 Macros for Grid-Position

The level-4s mechanism to represent grid-position, the one-dimensional index of each grid-voxel, in the nid element of S_particle is almost equivalent to that of level-4p, but the

grid-position given by $gidx(x, y, z)$ and the size of per-grid histogram G are *different* from those in level-4p as follows.

$$gidx(x, y, z) = x + (\delta_x^{\max} + 6e^g)(y + (\delta_y^{\max} + 6e^g)z)$$

$$G = \prod_{d=0}^{D-1} (\delta_d^{\max} + 6e^g)$$

The difference comes from that the per-grid histogram has $2e^g$ thick sending and receiving planes to exchange particle populations. Since the inside portion of sending plane of e^g thick is in the interior of the local node's subdomain, while its outside portion of e^g thick and $2e^g$ thick receiving plane are in the exterior, the per-grid histogram has $3e^g$ thick exterior at its lower and upper boundaries to make its width, depth and height $\delta_d^{\max} + 6e^g$. The reason why the sending/receiving planes are $2e^g$ thick is that the per-grid histogram `NOfPGridTotal[][][]` needs to have the population of grid-voxels in exterior halo planes, corresponding to the outside sending planes of per-grid histogram, to know halo particle population.

The difference above, however, does not affect the switch `OH_BIG_SPACE`, the data type `OH_nid_t`, the global variables `logGrid`, `gridMask` and `AbsNeighbors`, and macros `Decl_Grid_Info()`, `Subdomain_Id()` and `Primarize_Id()`, which header files other than ohhelp4s.h define. Therefore and since their definitions (but not necessarily their values) are perfectly equivalent to those discussed in §4.9.2, here we just show the level-4s functions which access the variables and use the macros.

- The integer variables `logGrid` and `gridMask` are initialized by `init4s()`.
- The two dimensional array `AbsNeighbors`$[2][3^D]$ are initialized/updated by `update_neighbors()` called from `init4s()`, `rebalance4s()` and `exchange_particles4s()`, and is referred to by `oh4s_map_particle_to_neighbor()` directly and other functions through the macro `Subdomain_Id()` or `Neighbor_Subdomain_Id()`.
- The macro `Decl_Grid_Info()` is used in the following functions;

  `rebalance4s()`, `count_population()`, `move_to_sendbuf_4s()`, `move_to_sendbuf_uw4s()`, `move_to_sendbuf_dw4s()`, `sort_particles()`, `move_and_sort()`, `sort_received_particles()`, `oh4s_map_particle_to_neighbor()`, `oh4s_map_particle_to_subdomain()`, `oh4s_remove_mapped_particle()`.

- The macro `Subdomain_Id`(i, p) is used in `oh4s_remove_mapped_particle()`.
- The macro `Primarize_Id`(π, m) is used in `rebalance4s()` and `oh4s_remove_mapped_particle()`.

The equivalence to level-4p also holds for the following macros defined in ohhelp4s.h and thus we just show the level-4s functions using them.

`Grid_Position()`

- The macro `Grid_Position`(i) is used in `count_population()` and `oh4s_remove_mapped_particle()` directly, in `move_to_sendbuf_4s()`, `move_to_sendbuf_uw4s()`, `move_to_sendbuf_dw4s()` and `move_and_sort()` through the macro `Move_Or_Do()`, and in `sort_particles()` and `sort_received_particles()` through the macro `Sort_Particle()`.

Combine_Subdom_Pos()

- The macro Combine_Subdom_Pos(σ, g) is used in count_population(), oh4s_map_particle_to_neighbor() and oh4s_map_particle_to_subdomain().

Primarize_Id_Only()

- The macro Primarize_Id_Only(π) is used in move_to_sendbuf_4s() and move_and_sort().

Secondarize_Id()

- The macro Secondarize_Id(π) is used in rebalance4s(), oh4s_map_particle_to_neighbor() and oh4s_map_particle_to_subdomain().

Secondary_Injected()

- The macro Secondary_Injected(i) is used in rebalance4s(). move_to_sendbuf_4s() and move_and_sort().

Neighbor_Subdomain_Id()

- The macro Neighbor_Subdomain_Id(i, p) is used in move_to_sendbuf_4s(), move_to_sendbuf_uw4s(), move_to_sendbuf_dw4s() and move_and_sort() through the macro Move_Or_Do().

```
#define Grid_Position(ID)         ((ID)&gridmask)
#define Combine_Subdom_Pos(ID, G) (((OH_nid_t)(ID)<<loggrid) + (G))
#define Primarize_Id_Only(P) \
  (P)->nid -= (OH_nid_t)(nOfNodes+OH_NEIGHBORS)<<loggrid
#define Secondarize_Id(P) \
  (P)->nid += (OH_nid_t)(nOfNodes+OH_NEIGHBORS)<<loggrid
#define Secondary_Injected(ID) \
  ((ID>>loggrid)>=nOfNodes+OH_NEIGHBORS)
#define Neighbor_Subdomain_Id(ID, PS) \
  AbsNeighbors[PS][(ID)>>loggrid]
```

4.12.3 Per-Grid Histograms and Related Variables

Next, we declare the following arrays of per-grid histograms and related variables, some of which are perfectly equivalent to those declared in the level-4p library.

PbufIndex

- PbufIndex[2][S] is perfectly equivalent to that declared in the level-4p library, is initialized by init4s() with NULL, is allocated by the first call of transbound4s() which also sets all elements in each call, and is referred to by oh4s_map_particle_to_neighbor(), oh4s_map_particle_to_subdomain() and oh4s_remove_mapped_particle() through the macro Check_Particle_Location().

NOfPGrid
NOfPGridTotal

- As in level-4p, NOfPGrid[2] and NOfPGridTotal[2] are double pointer arrays to implement three-dimensional dint arrays of [2][S][G] whose element [p][s][g] has the number of primary ($p = 0$) or secondary ($p = 1$) particles of species s in the grid-voxel whose index is g, namely $\mathcal{P}_L(p, s, g)$ and $\mathcal{P}_T(p, s, g)$ respectively. Besides the fact that the size G is larger than that in level-4p as discussed in §4.12.2, their roles are little bit different from those in level-4p.

  The allocation and initialization of both arrays in init4s(), reinitialization of NOfPGrid[][][] in transbound4s(), and its counting up/down in oh4s_map_particle_to_neighbor(), oh4s_map_particle_to_subdomain() or oh4s_remove_mapped_particle() are equivalent to those in level-4p, except for the reinitialization in transbound4s() which enlarging receiving plane thickness affects.

  Their first roles in transbound4s() are fundamentally equivalent to those in level-4p, but the process to build the complete per-grid histogram is a little bit *different* as follows.

– The complete histogram is always built in `NOfPGridTotal[][][]` regardless the mode in the next simulation step, by `exchange_population()`. Therefore, `NOfPGrid`[0]`[][][]` is copied into `NOfPGridTotal`[0]`[][][]` if the current mode is primary, while `NOfPGrid[][][]` of the primary family members are summed up by `reduce_population()` if the mode is secondary.
– Regardless the accommodation pattern, the sending planes of `NOfPGridTotal`[0]`[][]` are exchanged among neighbor nodes to have receiving planes in `exchange_population()`, because we always need to have halo particle population. Therefore, even if we have anywhere accommodation, `NOfPGridTotal`[0]`[][]` is built in the same way of normal accomodation but after the local histogram is built in `NOfPGrid[][][]` by `count_population()`.
– Regardless of the current/next mode and accommodation pattern, `NOfPGrid[][][]` is referred to by `make_send_sched_body()` through the macro `Make_Send_Sched_Body()` to know the number of particles to be sent to other nodes, and `NOfPGridTotal[][][]` is referred to by `make_send_sched_hplane()` to know the number of particles to be accommodated and transferred for halo particle exchange.
– The function `sched_recv()` does not refers to `NOfPGridTotal[][][]` but to the per-plane histogram `NOfPGridZ`$[z] = \mathcal{P}_Z(z) = \sum_s \sum_{x,y} \mathcal{P}_T(0, s, gidx(x, y, z))$ for each z to determine the subcuboids to be assigned to primary members.

Note that `NOfPGridTotal`[0]`[][]` is broadcasted in `make_recv_list()` to primary family members so that they have it as `NOfPGridTotal`[1]`[][]`, as done in level-4p.

Then `NOfPGrid[][][]` is modified by `make_send_sched_body()` (through the macro `Make_Send_Sched_Body()`) to have one of the followings after its original contents are examined.

(a) 0 means that the particles in the grid-voxel stays in the local node, as in level-4p.
(b) A positive number $(pS + s)N + m + 1 < 2^{32}$ means that the particles of species s in the grid-voxel is sent to the node m as its primary ($p = 0$) or secondary ($p = 1$) particle, as in level-4p.

Further, the array is modified by `make_bsend_sched()` to have one of the following level-4s's own values in addition to the values above.

(c) A positive number $(i + 1) \cdot 2^{32} + \sigma \geq 2^{32}$ means the particles in the grid-voxel stay in the local node ($\sigma = 0$) or is sent to other node ($\sigma > 0$) as in above two cases, and are copied to `BoundarySendBuf`$[i]$ after the reception of halo particles since the grid-voxel is in an *exterior pillar* being an intersection of a west/east vertical exterior halo plane and a south/north vertical interior halo plane.
(d) A negative number $-(i + 1) > -2^{32}$ means that the grid-voxel is in a vertical interior halo plane and thus particles in it definitely stay in the local node. The copy of the first particle in it goes to `BoundarySendBuf`$[i]$ to be sent to a node as its halo particle.
(e) A negative number $-(i + 1) \cdot 2^{32} - (j + 1) \leq -2^{32}$ means that the grid-voxel is in a *interior pillar* being the intersection of two vertical interior halo plane, and thus particles in it definitely stay in the local node too. In this case, the first copied particle goes to both `BoundarySendBuf`$[i]$ and `BoundarySendBuf`$[j]$ to send it to the south/north and west/east neighbors respectively.

The functions `move_to_sendbuf_4s()`, `move_to_sendbuf_uw4s()` and `move_to_sendbuf_dw4s()` refer to `NOfPGrid[][][]` having values (a) or (b) because they are called before `make_bsend_sched()`, while `sort_particles()` may see (a), (b), (d) or (e)[77], but (a) and (b) are not distinguish because the grid-voxel is definitely in a subcuboid of the local node. On the other hand, `move_and_sort()` may see all values but it is assured that a grid-voxel having (d) or (e) should have had (a) before `make_bsend_sched()` let it have them, because, without helpand-helper reconfiguration, the old and new secondary subdomain are same. Note that a grid-voxel having (c) can have had (a) because the neighbor sharing the vertical exterior halo plane as its vertical interior halo plane can be the local node itself with periodic boundary condition. Finally, `sort_received_particles()` can see (a), (d) or (e) because it should visit grid-voxels only in subcuboid of the local nodes.

On the other hand, `NOfPGridTotal[][][]` is modified by `exchange_particles4s()` to have the sorting index of `SendBuf[]` to which the first particle in the grid-voxel is moved, as in level-4p. A small *difference* from level-4p is that `SendBuf[]` will have halo particles and thus the sorting index must takes them into account. That is, for the local node n, it is initialized as follows and then its element is incremented each time a particle in the grid-voxel is moved by the functions `sort_particles()`, `move_and_sort()` or `sort_received_particles()`.

$$n^q = \begin{cases} n & q = 0 \\ parent(n) & q = 1 \end{cases}$$

$$\delta_d(m) = \delta_d^u(m) - \delta_d^l(m)$$
$$\mathcal{R}_d(q) = [-e^g,\ \delta_d(n^q) + e^g)$$
$$\mathcal{G}(q) = \{g \,|\, g = gidx(x,y,z),\ x \in \mathcal{R}_x(q),\ y \in \mathcal{R}_y(q),\ z \in \mathcal{R}_z(q)\}$$
$$\texttt{NOfPGridTotal}[p][s][g] = \sum_{q=0}^{p-1}\sum_{t=0}^{S-1}\sum_{h\in\mathcal{G}(q)} \texttt{NOfPGridOut}[q][t][h] + \sum_{t=0}^{s-1}\sum_{h\in\mathcal{G}(p)} \texttt{NOfPGridOut}[p][t][h] + \sum_{h\in\mathcal{G}(p),\ h<g} \texttt{NOfPGridOut}[p][s][h]$$

`NOfPGridOut`
`NOfPGridOutShadow`
`NOfPGridIndex`
`NOfPGridIndexShadow`

- As in level-4p, `NOfPGridOut[2]` is a double pointer array to implement a three-dimensional `int` array of $[2][S][G]$ whose element $[p][s][g]$ has the number of primary ($p = 0$) or secondary ($p = 1$) particles of species s in the grid-voxel whose index is g, namely $\mathcal{P}_O(p, s, g)$. Besides the enlargement of G again, it has a few *different* features from that in level-4p as follows.
  - `NOfPGridOut[][][]` has a shadow `NOfPGridOutShadow[][][]` whose body array is given or returned through the argument `pghgram` of `oh4s_per_grid_histogram()`, because it is referred to outside of `transbound4s()` and by `oh4s_exchange_border_data()` while no level-4p library function outside `transbound4p()` accesses the array. The substance array is allocated by `oh4s_per_grid_histogram()` using `Allocate_NOfPGrid()`, but the body of shadow is

[77] It cannot see (c) because it scans interior particles only, but can see (b) if helpand-helper reconfiguration takes place to let particles in a grid-voxel in the old secondary subdomain go to other nodes while the grid-voxel is in the new secondary subcuboid of the local node.

allocated if pghgram point NULL while its pointer array are allocated unconditionally. Copying the substance to the shadow is done by exchange_particles4s().

- NOfPGridOut[][][] has a correlated index array NOfPGridIndex[][][] and its shadow NOfPGridIndexShadow[][][] whose body is given or returned through the argument pgindex of oh4s_per_grid_histogram(), which allocates these index arrays in a manner similar to that of per-grid histograms. The substance index array has values same as the initial ones of NOfPGridTotal[][][] in its second role discussed in the previous item, but is let have values in exchange_particles4s(). The function also makes the shadow has the copy of the substance with C coded simulators, while each shadow element is made greater by one than the substance's with Fortran coded ones.
- NOfPGridOut[][][] is let have particle populations by make_send_sched_self() or make_send_sched_hplane() regardless of the next execution mode. Also regardless of the mode, it is referred to by exchange_particles4s() to let NOfPGridTotal[][][] have sorting indices.
- NOfPGridOut[][][] is also referred to by make_bsend_sched() and make_brecv_sched() to build the halo particle transfer schedules. Then, it is referred to by xfer_boundary_particles_v() and exchange_border_data_v() together with NOfPGridIndex[][][] to have the population and the index of SendBuf[] for particles, or of the buffer for particle-associated data in halo planes.

NOfPGridZ

- The one dimensional dint array NOfPGridZ[$\delta_z^{\max}$] is level-4s's own per-plane histogram to have the number of particles of each xy-plane at z-coordinate z of the local node's primary subdomain in its element $[z]$, namely $\mathcal{P}_Z(z)$. More specifically, the element $\mathcal{P}_Z(z)$ = NOfPGridZ[z] has the following for the local node n.

$$\mathcal{P}_Z(z) = \texttt{NOfPGridZ}[z] = \sum_{s=0}^{S-1} \sum_{y=0}^{\delta_y(n)-1} \sum_{x=0}^{\delta_x(n)-1} \mathcal{P}_T(0, s, gidx(x, y, z))$$

  After allocated by init4s(), the array is let have the values above by exchange_population(). Then it is referred to by sched_recv() to determine the primary family member node to which each xy-plane is assigned as a part of the subcuboid of the node.

ZBound
ZBoundShadow

- The int array ZBound[2][2] is level-4s's own to have the local z-coordinate of the lower ($\beta = 0$) or upper ($\beta = 1$) surface of the primary ($p = 0$) or secondary ($p = 1$) subcuboid $\zeta_p^\beta(n)$ of the local node n in ZBound[p][β]. It can be ZBound[p][0] = ZBound[p][1] = 0 to mean the corresponding subcuboid is empty because n is not assigned any particles, as (re)initialized by transbound4s() for $p \in \{0, 1\}$. After that, ZBound[p][β] is let have $\zeta_p^\beta(n)$ by make_send_sched_self(), and then is referred to by make_send_sched() to know if n has subcuboid, by make_bsend_sched() and make_brecv_sched() to examine if other node's subcuboid shares a vertical surface of n's subcuboid, and by xfer_boundary_particles_v() and exchange_border_data_v() to scan vertical halo planes.

  Since exchange_border_data_v() is outside of transbound4s() and the simulator body need $\zeta_p^\beta(n)$, ZBound[][] has its shadow ZBoundShadow[2][2]. The shadow is given through the zbound argument of oh4s_init(), or is allocated by init4s() and returned to the simulator body through the argument. The function init4s() also initializes the shadow letting $\zeta_0^l(n) = 0$ and $\zeta_0^u = \delta_z(n)$ to mean the local node n's

primary subdomain itself is n's primary subcuboid, and $\zeta_1^l(n) = \zeta_1^u(n) = 0$ to mean n does not have secondary subcuboid. Then all elements in the substance `ZBound[][]` are copied into those in the shadow by `transbound4s()` as a part of its post process.

`S_hplane` `HPlane`

- The array `HPlane`[2][2] of `S_hplane` structure with the following elements are level-4s's own to have per-node transfer schedules of particles and particle-associated data in horizontal halo planes.
  - `nbor` is the rank m of the node to/from which the particles in a plane is sent/received. This element can be `MPI_PROC_NULL` to mean the corresponding subcuboid is empty because the local node is not assigned any particles or its subcuboid's surface is at non-periodic system boundary.
  - `stag` has $(p \cdot 3^D + k)S$ to indicate that the particles are primary ($p = 0$) or secondary ($p = 1$) for the k-th neighbor node m. `rtag` also has $(p \cdot 3^D + k)S$ but indicates that the particles are primary ($p = 0$) or secondary ($p = 1$) for the local node and it is m's k-th neighbor. From both elements, the tag of `MPI_Isend()` and `MPI_Irecv()` to transfer halo particles of species s is generated as $(p \cdot 3^D + k)S + s$ to make each send/receive pair for each s unique for the local node even when a node occurs multiple times as receivers/senders of the transfer due to periodic boundary condition and/or two roles as helpand and a helper.
  - `nsend`[S] and `nrecv`[S] have the number of particles of species s to be sent/received in their element [s].
  - `sbuf`[S] and `rbuf`[S] have the head indices of the send/receive buffer in their element [s]. For particle transfer, they are indices of $hbuf_h^s(p, \beta, s)$ and $hbuf_h^r(p, \beta, s)$ in `SendBuf[]`, while they are those of `buf[]` argument array of `oh4s_exchange_border_data()` for particle-associated data transfer performed by the function.

  `HPlane`[p][β] has the transfer schedule for the lower ($\beta = 0$) or upper ($\beta = 1$) horizontal halo plane of primary ($p = 0$) or secondary ($p = 1$) subcuboid.

  The array elements of S of `HPlane[][]` are allocated by `init4s()` which also initializes `nbor` being `MPI_PROC_NULL` in order to keep `oh4s_exchange_border_data()` from doing anything before the first call of `oh4s_transbound()`. Besides the initialization, `HPlane[][]` is let have the transfer schedule by `make_send_sched()` and its callees `make_send_sched_self()` and `make_send_sched_hplane()`, and then referred to by `xfer_boundary_particles_h()` and `exchange_border_data_h()` to transfer particles and particle-associated data respectively.

`S_vplane` `VPlane` `VPlaneHead`

- The array `VPlane`[$2N + 6$] of `S_vplane` structure, similar to `S_hplane` but a little bit different from it, with the following elements are level-4s's own to have per-node transfer schedules of particles and particle-associated data in vertical halo planes.
  - `nbor` is same as that of `S_hplane` to have rank m of the node to/from which the particles in a plane is sent/received.
  - `stag` and `rtag` are similar to those of `S_hplane`, but have $p \cdot 3^D + k$ because particles in all species are trasferred at once.
  - `nsend` and `nrecv` have the number of particles to be sent/received to/from `nbor`. Since particles in all species are transferred to/from `nbor` at once, they are scalars rather than arrays of S elements.

– `sbuf` and `rbuf` have the head indices of the send/receive buffer, being $hbuf_v^s(d,p,\beta,m)$ in `BoundarySendBuf[]` and $hbuf_v^r(d,p,\beta,m)$ in `Particles[]` respectively in particle transfer for `nbor` $= m$ and `stag` or `rtag` being $p \cdot 3^D + k$ where k is defined as follows for west $(d,\beta) = (0,0)$, east $(0,1)$, south $(1,0)$ and north $(1,1)$ neighbors.

$$k = 3^2 + (1 + d(2\beta - 1)) \cdot 3^1 + (1 + (1-d)(2\beta - 1)) \cdot 3^0 = \begin{cases} 3^2 + 3^1 + 2\beta \cdot 3^0 & d = 0 \\ 3^2 + 2\beta \cdot 3^1 + 3^0 & d = 1 \end{cases}$$

In particle-associated data transfer by by `oh4s_exchange_border_data()`, they are indices of the argument array `sbuf[]` and `rbuf[]` of the function respectively.

An element of `VPlane[]` has the following for the local node n. Let n^p $(p \in \{0,1\})$ be n's primary $(n^0 = n)$ or secondary $(n^1 = parent(n))$ subdomain, $n^p_{d,\beta}$ be the d-th dimensional $(d \in \{0,1\})$ lower $(\beta = 0)$ or upper $(\beta = 1)$ neighbor subdomain of n^p. We define that n's primary/secondary subcuboid has contact with that of $m \in F(n^p_{d,\beta})$ iff $[\zeta^l_p(n), \zeta^u_p(n)) \cap [\zeta^l_q(m), \zeta^u_q(m)) \neq \emptyset$ where $q = 0$ if $m = n^p_{d,\beta}$ or $q = 1$ otherwise, and denote the subset of $F(n^p_{d,\beta})$ whose member has such contacting subcuboids as $F^c_n(n^p_{d,\beta}) = \{f_0, \ldots\}$ such that $j < j'$ iff $\zeta^l_q(f_j) < \zeta^l_q(f_{j'})$. With these definitions, the element `VPlane`$[i(d,p,\beta,j)]$ has the transfer schedule for $f_j \in F^c_n(n^p_{d,\beta})$ where $i(d,p,\beta,j)$ is defined as follows and $p_n \in \{0,1\}$ is 1 iff the local node n will have secondary subdomain in the next step.

$$i(d,p,\beta,j) = \sum_{e=0}^{d-1} \sum_{q=0}^{p_n} \sum_{\gamma=0}^{1} \left| F^c_n(n^q_{e,\gamma}) \right| + \sum_{q=0}^{p-1} \sum_{\gamma=0}^{1} \left| F^c_n(n^q_{d,\gamma}) \right| + \sum_{\gamma=0}^{\beta-1} \left| F^c_n(n^p_{d,\gamma}) \right| + j$$

Note that the concepually 3-dimensional `int` array `VPlaneHead`$[2 \times 2 \times 2 + 1]$ has $i(d,p,\beta,0)$ in its element $[d][p][\beta]$ including the last one $[2][0][0]$ being the number of elements in `VPlane[]`.

`VPlane[]` is allocated by `init4s()` which also initializes `VPlaneHead`$[d][p][\beta]$ to be 0 for all $d \in \{0,1\}$, $p \in \{0,1\}$ and $\beta \in \{0,1\}$ as well as the last element $[2][0][0]$, in order to keep `oh4s_exchange_border_data()` from doing anything before the first call of `oh4s_transbound()`.

The size of `VPlane[]` being $N_V = 2N + 6$ is determined as follows. It is assured that $n^0_{d,\beta} \neq n^1_{d,\beta}$ for all d and β, but it can be $n^p_{d,\beta} = n^q_{e,\gamma}$ if $(d,\beta) \neq (e,\gamma)$, because a neighbor of primary subdomain may be also a neighbor of secondary one with different index and, more complicatedly, a subdomain may have two or more neighbor indecies with periodic system boundary. If $n^p_{d,\beta} = n^p$, $|F^c_n(n^p_{d,\beta})| = 1$ because n's subcuboid has only one contacting subcuboid being itself. Otherwise, $|F^c_n(n^p_{d,\beta})| \leq |F(n^p_{d,\beta})|$ obviously but they can be equal. Since `VPlane[]` must have $\sum_d \sum_p \sum_\beta |F^c_n(n^p_{d,\beta})|$ and $\bigcup_d \bigcup_p \bigcup_\beta F^c_n(n^p_{d,\beta})$ can be the set of all nodes $\mathcal{N}$, it looks $N_V = N + 2 \times 4 - 1$ because $\bigcup_d \bigcup_p \bigcup_\beta H^c_n(n^p_{d,\beta})$ can be $\mathcal{N} - \{r\}$, where $H^c_n(n^p_{d,\beta})$ is the set of helpers in $F^c_n(n^p_{d,\beta})$ and r is the root of the family tree, and we have four, i.e., west, east, south and north, neighbors for each of primary and secondary subdomains.

However as mentioned above, we can have neighbor duplication to make N_V larger. Let $\mathcal{M}(n)$ be the set of non-self neighbors of n, i.e.,

$$\mathcal{M}(n) = \{n^p_{d,\beta} \,|\, d \in \{0,1\},\ p \in \{0,1\},\ \beta \in \{0,1\},\ n^p_{d,\beta} \neq n^p\} = \{m_1, \ldots\}$$

$occ(m_i)$ be the number of occurrence of m_i as the neighbor of n^0 and n^1, i.e., $occ(m_i) = |\{n^p_{d,\beta} \,|\, m_i = n^p_{d,\beta} \neq n^p\}|$, and $occ(m_1)$ be the largest without loss of generality. Then N_V must be the maximum of $N^n_V = \sum_i occ(m_i)|F^c_n(m_i)|$ because the transfer schedule for $n^p_{d,\beta}$ is different from $n^q_{e,\gamma}$ even if $m_i = n^p_{d,\beta} = n^q_{e,\gamma}$. Since $\bigcup_i H^c_n(m_i) \subseteq \mathcal{N} - \{r\}$ again, $H^c_n(m_i) \cap H^c_n(m_j) = \emptyset$ for any $i \neq j$, and $F^c_n(m_i)$ can be $\mathcal{N}$ if $m_i = n^p_{d,\beta} \neq n^p$, N^n_V is maximized with given $\mathcal{M}(n)$ when $F^c_n(m_1) = \mathcal{N}$ to let

$$N_V = occ(m_1)N + \sum_{i>1} occ(m_i) + \left|\{n^p_{d,\beta} \,|\, n^p_{d,\beta} = n^p\}\right| = occ(m_1)N + (2 \times 4 - occ(m_1))$$

because $F^c_n(m_i) = \{m_i\}$ for all $i > 1$ due to $F^c_n(m_1) = \mathcal{N}$.

Therefore, N^n_V is maximized by maximizing $occ(m_1)$ letting it be 2 with $\Pi_x \times \Pi_y = 2$ and periodic system boundaries perpendicular to z-axis. Suppose $\Pi_x = 2$ and $\Pi_y = 1$ to let n's primary subdomain has west and east neighbors commonly m_1, while n's north and south neighbors are commonly n itself. Since $F(m_1) = F^c_n(m_1) = \mathcal{N}$ to include n in it, n's secondary subdomain is m_1 having m_1 itself as north and south neighbors and n as west and east neighbors. Therefore, $occ(m_1) = 2$ to make $N_V = 2N + (2 \times 4 - 2) = 2N + 6$.

Besides the initialization, `VPlane[]` is let have the transfer schedule by `make_bsend_sched()` and `make_brecv_sched()`, while their caller `make_bxfer_sched()` determines `VPlaneHead[]`, and then referred to by `xfer_boundary_particles_v()` and `exchange_border_data_v()` together with `VPlaneHead[]` to transfer particles and particle-associated data.

`BoundarySendBuf`

- The array `BoundarySendBuf`$[K]$ of `S_particle` structure where

$$K = 2\mathcal{D}\delta^{\max}_z((\delta^{\max}_y + 2e^g)(\delta^{\max}_x + 2e^g) - \delta^{\max}_x\delta^{\max}_y)$$

is for the set of $hbuf^s_v(d, p, \beta, m)$ for primary ($p = 0$) or secondary ($p = 1$) particles accmmodated by the local node n and sent to the node m in the family for the d-th dimensional ($d \in \{0, 1\}$) lower ($\beta = 0$) or upper ($\beta = 1$) neighbor of n's primary/secondary subdomain as m's halo particles. The size K represents the maximum number of particles in vertical exterior halo plane (not interior) of primary and secondary subcuboids, because we need to have two copies of each particle in interior pillars and a copy of each in exterior pillar. That is, by shifting each vertical interior halo plane by e^g grids outward for a neighbor contacting at a surface, and each exterior pillar by e^g grids along y-axis outward for relaying particles in them from a west/east neighbor to a south/north one, we have a grid-voxel set whose cardinality is equal to those in vertical exterior halo planes. To the array, particles are copied by `sort_particles()` and `sort_received_particles()` through the macro `Sort_Particle()`, by `move_and_sort()` through the macro `Move_Or_Do()`, and by `xfer_boundary_particles_v()` for exterior pillars. Then the particles in the array is sent to other nodes by `xfer_boundary_particles_v()` using the indices and number of particles recorded in `VPlane[]`.

`S_griddesc` `GridDesc`

- The array `GridDesc`[3] of the `S_griddesc` structure is foundamentally equivalent to that in level-4p described in §4.9.3. However, its elements `w`, `d` and `h` are enlarged from $\delta^{\max}_d + 4e^g$ to $\delta^{\max}_d + 6e^g$ because of the outside portion of sending planes of e^g thick and receiving planes of $2e^g$ thick, as discussed in §4.12.2.

  As in level-4p, each array element is set by `set_grid_descriptor()` called from `init4s()` for [0], `rebalance4s()` or `exchange_particles4s()` for [1], and `make_`

recv_list() for [2]. Then the array is referred to by the following functions directly (*1), through macros Allocate_NOfPGrid() (*2), For_All_Grid() (*3), For_All_Grid_Abs() (*4), For_All_Grid_Z() (*5), For_All_Grid_XY_At_Z()(*6), and/or Neighbor_Grid_Offset() (*7).

init4s()(*1,*2), oh4s_per_grid_histogram()(*1,*2), transbound4s()(*3), exchange_particles4s()(*3), exchange_population()(*3), add_population()(*4), make_recv_list()(*1), sched_recv()(*1), make_send_sched_body()(*1,*5), make_send_sched_self()(*1,*5), make_send_sched_hplane()(*6), update_neighbors()(*1,*7), count_population()(*3), make_bsend_sched()(*1,*5), make_brecv_sched()(*1,*5), sort_particles()(*3), move_and_sort()(*3), xfer_boundary_particles_v()(*1,*3), exchange_border_data_v()(*1,*3), oh4s_map_particle_to_neighbor()(*1), oh4s_map_particle_to_subdomain()(*1).

S_interiorp
InteriorParts

- The array InteriorParts[2][S] of the S_interiorp structure is level-4s's own and its element [p][s] specifies the contiguos subregion $pbuf_i(p,s)$ in $pbuf(p,s)$ in which non-halo particles are stored after the packing by move_to_sendbuf_4s() and particle reception by xfer_particles() when partially position-aware particle transfer takes place. The structure elements head and size have the head index and the size of $pbuf_i(p,s)$ respectively. The reason why we need this array is that, *unlike* level-4p, $pbuf(p,s)$ may not be fully filled because halo particles are not yet received and thus sort_particles() cannot scan all particles in $pbuf(p,s)$ but should do only those in $pbuf_i(p,s)$.

  After allocated by init4s(), the array is let have the values shown above by move_to_sendbuf_4s() and its callees move_to_sendbuf_uw4s() and move_to_sendbuf_dw4s() to define each of $pbuf_i(p,s)$, and then referred to by sort_particles() to scan the particles in $pbuf_i(p,s)$.

```
EXTERN int *PbufIndex;                                      /* [2*ns+1] */
EXTERN dint **NOfPGrid[2], **NOfPGridTotal[2];              /* [2][ns][z][y][x] */
EXTERN int **NOfPGridOut[2], **NOfPGridOutShadow[2];        /* [2][ns][z][y][x] */
EXTERN int **NOfPGridIndex[2], **NOfPGridIndexShadow[2];/* [2][ns][z][y][x] */
EXTERN dint *NOfPGridZ;                                     /* [z] */
EXTERN int ZBound[2][2], (*ZBoundShadow)[2];
struct S_hplane {
  int nbor, stag, rtag;
  int *nsend, *nrecv, *sbuf, *rbuf;                         /* [ns] */
};
EXTERN struct S_hplane HPlane[2][2];                        /* [2][2] */
struct S_vplane {
  int nbor, stag, rtag;
  int nsend, nrecv, sbuf, rbuf;
};
EXTERN struct S_vplane *VPlane;                             /* [2*nn+6] */
EXTERN int VPlaneHead[2*2*2+1];
EXTERN struct S_particle *BoundarySendBuf;

struct S_griddesc {
  int x, y, z, w, d, h, dw;
};
```

```
EXTERN struct S_griddesc GridDesc[3];

struct S_interiorp {
  int head, size;
};
EXTERN struct S_interiorp *InteriorParts;
```

In addition, we use the following variables a little bit diffenently from their usages in lower level libraries but similarly to level-4p.

`Particles` `SendBuf`

- As in level-4p, `Particles`$[P_{lim}]$ and `SendBuf`$[P_{lim}]$ are combined, and are used alternately. However the size of each P_{lim} is now calculated by `init4s()` a little bit differently as discussed later, and thus receiving the buffer from the simulator body or allocating it is done by level-4s's own function `oh4s_particle_buffer()`.

  The following functions refer to and/or modify both `Particles`[] and `SendBuf`[] directly or through macros.

  `move_to_sendbuf_4s()`, `move_to_sendbuf_uw4s()`, `move_to_sendbuf_dw4s()`, `sort_particles()`, `move_and_sort()`, `sort_received_particles()`, `xfer_particles()`, `xfer_boundary_particles_v()`.

  The function `xfer_boundary_particles_h()` refers to and modifies only `SendBuf`[], while the following functions refer to and/or modify only `Particles`[].

  `count_population()` `oh4s_map_particle_to_neighbor()`, `oh4s_map_particle_to_subdomain()`, `oh4s_inject_particle()`, `oh4s_remove_mapped_particle()`.

`nOfLocalPLimit` `nOfLocalPLimitShadow`

- *Unlike* level-4p, the variable `nOfLocalPLimitShadow` $= P'_{lim}$ is calculated by `init4s()` to let it be as follows.

$$
\begin{aligned}
P_{halo} &= \mathcal{D}((\delta_x^{\max}+2)(\delta_y^{\max}+2)(\delta_z^{\max}+2) - \delta_x^{\max}\delta_y^{\max}\delta_z^{\max}) \\
P_{mgn} &= \mathcal{D}\delta_x^{\max}\delta_y^{\max} \\
P'_{lim} &= \max(\lceil \overline{P}(100+\alpha)/100 \rceil,\ \overline{P} + \texttt{minmargin}) + 2(P_{halo} + P_{mgn})
\end{aligned}
$$

  This calculation ensures that each node can have up to $2P_{halo}$ halo particles for largest possible primary and secondary subcuboid, and that we have the margins of P_{mgn} for each of primary and secondary particle sets due to the coarse unit of particle assignment being those in a xy-plane of subdomain. The calculated value is also reported to the simulator body through the `maxlocalp` argument of `oh4s_init()`.

  Then $P_{lim} =$ `nOfLocalPLimit` is given by the simulator body through the argument `maxlocalp` of `oh4s_particle_buffer()` to let the library know the real amount and confirm that it is not less than P'_{lim}. The variable is referred to by level-4s functions `exchange_particles4s()`, `move_to_sendbuf_4s()` and `oh4s_inject_particle()`.

`NOfPLocal`

- As in level-4p, `NOfPLocal`$[2][S][N]$ is private to level-4s library and maintained by `oh4s_map_particle_to_neighbor()`, `oh4s_map_particle_to_subdomain()` and `oh4s_remove_mapped_particle()`. In level-4s library `NOfPLocal`[][][] is also referred to by `transbound4s()`.

RecvBufBases

- As in level-4p, the pointer array `RecvBufBases`[2][S] has one extra element conceptually [2][0] so that `sort_received_particles()` can know the tail of $rbuf(p, s)$. This extra element is set by `move_and_sort()` or `move_and_sort_secondary()` and referred to by `sort_received_particles()`, while other elements are referred to also by them and `move_to_sendbuf_4s()`, `move_to_sendbuf_uw4s()` and `xfer_particles()`.

- Besides `Particles`[], `SendBuf`[], `nOfLocalPLimit`, `NOfPLocal`[][][] and `RecvBufBases`[][], some other variables for particle buffers and population are also used in the level-4s functions in their original meanings as follows.

  - `TotalP`[][] in `transbound4s()`, `move_and_sort_primary()`, `move_to_sendbuf_uw4s()`, `move_to_sendbuf_dw4s()` and `move_and_sort()`.
  - `TotalPNext`[][] in `transbound4s()`, `count_population()`, `make_send_sched()`, `make_send_sched_hplane()`, `move_to_sendbuf_4s()`, `move_to_sendbuf_uw4s()`, `move_to_sendbuf_dw4s()` and `sort_particles()`.
  - `primaryParts` in `move_to_sendbuf_4s()` and `move_and_sort()` together with the pointer to its shadow `secondaryBase`, while in `count_population()` solitarily.
  - `totalParts` in `oh4s_particle_buffer()`, `transbound4s()`, `rebalance4s()`, `count_population()`, `move_to_sendbuf_4s()`, `oh4s_map_particle_to_neighbor()`, `oh4s_map_particle_to_subdomain()`, `oh4s_inject_particle()` and `oh4s_remove_mapped_particle()` directly, and through the macro `Check_Particle_Location()`. The function `transbound4s()` also refers to the pointer to the shadow `totalLocalParticles`.
  - `nOfInjections` in `transbound4s()`, `rebalance4s()`, `count_population()`, `move_to_sendbuf_4s()`, `move_and_sort()`, and `oh4s_inject_particle()` directly, and in `oh4s_map_particle_to_neighbor()` and `oh4s_map_particle_to_subdomain()` through the macro `Check_Particle_Location()`.
  - `InjectedParticles`[][] in `transbound4s()`, `rebalance4s()`, `oh4s_map_particle_to_neighbor()`, `oh4s_map_particle_to_subdomain()` and `oh4s_remove_mapped_particle()`.

nOfFields
FieldTypes
FieldDesc

- As in level-4p, `init4s()` *intercepts* its argument `ftypes` to make its substance `FieldTypes`[][] have the following additional entry $f = F - 1$ to its tail for per-grid histogram.

  - [0] = $\varepsilon(f) = 1$ means that an entry of per-grid histogram has one (always 64-bit integer, *unlike* level-4p) element.
  - [1:2] = $\{e_l(f), e_u(f)\} = \{0, 0\}$ means per-grid histogram does not have any special extensions, as in level-4p.
  - [3:4] = $\{e^b_l(f), e^b_u(f)\} = \{-e^g, e^g\}$ means the broadcast of per-grid histogram should include e^g thick extensions, *unlike* level-4p. These extensions are necessary to let helpers have helpand's sending or in other words exterior halo planes.
  - [5:6] = $\{e^r_l(f), e^r_u(f)\} = \{-e^g, e^g\}$ means the reduction of per-grid histogram should include its sending planes of e^g thick, as in level-4p.

On the other hand, the mechanism of initialization and adjustment of `FieldDesc[]` is same as that of level-4p. That is, `FieldDesc[]` is allocated and initialized by `init_fields()` called from `init3()` called from `init4s()`. Then `adjust_field_descriptor()` adds $(S-1)\prod_{d=0}^{D-1}\Phi_d(F-1) = (S-1)G$ to `FieldDesc`$[F-1]$`.{bc, red}.size`$[0]$ so that the broadcast and reduction for per-grid histogram are performed on the whole of $[p][S][G]$ rather than the one array element $[p][s][G]$. The function is called from `init4s()` and `update_descriptors()`, the latter of which is called from `exchange_particles4s()` and `make_recv_list()` on the helpand-helper reconfiguration with anywhere and normal accommodation respectively. It is also same as level-4p that `FieldDesc`$[F-1]$`.{bc,red}` are referred to by `reduce_population()` and `make_recv_list()` for reduction and broadcast of per-grid histogram respectively.

`nOfExc`
`BoundaryCommFields`
`BoundaryCommTypes`
`BorderExc`

- As in level-4p, `init4s()` *intercepts* its arguments `cfields` and `ctypes` to make their substances `BoundaryCommFields[]` and `BoundaryCommTypes[][][][]` have one additional entry $C-1$ for each, to have $F-1$ for the former and the followings for the latter.
  - *Unlike* level-4p, $[0][0][] = \{-e^g, e^g, 2e^g\}$ to mean that the sending planes of the downward communication are $2e^g$ thick and consist of the lower boundary plane(s) and the plane(s) just below it (or them), while receiving planes are just above the upper sending planes.
  - Also *unlike* level-4p, $[0][1][] = \{-e^g, -3e^g, 2e^g\}$ means that the sending planes of the upward communication are $2e^g$ thick and consist of the upper boundary plane(s) and the plane(s) just above it (or them), while receiving planes are just below the lower sending planes.

  On the other hand, `BoundaryCommTypes`$[C-1][b][][]$ for all $b \in [1, B)$ are set to 0 to mean no communication for non-periodical system boundaries, as in level-4p.

  It is also same as level-4p that `BorderExc[][][][]` is allocated and initialized by `init_fields()` called from `init3()` called from `init4s()`, and its element $[C-1][][][]$ is referred to by `oh3_exchange_borders()` called from `exchange_population()`.

4.12.4 Variables for Particle Transfer Scheduling

The next variable group is for the particle transfer scheduling. Before showing them, we revisit the following variables whose usages are slightly different from those in lower level libraries and from level-4p in some of them.

`S_commlist`
`CommList`
`SecRList`
`RLIndex`

- As done in the level-1 and level-4p library, we build the secondary mode particle transfer schedule in the array of `S_commlist` structure `CommList[]`. However, some of the structure elements have meanings different from those in level-1 *and* level-4p as follows.
  - `rid` is the ID r of the node by which particles specified by the record should be accommodated, as in level-1 and level-4p.
  - *Unlike* level-4p, `region` is the z-coordinate of the topmost xy-plane of the subcuboid assigned to r. That is, r will accommodate particles in the subcuboid whose z-coordinates are in $(z', z]$ where z' is `region` of the previous record or -1 if the record in question is the first one.
  - `tag` is 0 for primary particles of r or NS for its secondary ones, as in level-1 and level-4p.

– `count` is always 0, *unlike* level-4p because we don't have hot-spot records.
– `sid` is always 0, *unlike* level-4p because we don't have hot-spot records.

As for the blocks in `CommList`[], they are very similar to those in level-4p but the sizes are *differnt* from them and the alternative secondary receiving block is followed by alternative secondary sending block.

primary receiving block is build by each node for particles in its primary subdomain to be accommodated by the node itself or its helpers. Since for a subdomain n, the records for each member of $F(n)$ appear at most once, the size of this block is at most $|F(n)| \leq N$.

primary sending block is exchanged by neighboring node (subdomain) pairs. A node receives the whole primary receiving block from each neighbor for particles sent from the family members rooted by the node to the family members rooted by the neighbor. Since we avoid receiving a primary receiving block twice or more from a neighbor and a node can appear at most two primary receiving blocks as a helpand and a helper, the size of this block is at most $2N$.

As in level-4p, the element $[k]$ $(k < 3^D)$ of the integer array `RLIndex`$[3^D + 1]$ has the `CommList`'s index of the first record of primary receiving block received from the k-th neighbor node, or 0 if the node is the local node itself. Also as in level-4p, all elements for a node having multiple neighbor indices commonly have the index of the sole primary receiving block of the node, and `RLIndex`$[3^D]$ has the index of the record just following the primary sending block, or the combined size of primary receiving and primary sending blocks in other words.

secondary receiving and **secondary sending** blocks for a node are the copies of the primary receiving and primary sending block of its helpand which broadcasts the blocks to the helpers to show them particle accommodations for helpand's primary subdomain and its neighbors and thus helper's secondary subdomain and its neighbors, as in level-4p. Therefore, the size of this block is at most $3N$. The pointer `SecRList` points the head of the secondary receiving block, and `SecRLIndex`$[3^D + 1]$ shown later is the copy of `RLIndex`[] of the helpand, also as in level-4p.

alternative secondary receiving block for a node is the copy of primary receiving block given from the new helpand on helpand-helper reconfiguration as in level-1 and level-4p, but is *different* from them because it is followed by **alternative secondary sending** block being the copy of the primary sending block of the helpand. The reason why we need alternative secondary sending block is that the node should know subcuboids of the neighbor family members of its *new* helpand to build the transfer schedule of halo particles. The combined size of these blocks is at most $3N$. The pointer `AltSecRList` shown later points the head of alternative secondary receiving block as in level-4p, but we also have `AltSecRLIndex`$[3^D + 1]$ being the copy of `RLIndex`[] of the new helpand and having indices in alternative secondary receiving/sending blocks.

Note that we don't have hot-spot sending block because we have no hot-spots. Therefore, the total of the maximum size of each block is $9N$ being definitely less than $2 \cdot 3^D(NS + 1) + N(S + 3) \geq 10N$ that level-1 requires. Therefore, `init1()` may be unaware of the amount required for level-4s[78].

[78] The function `init1()` is still aware of the amount required for level-4p, but in 3-dimensional simulations only to which the level-4s library can be applied, level-1's requirement is always larger than level-4p's.

NOfSend
NOfRecv

- As in level-4p, we use `NOfSend`[2][S][N] for the per-receiver sending histogram so that its element [p][s][m] has the number of particles of species s which the local node should send to the receiver node m as m's (not the local node's) primary ($p = 0$) or secondary ($p = 1$) particles. Each element is accumulated by `make_send_sched_body()` using the macro `Make_Send_Sched_Body()`. Then we perform a hand-made all-to-all communication among neighboring family members in `exchange_xfer_amount()` to exchange `NOfSend`[] to have the per-sender receiving histogram in `NOfRecv`[2][S][N] so that its element [p][s][m] has the number of particles of species s which the local node should receive from the sender node m as the local node's (not m's) primary ($p = 0$) or secondary ($p = 1$) particles.

  Also as in level-4p, `NOfSend`[p][s][m] then acts as the index of a portion of `SendBuf`[], $sbuf(p, s, m)$, to which a particle of species s to be sent to m as m's primary ($p = 0$) or secondary ($p = 1$) particle is moved. This role change is done by `set_sendbuf_disps4s()`, and then the macro `Move_Or_Do()` used in `move_to_sendbuf_4s()`, `move_to_sendbuf_uw4s()`, `move_to_sendbuf_dw4s()` and `move_and_sort()` increments an element each time a particle is moved from `Particles`[] to `SendBuf`[] for sending.

  Then particles are sent in `xfer_particles()` referring to `NOfSend`[][][] for the send count, each element referred to and thus possibly having non-zero is zero-cleared for the accumulation in the next call of `transbound4s()`. All the entries, in addtion, are also zero-cleared in `init4s()` at the very beginning and before the first call of `transbound4s()`, and in `exchange_particles4s()` when we have anywhere accommodation with which `NOfSend`[] are modified by `exchange_particles()`.

  On the other hand, `NOfRecv`[][][] does not have any other roles because of no hot-spots.

Requests
Statuses

- The usage of `Requests`[] and `Statuses`[] to keep the requests/statuses of asynchronous MPI communications is not changed, and their required sizes for level-4s's own communications for halo particle (or particle-associated data) transfer is less than $4SN + 2 \cdot 3^D$, because those for particles in vertical halo planes and horizontal halo planes are up to $2 \times (2N + 6)$ and $(2 \times 2 \times 2)S$ respectively as discussed in §4.12.3[79].

  `Requests`[] and `Statuses`[] are referred to in;

  `exchange_xfer_amount()`, `xfer_particles()`, `xfer_boundary_particles_v()`, `xfer_boundary_particles_h()`, `exchange_border_data_v()` and `exchange_border_data_h()`.

Now we show the variables and `struct` data types for the particle transfer scheduling.

AltSecRList

- As in level-4p, the `S_commlist`-type pointer `AltSecRList` is let point the head of alternative secondary receiving block in `CommList`[] by `make_recv_list()`, but the block is followed by alternative secondary sending block *unlike* level-4p. Then it is referred to by `make_send_sched()` and `make_bxfer_sched()` directly, and by `make_send_sched_self()`, `make_bsend_sched()` and `make_brecv_sched()` through their arguments `rlist`.

PrimaryCommList

- The array of `S_commlist` structure `PrimaryCommList`[2][3^D] is level-4s's own to have the trivial receiving record for the k-th neighbor in its element [p][k] to be used when we will in primary mode in the next step to determine the node to which we send

[79] $8S$ can be larger than $4SN$ when $N = 1$ but, since the very single node cannot have secondary subdomain if so, the required amount is $4S = 4SN$.

primary ($p = 0$) and secondary ($p = 1$) particles. That is, `PrimaryCommList`$[p][k]$ has m_k = `Neighbors`$[p][3^D - k - 1]$ in `rid`, $\delta_z(m_k) - 1$ in `region` to mean m_k's subcuboid is the primary subdomain of m_k itself, and 0 in other elements particularly for `tag` to mean the record is for primary particles for m_k. This list is required to use `exchange_particles4s()` regardless of the next execution mode and thus even in the mode is primary. The array is let have the elements above in `update_neighbors()`, and is referred to by `exchange_particles4s()` directly and by `make_send_sched()`, `make_bxfer_sched()` and their callees through their `rlist` arguments.

`SecRLIndex`

- As in level-4p, the integer array `SecRLIndex`$[3^D + 1]$ has the index of secondary receiving and secondary sending blocks in `CommList`[] in its element $[k]$ for the k-th neighbor of the local node's helpand if $k < 3^D$, while the element $[3^D]$ has the index of the block following secondary sending block, i.e., alternative secondary receiving block, or the combined size of secondary receiving and secondary sending blocks in other words. The array is obtained from the helpand by its broadcast of `RLIndex`[] in `make_recv_list()`, and then is referred to by `exchange_particles4s()` directly and by `make_send_sched()` and `make_bxfer_sched()` through their `rlidx` arguments.

`AltSecRLIndex`

- The integer array `AltSecRLIndex`$[3^D + 1]$ is level-4s's own and has the index of alternative secondary receiving/sending blocks in `CommList`[] in its element $[k]$ ($k < 3^D$) for the k-th neighbor of the local node's *new* helpand when helpand-helper reconfiguration takes place, while the element $[3^D]$ has the size of the block. The necessity of this array was shown in the discussion of `CommList`[] in this section. The array is obtained from the helpand by its broadcast of `RLIndex`[] in `make_recv_list()`, and then is referred to by `make_send_sched()` and `make_bxfer_sched()`.

`PrimaryRLIndex`

- The integer array `PrimaryRLIndex`$[3^D]$ is level-4s's own and has the trivial index k in its element $[k]$ to show the record for the k-th neighbor is in `PrimaryCommList`$[p][k]$ as discussed above. The array is let have the elements above in `init4s()`, and is referred to by `exchange_particles4s()` directly and by `make_send_sched()` and `make_bxfer_sched()` through their `rlidx` arguments.

`S_recvsched_context`

- The `struct` named `S_recvsched_context` is to keep the execution context of the function `sched_recv()` as in level-4p, but its elements are *different* from level-4p as follows.
  - `z` are the local z-coordinate of the xy-plane of per-grid histogram to be vistied.
  - `nptotal` is the number of particles which have already processed, as in level-4p.
  - `nplimit` is the total number of particles which the nodes already visited are expected to accommodate by the balancing algorithm, as in level-4p.
  - `cptr` is the pointer to a record in `CommList`[] to be built, as in level-4p.

  The structure is initialized by the caller `make_recv_list()` and then referred to and updated by `sched_recv()`.

`T_Hgramhalf`

- The `MPI_Datatype` variable `T_Hgramhalf` is perfectly equivalent to its level-4p counterpart, and thus has the MPI data-type for a slice $[p][*][m]$ in `NOfSend`[][][] and `NOfRecv`[][][] to send/receive the particle populations the node m should accommodate as its primary ($p = 0$) or secondary ($p = 1$) particles. The value of this variable is created by `MPI_Type_vector()` called in `init4s()` so that the type has S elements with the stride of N, and is used in `exchange_xfer_amount()`.

Note that we have neither variables nor structures for hot-spots because we have none of them.

```
EXTERN struct S_commlist *AltSecRList, PrimaryCommList[2][OH_NEIGHBORS];
EXTERN int SecRLIndex[OH_NEIGHBORS+1], AltSecRLIndex[OH_NEIGHBORS+1];
EXTERN int PrimaryRLIndex[OH_NEIGHBORS];

struct S_recvsched_context {
  int z;
  dint nptotal, nplimit;
  struct S_commlist *cptr;
};
EXTERN MPI_Datatype T_Hgramhalf;
```

4.12.5 Variables for Neighboring Information

Next, we declare arrays to hold neighboring information. Since they are perfectly equivalent to their level-4p counterparts discussed in §4.9.5, we briefly discuss them focusing on the functions referring to them.

FirstNeighbor

- The element $[k]$ of the integer array `FirstNeighbor`$[3^D]$ has k if $m =$ `SrcNeighbors`$[k] \geq 0$ or $m = -N-1$ to mean the first occurrence, or k' such that $m = -($`SrcNeighbors`$[k']+1)$. The array is let have these values by `init4s()` and then is referred to by `make_recv_list()`.

GridOffset

- The array `GridOffset`$[2][3^D]$ has the offset $goff(k)$ to translate a grid-position of the k-th neighbor m of the local node n's primary ($p=0$) or secondary ($p=1$) subdomain $n^p = \{n, parent(n)\}[p]$ into the corresponding grid-position of n^p in the element $[p][k]$. The values $[p][]$ are initialized/updated when `Neighbors`$[p][]$ is initialized/updated by the function `update_neighbors()`, and then is referred to through the macro `Local_Grid_Position()` invokded in the macro `Move_Or_Do()` and in the function `oh4s_remove_mapped_particle()`.

S_realneighbor
RealDstNeighbors
RealSrcNeighbors

- The arrays `RealDstNeighbors`$[2][2]$ and `RealSrcNeighbors`$[2][2]$ of `S_realneighbor` structure have the sets of nodes in the neighboring families of the local node. The structure element `nbor`$[N]$ is the array of a node set and `n` has its cardinality. The arrays are allocated by `init4s()`, are updated by `update_real_neighbors()` and its callee `upd_real_nbr()`, and then are referred to by `exchange_xfer_amount()`, `move_and_sort()`, `set_sendbuf_disps4s()` and `xfer_particles()`.

On the other hand, we modified the definitions of the following arrays declared in level-1 library, not only in the manner done in level-4p but also in a level-4p's own way.

Neighbors
DstNeighbors
SrcNeighbors

- The element array $[2][]$ of `Neighbors`$[3][N]$ temporarily has the neighbors of the local node's helpand by `build_new_comm()`, as in level-4p. The added element is referred to by `upd_real_nbr()` to construct `RealSrcNeighbors`$[1][1]$ and then copied into `Neighbors`$[1][]$ by `rebalance4s()`. Besides this extra role, level-4s's `Neighbors`$[0][] =$ `DstNeighbors`$[]$ and `SrcNeighbors`$[]$ are *different* from that in lower levels and level-4p, because their elements should have $-(N+1)$ if the corresponding neighbors in level-1's definition are beyond non-periodic system boundaries. This modification is done by `init4s()` referring to the values set by `init1()`, while letting `Neighbors`$[1][]$ have the helpand's `Neighbors`$[0][]$ is done in the way perfectly equivalent to other

levels, i.e., by `build_new_comm()`. Besides these assignments, `Neighbors[][]` is referred to by `make_send_sched()`, `update_neighbors()` and `make_bxfer_sched()`, while `DstNeighbors[]` and `SrcNeighbors[]` are referred to by `make_recv_list()`.

`TempArray`

- The array `TempArray[]` has $4N$ elements for `update_real_neighbors()` and its callee `upd_real_nbr()`, while `init4s()` allocates it and uses its first $2N$ elements to build `DstNeighbors[]` = `Neighbors[0][]`, `SrcNeighbors[]` and `FirstNeighbor[]`.

```
EXTERN int FirstNeighbor[OH_NEIGHBORS], GridOffset[2][OH_NEIGHBORS];
struct S_realneighbor {
  int n, *nbor;
};
EXTERN struct S_realneighbor RealDstNeighbors[2][2], RealSrcNeighbors[2][2];
```

4.12.6 Variable for Boundary Condition

`BoundaryCondition`

The last variable `BoundaryCondition`$[D][2]$ is perfectly equivalent to level-4p's discussed in §4.9.6, and thus is the substance of the `oh4s_init()`'s argument `bcond` to have the system boundary conditions. The array is initialized in `init4s()` and is referred to by the macro `Map_To_Grid()` used in `oh4s_map_particle_to_subdomain()`.

```
EXTERN int BoundaryCondition[OH_DIMENSION][2];
```

4.12.7 Function Prototypes

The next and last block is to declare the prototypes of the API function pairs each of which consists of API for Fortran and C, as listed below with marks "[E]" for those equivalent to level-4p's, "[M]" for those different from level-4p's, and "[N]" for those newly introduced for level-4s.

- The function `oh4s_init[_]()` [M] initializes data strucutures of the level-4s and lower level libraries.
- The function `oh4s_particle_buffer[_]()` [N] defines P_{lim} and particle buffers `Particles[]` and `SendBuf[]`.
- The function `oh4s_per_grid_histogram[_]()` [M] defines arrays for per-grid histogram and per-grid index.
- The function `oh4s_transbound[_]()` [M] at first performs what its level-1 counterpart `oh1_transbound[_]()` does to have the fundamental particle assignment for load balancing, and then modifies it to have position-aware particle distribution by the level-4s's own particle transfer.
- The function `oh4s_exchange_border_data[_]()` [N] transfers halo part of particle-associated array data.
- The function `oh4s_map_particle_to_neighbor[_]()` [E] finds the subdomain in which a given particle resides, providing that the subdomain is a neighbor of the primary/secondary subdomain of the local node, and maintains per-subdomain and per-grid histograms of particle population.

- The function `oh4s_map_particle_to_subdomain[_]()` [E] finds the subdomain in which a given particle resides, allowing that the subdomain is not necessary to be a neighbor of the primary/secondary subdomain of the local node, and maintains per-subdomain and per-grid histograms of particle population.
- The function `oh4s_inject_particle[_]()` [E] injects a particle and place it at the bottom of `Particles[]` maintaining per-subdomain and per-grid histograms of particle population.
- The function `oh4s_remove_mapped_particle[_]()` [E] removes a particle which has been mapped to a subdomain or been injected into a subdomain.
- The function `oh4s_remap_particle_to_neighbor[_]()` [E] does what functions `oh4s_remove_mapped_particle()` and `oh4s_map_particle_to_neighbor()` do.
- The function `oh4s_remap_particle_to_subdomain[_]()` [E] does what functions `oh4s_remove_mapped_particle()` and `oh4s_map_particle_to_subdomain()` do.

As done in §4.2.11, §4.4.5, §4.6.6, and §4.9.7, prior to showing the function prototypes, we show the fifth and (so far) last part of the header files ohhelp_c.h for C-coded simulators and ohhelp_f.h for Fortran-coded ones, which defines the aliases of level-4s API functions. In the `#else` part of `#ifdef␣OH_LIB_LEVEL_4P`, at first they `#define` the aliases of API functions.

```
#else
#define \
oh_init(A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,A15,A16,A17,A18,
A19,A20,A21,A22,A23,A24,A25,A26) \
oh4s_init(A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,A15,A16,A17,A18,
A19,A20,A21,A22,A23,A24,A25,A26)
#define oh_particle_buffer(A1,A2) oh4s_per_grid_histogram(A1,A2)
#define oh_per_grid_histogram(A1,A2) oh4s_per_grid_histogram(A1,A2)
#define oh_transbound(A1,A2)      oh4s_transbound(A1,A2)
#define oh_exchange_border_data(A1,A2,A3,A4) \
        oh4s_exchange_border_data(A1,A2,A3,A4)
#define oh_map_particle_to_neighbor(A1,A2,A3) \
        oh4s_map_particle_to_neighbor(A1,A2,A3)
#define oh_map_particle_to_subdomain(A1,A2,A3) \
        oh4s_map_particle_to_subdomain(A1,A2,A3)
#define oh_inject_particle(A1,A2) oh4s_inject_particle(A1,A2)
#define oh_remove_mapped_particle(A1,A2,A3) \
        oh4s_remove_mapped_particle(A1,A2,A3)
#define oh_remap_particle_to_neighbor(A1,A2,A3) \
        oh4s_remap_particle_to_neighbor(A1,A2,A3)
#define oh_remap_particle_to_subdomain(A1,A2,A3) \
        oh4s_remap_particle_to_subdomain(A1,A2,A3)
```

Then ohhelp_c.h gives the prototypes of the functions above, which are also given in ohhelp4s.h[80], while their Fortran versions are given in oh_mod4s.F90 as shown in §3.8.

```
void oh4s_init(int **sdid, const int nspec, const int maxfrac,
               const long long int npmax, const int minmargin,
```

[80] Prototypes of `oh4s_init()` in ohhelp_c.h and ohhelp4s.h are slightly different, i.e., the type of its fourth argument `npmax` is `long long int` in the former, while in the latter is `dint`.

```
              const int maxdensity, int **totalp, int **pbase,
              int *maxlocalp, int *cbufsize, void *mycomm, int **nbor,
              int *pcoord, int **sdoms, int *scoord, const int nbound,
              int *bcond, int **bounds, int *ftypes, int *cfields,
              int *ctypes, int **fsizes, int **zbound,
              const int stats, const int repiter, const int verbose);
void oh4s_particle_buffer(const int maxlocalp, struct S_particle **pbuf);
void oh4s_per_grid_histogram(int **pghgram, int **pgindex);
int  oh4s_transbound(int currmode, int stats);
void oh4s_exchange_border_data(void *buf, void *sbuf, void *rbuf,
                               MPI_Datatype type);
int  oh4s_map_particle_to_neighbor(struct S_particle *part, const int ps,
                                   const int s);
int  oh4s_map_particle_to_subdomain(struct S_particle *part, const int ps,
                                    const int s);
int  oh4s_inject_particle(const struct S_particle *part, const int ps);
void oh4s_remove_mapped_particle(struct S_particle *part, const int ps,
                                 const int s);
int  oh4s_remap_particle_to_neighbor(struct S_particle *part, const int ps,
                                     const int s);
int  oh4s_remap_particle_to_subdomain(struct S_particle *part, const int ps,
                                      const int s);
```

Then ohhelp4s.h continues prototype declaration for Fortran API functions.

```
void oh4s_init_(int *sdid, const int *nspec, const int *maxfrac,
                const dint *npmax, const int *minmargin, const int *maxdensity,
                int *totalp, int *pbase, int *maxlocalp, int *cbufsize,
                struct S_mycommf *mycomm, int *nbor, int *pcoord, int *sdoms,
                int *scoord, const int *nbound, int *bcond, int *bounds,
                int *ftypes, int *cfields, int *ctypes, int *fsizes,
                int *zbound,
                const int *stats, const int *repiter, const int *verbose);
void oh4s_particle_buffer_(const int *maxlocalp, struct S_particle *pbuf);
void oh4s_per_grid_histogram_(int *pghgram, int *pgindex);
int  oh4s_transbound_(int *currmode, int *stats);
void oh4s_exchange_border_data_(void *buf, void *sbuf, void *rbuf, int *type);
int  oh4s_map_particle_to_neighbor_(struct S_particle *part, const int *ps,
                                    const int *s);
int  oh4s_map_particle_to_subdomain_(struct S_particle *part, const int *ps,
                                     const int *s);
int  oh4s_inject_particle_(const struct S_particle *part, const int *ps);
void oh4s_remove_mapped_particle_(struct S_particle *part, const int *ps,
                                  const int *s);
int  oh4s_remap_particle_to_neighbor_(struct S_particle *part, const int *ps,
                                      const int *s);
int  oh4s_remap_particle_to_subdomain_(struct S_particle *part, const int *ps,
                                       const int *s);
```

4.13 C Source File ohhelp4s.c

4.13.1 Header File Inclusion

The first job done in ohhelp4s.c is the inclusion of the header files ohhelp1.h, ohhelp2.h, ohhelp3.h and ohhelp4s.h. Before the inclusion of ohhelp1.h, ohhelp2.h and ohhelp3.h, we `#define` the macro `EXTERN` as `extern` so as to make variables declared in the files external, but after that we make it `#undef`'iend and then `#define` it as empty so as to provide variables declared in ohhelp4p.h with their homes, as discussed in §4.2.3.

```
#define EXTERN extern
#include "ohhelp1.h"
#include "ohhelp2.h"
#include "ohhelp3.h"
#undef  EXTERN
#define EXTERN
#include "ohhelp4s.h"
```

4.13.2 Function Prototypes

The next and last job to do prior to macro and function definitions is to declare the prototypes of the following functions private for the level-4s library. Note that the marks "[E]", "[M]" and "[N]" in the list are for the indications as same as those in §4.12.7.

- The function `init4s()` [M] is the body of `oh4s_init()`.
- The function `transbound4s()` [M] is the body of `oh4s_transbound()`.
- The function `try_primary4s()` [M] performs position-aware particle transfer in primary mode after calling its level-1 couterpart `try_primary1()` to check if we will be in primary mode in the next step.
- The function `try_stable4s()` [M] performs position-aware particle transfer in secondary mode after calling its level-1 couterpart `try_stable1()` to check if we can keep the helpand-helper configuration.
- The function `rebalance4s()` [M] performs position-aware particle transfer in secondary mode after calling its level-1 counterpart `rebalance1()` to establish a new helpand-helper configuration.
- The function `exchange_particles4s()` [M] is the core of position-aware particle transfer in both primary and secondary modes.
- The function `count_population()` [E] accumulates the number of particles in each grid-voxel in primary and secondary subdomains to have the local per-grid histogram.
- The function `exchange_population()` [M] gathers particle population in each grid-voxel to build per-grid histogram.
- The function `reduce_population()` [M] sums per-grid histograms of the family members.
- The function `add_population()` [E] adds particle populatoin in each grid-voxel in receiving planes to that in boundary planes.

- The function `make_recv_list()` [M] scans per-plane histogram to build primary receiving block, and then exchanges the block between neighbors to have primary sending block and broadcast them for secondary receiving/sending and alternative secondary receiving/sending blocks for helpers.
- The function `sched_recv()` [M] scans per-plane histogram to determine the subcuboid for each node.
- The function `make_send_sched()` [M] scans primary receiving/sending, secondary receiving/sending and alternative secondary receiving/sending blocks to determine the node to which the local node sends the particles which the node currently accommodates, the head index of `SendBuf[]` for particles in each grid-voxel and in the next step, and the transfer schedulde of particles in horizontal halo planes.
- The function `make_send_sched_body()` [M] scans a primary receiving block created by the local node itself, or that given from a neighbor, the helpand or a neighbor of the helpand to determine the node which accommodates the particles in each xy-plane.
- The function `make_send_sched_self()` [N] scans the primary receiving or secondary receiving block to determine the particle population in each grid-voxel in the local node's primary and secondary subcuboids, and the transfer schedule of halo particles in horizontal halo planes.
- The function `make_send_sched_hplane()` [N] scans grid-voxels in a xy-plane in the local node's subcuboid in the next step to have $\mathcal{P}_O(p, s, g)$ in the plane, and to determine the head index and the size of the send/receive buffer for halo particle transfer if the plane is a horizontal halo plane.
- The function `update_descriptors()` [E] updates elements in `FieldDesc[][][]` and `BorderExc[][][][]` for the secondary subdomain newly assigned to the local node by rebalancing.
- The function `update_neighbors()` [M] initializes/updates `AbsNeighbors[][]`, `GridOffset[][]` and `PrimaryCommList[][]` for the local node's primary or secondary subdomain.
- The function `set_grid_descriptor()` [M] sets an element of `GridDesc[]` according to a subdomain.
- The function `adjust_field_descriptor()` [E] adjusts `FieldDesc`$[F-1]$`.{bc,red}.size[]` for the broadcast and reduction of per-grid histogram.
- The function `update_real_neighbors()` [E] updates `RealDstNeighbors[][]` and `RealSrcNeighbors[][]`.
- The function `upd_real_nbr()` [E] updates an element array of `RealDstNeighbors[][]` or `RealSrcNeighbors[][]`.
- The function `exchange_xfer_amount()` [M] performs a hand-made all-to-all communication to send `NOfSend[][][]` and to receive it to `NOfRecv[][][]`.
- The function `make_bxfer_sched()` [N] scans primary receiving blocks created by neighbors of the local node or its helpand and grid-voxels in vertical halo planes to build transfer schedule for halo particles of the local node, its helpand, the neighbors of them, and the helpers of those nodes.

- The function `make_bsend_sched()` [N] scans a primary receiving block created by a neighbor of the local node or its helpand and grid-voxels in a vertical interior halo plane corresponding to the neighbor to build the sending schedule for halo particles of the neighbor and its helpers.
- The function `make_brecv_sched()` [N] scans a primary receiving block created by a neighbor of the local node or its helpand and grid-voxels in a vertical exterior halo plane corresponding to the neighbor to build the receiving schedule for halo particles of the local node from the neighbor and its helpers.
- The function `move_to_sendbuf_4s()` [M] is the level-4s counterpart of `move_to_sendbuf_primary()` and `move_to_sendbuf_secondary()` to move particles to be sent to `SendBuf[]` and pack those to stay in the local node in `Particles[]`.
- The function `move_to_sendbuf_uw4s()` [M] is the level-4s counterpart of `move_to_sendbuf_uw()` to move particles to be sent to `SendBuf[]` and pack those to stay in the local node shifting upward in `Particles[]`.
- The function `move_to_sendbuf_dw4s()` [M] is the level-4s counterpart of `move_to_sendbuf_dw()` to move particles to be sent to `SendBuf[]` and pack those to stay in the local node shifting downward in `Particles[]`.
- The function `sort_particles()` [M] performs bucket sorting on `Particles[]` to have sorted result in `SendBuf[]`.
- The function `move_and_sort()` [M] moves all particles in `Particles[]` to `SendBuf[]` sorting those staying in the local node.
- The function `sort_received_particles()` [M] moves all received particles in each $rbuf(p, s)$ to `SendBuf[]` sorting them.
- The function `set_sendbuf_disps4s()` [M] is the level-4s counterpart of `set_sendbuf_disps()` to updates entries of `NOfSend[][][]` so that each of its entry has the displacement of the head of $sbuf(p, s, m)$.
- The function `xfer_particles()` [M] performs a hand-made all-to-all communication to exchange particles.
- The function `xfer_boundary_particles_v()` [N] transfers particles in vertical halo planes.
- The function `xfer_boundary_particles_h()` [N] transfers particles in horizontal halo planes.
- The function `exchange_border_data_v()` [N] transfers particle-associated data in vertical halo planes.
- The function `exchange_border_data_h()` [N] transfers particle-associated data in horizontal halo planes.

```
static void init4s(int **sdid, const int nspec, const int maxfrac,
                   const dint npmax, const int minmargin, const int maxdensity,
                   int **totalp, int **pbase, int *maxlocalp, int *cbufsize,
                   struct S_mycommc *mycommc, struct S_mycommf *mycommf,
```

```
                  int **nbor, int *pcoord, int **sdoms, int *scoord,
                  const int nbound, int *bcond, int **bounds, int *ftypes,
                  int *cfields, const int cfid, int *ctypes, int **fsizes,
                  int **zbound,
                  const int stats, const int repiter, const int verbose);
static int  transbound4s(int currmode, int stats, const int level);
static int  try_primary4s(const int currmode, const int level,
                          const int stats);
static int  try_stable4s(const int currmode, const int level, const int stats);
static void rebalance4s(const int currmode, const int level, const int stats);
static void exchange_particles4s(int currmode, const int nextmode,
                                 const int level, int reb, int oldp, int newp,
                                 const int stats);
static void count_population(const int nextmode, const int psnew,
                             const int stats);
static void exchange_population(const int currmode);
static void reduce_population();
static void add_population(dint *npd, const int xl, const int xu,
                           const int yl, const int yu, const int zl,
                           const int zu, const int src);
static void make_recv_list(const int currmode, const int level, const int reb,
                           const int oldp, const int newp, const int stats);
static void sched_recv(const int reb, const int get, const int stay,
                       const int nid, const int tag,
                       struct S_recvsched_context *context);
static void make_send_sched(const int reb, const int pcode, const int oldp,
                            const int newp, struct S_commlist *rlist[2],
                            int *rlidx[2], int *nacc, int *nsendptr);
static int  make_send_sched_body(const int ps, const int n, const int sdid,
                                 struct S_commlist *rlist);
static void make_send_sched_self(const int psor2, struct S_commlist *rlist,
                                 int *naccptr);
static void make_send_sched_hplane(const int psor2, const int z, int *naccptr,
                                   int *np, int *buf);
static void update_descriptors(const int oldp, const int newp);
static void update_neighbors(const int ps);
static void set_grid_descriptor(const int idx, const int nid);
static void adjust_field_descriptor(const int ps);
static void update_real_neighbors(const int mode, const int dosec,
                                  const int oldp, const int newp);
static void upd_real_nbr(const int root, const int psp, const int pss,
                         const int nbr, const int dosec, struct S_node *node,
                         struct S_realneighbor rnbrptr[2], int *occur[2]);
static void exchange_xfer_amount(const int trans, const int psnew,
                                 const int nextmode);
static void make_bxfer_sched(const int trans, const int psnew,
                             struct S_commlist *rlist[2], int *rlidx[2]);
static void make_bsend_sched(const int psor2, const int n, const int nx,
                             const int ny, struct S_commlist *rlist,
                             int *nsendptr, int *vpptr);
static void make_brecv_sched(const int psor2, const int n, const int nx,
                             const int ny, struct S_commlist *rlist,
                             int *nrecvptr, int vpidx);
static void move_to_sendbuf_4s(const int nextmode, const int psold,
```

```
                                const int psnew, const int trans,
                                const int oldp, const int *nacc,
                                const int nsend, const int stats);
static void move_to_sendbuf_uw4s(const int ps, const int mysd, const int cbase,
                                 const int nbase);
static void move_to_sendbuf_dw4s(const int ps, const int mysd, const int ctail,
                                 const int ntail);
static void sort_particles(const int nextmode, const int psnew,
                           const int stats);
static void move_and_sort(const int nextmode, const int psold, const int psnew,
                          const int oldp, const int *nacc, const int stats);
static void sort_received_particles(const int nextmode, const int psnew,
                                    const int stats);
static void set_sendbuf_disps4s(const int nextmode, const int trans);
static void xfer_particles(const int trans, const int psnew,
                           const int nextmode, struct S_particle *sbuf);
static void xfer_boundary_particles_v(const int psnew, const int pcode,
                                      const int d);
static void xfer_boundary_particles_h(const int psnew);
static void exchange_border_data_v(void *buf, void *sbuf, void *rbuf,
                                   MPI_Datatype type, const MPI_Aint esize,
                                   const int d);
static void exchange_border_data_h(void *buf, MPI_Datatype type,
                                   const MPI_Aint esize);
```

In addition, we use the following lower level library functions, the set of which is equivalent to level-4p's.

- The function `mem_alloc()` allocates a memory space by `malloc()`. It is called from `init4s()` directly or through the macro `Allocate_NOfPGrid()`, `oh4s_per_grid_histogram()` through the macro, `oh4s_particle_buffer()` and `transbound4s()`.
- The function `mem_alloc_error()` aborts the simulation due to the memory shortage reporting its cause. It is called from `init4s()`.
- The function `errstop()` aborts the simulation due to an error detected by all processes reporting given error message. It is called from `init4s()` and `oh4s_particle_buffer()`.
- The function `local_errstop()` aborts the simulation due to an error detected by the local process reporting given error message. It is called from `sched_recv()` and `oh4s_inject_particle()` directly and from `oh4s_map_particle_to_neighbor()`, `oh4s_map_particle_to_subdomain()` and `oh4s_remove_mapped_particle()` through the macro `Check_Particle_Location()`.
- The function `transbound1()` is the body of `oh1_transbound()`. It is called from `transbound4s()`.
- The function `try_primary1()` is to examine whether particle distribution among subdomains is balanced well and thus we can perform the simulation in primary mode. It is called from `try_primary4s()`.
- The function `try_stable1()` is to examine whether particle distribution among nodes is balanced well and thus we can keep the current helpand-helper configuration. It is called from `try_stable4s()`.

- The function `rebalance1()` is to (re)build the helpand-helper configuration to cope with an unacceptable load imbalance. It is called from `rebalance4s()`.
- The function `build_new_comm()` is to build communicators for the helpand-helper families built by `rebalance1()`. It is called from `make_recv_list()`.
- The function `exchange_primary_particles()` is the core of the particle transfer in primary mode. It is called from `exchange_particles4s()`.
- The function `move_to_sendbuf_primary()` moves particles to be transferred from `Particles[]` to `SendBuf[]` and packs those remaining in `Particles[]` in primary mode. It is called from `exchange_particles4s()`.
- The function `exchange_particles()` is the core of the particle transfer in secondary mode. It is called from `exchange_particles4s()`.
- The function `init3()` is the body of `oh3_init()`. It is called from `init4s()`.
- The function `set_field_descriptors()` sets `FieldDesc`$[f]$`.{bc,red}.size`$[p]$ for all $f \in [0, F)$ and given $p \in \{0, 1\}$. It is called from `update_descriptors()`.
- The function `clear_border_exchange()` initializes `BorderExc`$[c][1][d][\beta]$`.{send, recv}` for all $c \in [0, C)$, $d \in [0, D)$ and $\beta \in \{0, 1\}$, or reinitializes them for the subdomain which the local node has had as the secondary one but discarded by rebalancing. It is called from `update_descriptors()`.
- The function `map_irregular_subdomain()` finds the subdomain of irregular process coordinate in which a particle resides. It is called from `oh4s_map_particle_to_subdomain()`.

4.13.3 Macros `If_Dim()`, `For_Y()`, `For_Z()`, `Do_Y()`, `Do_Z()` and `Coord_To_Index()`

Before starting to define functions, we define macros generally used in level-4s functions. The first group is for dimension dpendent operations in level-4p, but the macros in this group are *different* from level-4p's and are definitely expanded to those for $D = 3$ in level-4s. The reason why we keep these macro giving definitions (almost) equivalent to those of $D = 3$ case in level-4p is to minimize the difference between level-4s and level-4p codes.

`If_Dim()` The macro `If_Dim`(d, e_t, e_f) is always expanded to e_t regardless of d. The macro is used in `init4s()`, `set_grid_descriptor()` and `oh4s_map_particle_to_subdomain()`.

`For_Y()` `For_Z()` The macro `For_Y`(i, c, n) and `For_Z`(i, c, n) are expanded to the for-loop header `for(`i`;`c`;`n`)` to construct a for-loop for the dimension 2 (y) or 3 (z). They are commonly used in macros `For_All_Grid()`, `For_All_Grid_Abs()` and `For_All_Grid_XY_At_Z()`, while `For_Y()` is used also in `For_All_Grid_XY()` and `For_Z()` is used in `For_All_Grid_Z()`.

`Do_Z()` `Do_Y()` The macro `Do_Y`(a) and `Do_Z`(a) are expanded to a. They are used in `update_neighbors()`, `oh4s_map_particle_to_neighbor()` and `oh4s_map_particle_to_subdomain()`.

`Coord_To_Index()` The macro `Coord_To_Index`$(x, y, z, w, d{\cdot}w)$ is expanded to $x + y \cdot w + z \cdot d \cdot w$ to give the one-dimensional index of the element $[z][y][x]$ in a (conceptual) D-dimensional array of $[h][d][w]$, i.e., $gidx(x, y, z)$. The macro is used in macros `For_All_Grid()`, `For_All_Grid_Abs()`, `For_All_Grid_Z()`, `For_All_Grid_XY_At_Z()` and `Allocate_NOfPGrid()`, and functions `init4s()`, `update_neighbors()`, `oh4s_map_particle_to_neighbor()` and `oh4s_map_particle_to_subdomain()`.

```
#define If_Dim(D, ET, EF)  (ET)
#define For_Y(LINIT, LCONT, LNEXT) for(LINIT; LCONT; LNEXT)
#define For_Z(LINIT, LCONT, LNEXT) for(LINIT; LCONT; LNEXT)
#define Do_Y(ACT) ACT
#define Do_Z(ACT) ACT
#define Coord_To_Index(GX, GY, GZ, W, DW)  ((GX) + (GY)*(W) + (GZ)*(DW))
```

4.13.4 Macros `Decl_For_All_Grid()`, `For_All_Grid()`, `For_All_Grid_Abs()`, `The_Grid()`, `Grid_X()`, `Grid_Y()` and `Grid_Z()`

The next group of generally used macros are for traversing per-grid histogram. Since they are perfectly equivalent to those in level-4p, we briefly discuss them focusing on the functions using them.

`Decl_For_All_Grid()` The macro `Decl_For_All_Grid()` declares local variables `fag_`v used in `For_All_Grid()`, `For_All_Grid_Abs()`, `For_All_Grid_Z()`, `For_All_Grid_XY()` and `For_All_Grid_XY_At_Z()`. The macro is used in functions that use the macros listed above.

`For_All_Grid()` `For_All_Grid_Abs()` The macro `For_All_Grid`$(p, x_0, y_0, z_0, x_1, y_1, z_1)$ constracts nested for-loops to traverse grid-voxels (x, y, z) in the per-grid histogram of local node n's primary ($p = 0$) or secondary ($p = 1$) subdomains, where $x \in [x_0, \delta_x(n^p)+x_1)$, $y \in [y_0, \delta_y(n^p)+y_1)$ and $z \in [z_0, \delta_z(n^p)+z_1)$, and $n^p = \{n, parent(n)\}[p]$. The macro `For_All_Grid_Abs`$(p, x_0, y_0, z_0, x_1, y_1, z_1)$ acts similarly but the ranges are $x \in [x_0, x_1)$, $y \in [y_0, y_1)$ and $z \in [z_0, z_1)$.

The macro `For_All_Grid()` is used in `transbound4s()`, `exchange_particles4s()`, `count_population()`, `exchange_population()`, `sort_particles()`, `move_and_sort()`, while `For_All_Grid_Abs()` is used solely in `add_population()`. Note that we have level-4s's own relatives `For_All_Grid_Z()`, `For_All_Grid_XY()` and `For_All_Grid_XY_At_Z()` defined afterward.

`The_Grid()` `Grid_X()` `Grid_Y()` `Grid_Z()` The macro `The_Grid()` is to use in the body part of `For_All_Grid()` and its relatives to give $gidx(x, y, z)$ stored in `fag_gx` but without referring to the special variable name. The other special variables `fag_xidx`, `fag_yidx` and `fag_zidx` for x, y and z can be also referred to by the macros `Grid_X()`, `Grid_Y()` and `Grid_Z()`. The macro `The_Grid()` is used in all functions using `For_All_Grid()` and its relatives except for `For_All_Grid_Z()`. The functions `exchange_population()`, `make_send_sched_self()` and `make_brecv_sched()` also uses `Grid_Z()`, while `make_bsend_sched()` uses `Grid_X()` and `Grid_Z()`.

```
#define Decl_For_All_Grid()\
  int fag_x1, fag_y1, fag_z1;\
  int fag_xidx, fag_yidx, fag_zidx, fag_gx, fag_gy, fag_gz;\
  int fag_w, fag_dw;
#define For_All_Grid(PS, X0, Y0, Z0, X1, Y1, Z1)\
  For_Z((fag_zidx=(Z0), fag_x1=GridDesc[PS].x+(X1),\
         fag_y1=GridDesc[PS].y+(Y1), fag_z1=GridDesc[PS].z+(Z1),\
         fag_w=GridDesc[PS].w, fag_dw=GridDesc[PS].dw,\
         fag_gz=Coord_To_Index(X0,Y0,Z0,fag_w,fag_dw)),\
        (fag_zidx<fag_z1), (fag_zidx++,fag_gz+=fag_dw))\
    For_Y((fag_yidx=(Y0), fag_gy=fag_gz),\
          (fag_yidx<fag_y1), (fag_yidx++,fag_gy+=fag_w))\
      for (fag_xidx=(X0),fag_gx=fag_gy; fag_xidx<fag_x1; fag_xidx++,fag_gx++)
#define For_All_Grid_Abs(PS, X0, Y0, Z0, X1, Y1, Z1)\
```

```
  For_Z((fag_zidx=(Z0), fag_x1=(X1), fag_y1=(Y1), fag_z1=(Z1),\
         fag_w=GridDesc[PS].w, fag_dw=GridDesc[PS].dw,\
         fag_gz=Coord_To_Index(X0,Y0,Z0,fag_w,fag_dw)),\
        (fag_zidx<fag_z1), (fag_zidx++,fag_gz+=fag_dw))\
    For_Y((fag_yidx=(Y0), fag_gy=fag_gz),\
          (fag_yidx<fag_y1), (fag_yidx++,fag_gy+=fag_w))\
      for (fag_xidx=(X0),fag_gx=fag_gy; fag_xidx<fag_x1; fag_xidx++,fag_gx++)
#define The_Grid()  (fag_gx)
#define Grid_X()  (fag_xidx)
#define Grid_Y()  (fag_yidx)
#define Grid_Z()  (fag_zidx)
```

4.13.5 Constants URN_PRI, URN_SEC and URN_TRN

URN_PRI
URN_SEC
URN_TRN

The last group of macro definitions for constants of `update_real_neighbors()`'s operation mode is also equivalent to that in level-4p. To the function, `URN_PRI` = 0 to turn to primary mode is given by `init4s()` or `try_primary4s()`, `URN_SEC` = 1 on helper-helpand reconfiguration with anywhere accommodatation is given by `exchange_particles4s()`, and `URN_TRN` = 2 meaning the awareness of transitional state of helper-helpand configuration is given by `make_recv_list()`.

```
#define URN_PRI 0
#define URN_SEC 1
#define URN_TRN 2
```

4.13.6 oh4s_init() and init4s()

oh4s_init_()
oh4s_init()

The API functions `oh4s_init_()` for Fortran and `oh4s_init()` for C receive a set of array/structure variables through which level-1 to level-4s library functions communicate with the simulator body, and a few integer parameters to specify the behavior of the library. The argument set of the functions are *different* from that of level-4p counterparts `oh4p_init[_]()` described in §4.10.6 as follows.

- New arguments `npmax`, `minmargin` and `maxdensity` = $\mathcal{D}$ are added to calculate `maxlocalp` = P'_{lim} which is input argument in level-2/3/4p but is output in level-4s. Since the margin factors P_{halo} and P_{mgn} to determine P'_{lim} and thus P_{lim} depends on the largest subdomain size $\delta_d^{\max}$, we cannot calculate P'_{lim} prior to `oh4s_init()` while it is done in level-2/3 with `oh2_max_local_particles()` and in level-4p with `oh4p_max_local_particles()`. Therefore, we let `init4s()`, being the body of `oh4s_init()`, calculate P'_{lim} adding these three arguments and changing the role of `maxlocalp`. This also eliminates `pbuf` from the argument set because `init4s()` does not have P_{lim} possibly greater than P'_{lim}, and thus the association `pbuf` to `Particles[]` and `SendBuf[]` is done by `oh4s_particle_buffer()` called by simulator body after `oh4s_init()`.
- The argument `cbufsize` is added to report P_{comm} being the required size of send/receive buffers for halo part transfer of particle-associated arrays.
- The argument `zbound`[2][2] is added to associate an array in simulator body to `ZBoundShadow`[2][2] for $\zeta_p^{\beta}(n)$.

On the other hand, the argument addition and modification for the call of `init4s()` and setting `specBase` to 0 or 1 are same as those in `oh4s_init[_]()` discussed in §4.10.6.

```
void
oh4s_init_(int *sdid, const int *nspec, const int *maxfrac, const dint *npmax,
           const int *minmargin, const int *maxdensity, int *totalp,
           int *pbase, int *maxlocalp, int *cbufsize, struct S_mycommf *mycomm,
           int *nbor, int *pcoord, int *sdoms, int *scoord, const int *nbound,
           int *bcond, int *bounds, int *ftypes, int *cfields, int *ctypes,
           int *fsizes, int *zbound,
           const int *stats, const int *repiter, const int *verbose) {
  specBase = 1;
  init4s(&sdid, *nspec, *maxfrac, *npmax, *minmargin, *maxdensity, &totalp,
         &pbase, maxlocalp, cbufsize, NULL, mycomm, &nbor, pcoord, &sdoms,
         scoord, *nbound, bcond, &bounds, ftypes, cfields, -1, ctypes, &fsizes,
         &zbound,
         *stats, *repiter, *verbose);
}
void
oh4s_init(int **sdid, const int nspec, const int maxfrac, const dint npmax,
          const int minmargin, const int maxdensity, int **totalp,
          int **pbase, int *maxlocalp, int *cbufsize, void *mycomm,
          int **nbor, int *pcoord, int **sdoms, int *scoord, const int nbound,
          int *bcond, int **bounds, int *ftypes, int *cfields, int *ctypes,
          int **fsizes, int **zbound,
          const int stats, const int repiter, const int verbose) {
  specBase = 0;
  init4s(sdid, nspec, maxfrac, npmax, minmargin, maxdensity, totalp,
         pbase, maxlocalp, cbufsize, (struct S_mycommc*)mycomm, NULL, nbor,
         pcoord, sdoms, scoord, nbound, bcond, bounds, ftypes, cfields, 0,
         ctypes, fsizes, zbound,
         stats, repiter, verbose);
}
```

`Allocate_NOfPGrid()` As done in level-4p, we define the macro `Allocate_NOfPGrid`(π, h, t, σ, ν) used in `init4s()` to allocate and initialize a per-grid histogram, namely `NOfPGrid`$[2][S][\sigma]$ and `NOfPGridTotal`$[2][S][\sigma]$ of $t =$ `dint`. The definition is almost equivalent to that in level-4p described in §4.10.6 but the offset from the first element of the body arrays of $[p][s][\sigma]$ to the element corresponding to $(0, 0, 0)$ is *differently* given by $gidx(3e^g, 3e^g, 3e^g)$ rather than $gidx(2e^g, 2e^g, 2e^g)$ for level-4p, because we have $3e^g$ thick planes in the exterior of a subdomain for $2e^g$ thick receiving planes and e^g thick outside sending planes. Also as in level-4p, the macro has the sole other user `oh4s_per_grid_histogram()` with $t =$ `int`, but the function uses the macro not only for `NOfPGridOut`[][][] but also for `NOfPGridOutShadow`[][][], `NOfPGridIndex`[][][] and `NOfPGridIndexShadow`[][][].

```
#define Allocate_NOfPGrid(BODY, NPG, TYPE, SIZE, MSG) {\
  const int ns2 = nOfSpecies<<1;\
  const int gridsize = SIZE;\
  TYPE *npg = BODY;\
  TYPE **npgp = (TYPE**)mem_alloc(sizeof(TYPE*), ns2, MSG);\
  int s, g, exto=OH_PGRID_EXT*3;\
  const int base = Coord_To_Index(exto, exto, exto,\
```

```
                                        GridDesc[0].w, GridDesc[0].dw);\
  if (!npg)\
    BODY = npg = (TYPE*)mem_alloc(sizeof(TYPE), ns2*gridsize, MSG) + base;\
  for (s=0; s<ns2; s++,npg+=gridsize) {\
    npgp[s] = npg;\
    for (g=0; g<gridsize; g++)  npg[g-base] = 0;\
  }\
  NPG[0] = npgp;  NPG[1] = npgp + nOfSpecies;\
}
```

`nOfLocalPLimitShadow` As in level-4p, we declare a global variable `nOfLocalPLimitShadow` private to ohhelp4s.c to keep P'_{lim}. However, the functions referring to the variable are *different*; it is (re)initialized by `init4s()`, and then referred to by `oh4s_particle_buffer()` to confirm that the function is called after `oh4s_init()` and the argument `maxlocalp` $= P_{lim}$ given to the function is not less than P'_{lim} stored in the variable.

`init4s()` The function `init4s()`, called from `oh4s_init[_]()` implements the initialization for those API functions. The arguments of the function are almost same as `oh4s_init()` but its `mycomm` is split into two arguments `mycommc` and `mycommf` and there is an additional argument `cfid` as discussed in §4.7.3.

```
static int nOfLocalPLimitShadow = -1;
static void
init4s(int **sdid, const int nspec, const int maxfrac, const dint npmax,
       const int minmargin, const int maxdensity, int **totalp, int **pbase,
       int *maxlocalp, int *cbufsize, struct S_mycommc *mycommc,
       struct S_mycommf *mycommf, int **nbor, int *pcoord, int **sdoms,
       int *scoord, const int nbound, int *bcond, int **bounds, int *ftypes,
       int *cfields, const int cfid, int *ctypes, int **fsizes, int **zbound,
       const int stats, const int repiter, const int verbose) {
  int nn, me, nnns, nnns2, n;
  int (*ft)[OH_FTYPE_N] = (int(*)[OH_FTYPE_N])ftypes;
  int *cf = cfields;
  int (*ct)[2][OH_CTYPE_N] = (int(*)[2][OH_CTYPE_N])ctypes;
  int nf, ne, c, b, size, ps, s, tr, i, x, y, z;
  int *nphgram = NULL;
  int *rnbr, *iptr;
  dint *npgdummy = NULL,  *npgtdummy = NULL;
  int loggrid;
  dint idmax;
  const int ext = OH_PGRID_EXT, ext2 = ext<<1, ext3 = ext*3;
  struct S_particle pbufdummy, *pbufdummyptr = &pbufdummy;
  dint npl;
```

The structure of `init4s()` is similar to `init4p()` described in §4.10.6 and some of its portions are equivalent to those of the counterpart. However the function has various *difference* from the counterpart of course for level-4s's own initialization, and the first *difference* appears its very beginning. That is, at first we check if $D = 3$ because the level-4s extension is only for 3-dimensional simulations, and if $e^g = 1$ because having horizontal halo planes of two or more grids thick is not easy to implement or we need;

- to send/receive particles to/from two or more nodes for the lower or upper set of horizontal halo planes, or;

- to restrict the height of a subcuboid to be e^g or larger in order to make the communication above performed with only one node.

Though removing the restriction on e^g is not extremely tough especially with the second solution above and only a few functions need the restriction, we so far abandon to cope with cases of $e^g > 1$ because it is very unlike that a simulator requires $e^g > 1$. Therefore, we confirm that both conditions are satisfied, or abort execution by `errstop()` if either of them does not hold.

```
if (OH_DIMENSION!=3)
  errstop("dimension size %d is not 3 which level-4s extension requires.",
          OH_DIMENSION);
if (OH_PGRID_EXT!=1)
  errsotp("boundary plane thickness %d is not 1 which level-4s extension "
          "requires.", OH_PGRID_EXT);
```

The next part is equivalent to the first half of the corresponding part of `init4p()` to get N by `MPI_Comm_size()` and then to allocate `TempArray`$[4][N]$ by `mem_alloc()`. However, the second half is *eliminated* because the association of `pbuf` in simulator body to `Particles`[] and `SendBuf`[] is done by `oh4s_particle_buffer()` in level-4s.

```
MPI_Comm_size(MCW, &nn);  nnns = nn * nspec;  nnns2 = nnns << 1;
TempArray = (int*)mem_alloc(sizeof(int), nn<<2, "TempArray");
```

The next part is very similar to that in `init4p()` to *intercept* arguments `ftypes`, `cfields` and `ctypes` so that their substances `FieldTypes`[], `BoundaryCommFields`[] and `BoundaryCommTypes`[][][][] have one additional element for each for per-grid histogram. However, their additional last elements are *different* from those in level-4p and have the followings as discussed in §4.12.3.

$$\begin{aligned}
\texttt{FieldTypes}[F-1][] &= \{1, 0, 0, -e^g, e^g, -e^g, e^g\} \\
\texttt{BoundaryCommFields}[C-1] &= F - 1 \\
\texttt{BoundaryCommTypes}[C-1][b][][] &= \begin{cases} \{\{-e^g, e^g, 2e^g\}, \{-e^g, -3e^g, 2e^g\}\} & b = 0 \\ \{\{0,0,0\}, \{0,0,0\}\} & b > 0 \end{cases}
\end{aligned}$$

```
for (nf=0; ft[nf][OH_FTYPE_ES]>0; nf++);
for (ne=0; cf[ne]+cfid>=0; ne++);
FieldTypes = (int(*)[OH_FTYPE_N])
             mem_alloc(sizeof(int), (nf+2)*OH_FTYPE_N, "FieldTypes");
BoundaryCommFields = cf =
  (int(*))mem_alloc(sizeof(int), ne+2, "BoundaryCommFields");
BoundaryCommTypes = (int(*)[2][OH_CTYPE_N])
                    mem_alloc(sizeof(int), (ne+1)*nbound*2*OH_CTYPE_N,
                              "BoundaryCommTypes");
memcpy(FieldTypes, ft, sizeof(int)*nf*OH_FTYPE_N);
for (c=0; c<ne; c++)  cf[c] = cfields[c] + cfid;
memcpy(BoundaryCommTypes, ct, sizeof(int)*ne*nbound*2*OH_CTYPE_N);
ft = FieldTypes;  ct = BoundaryCommTypes + ne * nbound;
ft[nf][OH_FTYPE_ES] = 1;
ft[nf][OH_FTYPE_LO] = 0;  ft[nf][OH_FTYPE_UP] = 0;
ft[nf][OH_FTYPE_BL] = -ext;  ft[nf][OH_FTYPE_BU] = ext;
```

```
ft[nf][OH_FTYPE_RL] = -ext;  ft[nf][OH_FTYPE_RU] = ext;
ft[nf+1][OH_FTYPE_ES] = -1;
cf[ne] = nf;  cf[ne+1] = -1;
ct[0][OH_LOWER][OH_CTYPE_FROM] = -ext;
ct[0][OH_LOWER][OH_CTYPE_TO]   = ext;
ct[0][OH_LOWER][OH_CTYPE_SIZE] = ext2;
ct[0][OH_UPPER][OH_CTYPE_FROM] = -ext;
ct[0][OH_UPPER][OH_CTYPE_TO]   = -ext3;
ct[0][OH_UPPER][OH_CTYPE_SIZE] = ext2;
for (b=1; b<nbound; b++)
  ct[b][OH_LOWER][OH_CTYPE_FROM] =
    ct[b][OH_LOWER][OH_CTYPE_TO]   =
    ct[b][OH_LOWER][OH_CTYPE_SIZE] =
    ct[b][OH_UPPER][OH_CTYPE_FROM] =
    ct[b][OH_UPPER][OH_CTYPE_TO]   =
    ct[b][OH_UPPER][OH_CTYPE_SIZE] = 0;
```

Now we call `init3()` passing almost all arguments of `init4s()` but with the following exceptions, some of them are *different* from `init4p()`'s.

- `npmax`, `minmargin` and `maxdensity`, `cbufsize` and `zbound` are not passed because they are `init4s()`'s own.
- As in `init4p()`, `nphgram` is the pointer to `init4s()`'s local variable of the same name which has `NULL` to let `init3()` allocate `NOfPLocal[][][]`, because `init4s()` does not have the argument.
- As in `init4p()`, `rcounts` and `scounts` are `NULL` because they are unnecessary.
- *Unlike* `init4p()`, `pbuf` is the double pointer to `init4s()`'s local variable `pbufdummy` to avoid the particle buffer allocation in `init3()`. Another *difference* is that `maxlocalp` = 0 for `init3()` because this argument is meaningless.
- As in `init4p()`, `ftypes`, `cfields` and `ctypes` are `FieldTypes[][]`, `BoundaryCommFields[]` and `BoundaryCommTypes[][][][]` respectively, and the arrays themselves are neither allocated nor initialized by `init3()`.
- As in `init4p()`, `skip2` is 0 because we need level-2 initialization.

Note that, as in `init4p()`, `cfid` is passed unmodified.

```
init3(sdid, nspec, maxfrac, &nphgram, totalp, NULL, NULL, &pbufdummyptr,
      pbase, 0, mycommc, mycommf, nbor, pcoord, sdoms, scoord, nbound,
      bcond, bounds, (int*)ft, cf, cfid, (int*)BoundaryCommTypes, fsizes,
      stats, repiter, verbose, 0);
```

The next few parts are very *different* from `init4p()`'s. First the check of P'_{lim} = `nOfLocalPLimitShadow` is eliminated because the calculation of P'_{lim} is now done in `init4s()` as follows. As done in `oh4p_max_local_particles()`, we call `oh2_max_local_particles()` with `npmax`, `maxfrac` and `minmargin` to have the baseline of P'_{lim}, and then add $2(P_{halo}+P_{mgn})$ to it where P_{halo} and P_{mgn} are defined as follows as discussed in §4.12.3.

$$P_{halo} = \mathcal{D}((\delta_x^{\max}+2)(\delta_y^{\max}+2)(\delta_z^{\max}+2) - \delta_x^{\max}\delta_y^{\max}\delta_z^{\max})$$
$$P_{mgn} = \mathcal{D}\delta_x^{\max}\delta_y^{\max}$$

Then also as in `oh4p_max_local_particles()`, we examine P'_{lim} is not greater than the maximum positive `int`-type number `INT_MAX` $= 2^{31} - 1$ to abort execution by `mem_alloc_error()` unless it holds, and store P'_{lim} into the simulator body's variable pointed by `maxlocalp` and also into `nOfLocalPLimitShadow` for the consistency check in `oh4s_particle_buffer()`.

We also calculate K by;

$$K = 2\mathcal{D}\delta_z^{\max}((\delta_x^{\max} + 2e^g)(\delta_y^{\max} + 2e^g) - \delta_x^{\max}\delta_y^{\max})$$

to allocate `BoundarySendBuf[]` of K particles by `mem_alloc()`. On the other hand, the size P_{comm} of `sbuf[]` and `rbuf[]` arguments of `oh4s_exchange_border_data()`, being reported through the argument `cbufsize`, can be smaller because these buffers does not need to have four vertical halo planes but just to have a pair of them. Therefore, P_{comm} is defined as;

$$P_{comm} = 2\mathcal{D}\delta_z^{\max} \max(\delta_x^{\max} + 2e^g,\ \delta_y^{\max})$$

taking it into account that a xz-plane should include exterior pillars while a yz-plane may exclude them.

```
size =
  ((Grid[OH_DIM_X].size+ext2)*(Grid[OH_DIM_Y].size+ext2)*
   (Grid[OH_DIM_Z].size+ext2)-
   Grid[OH_DIM_X].size*Grid[OH_DIM_Y].size*Grid[OH_DIM_Z].size) +
  Grid[OH_DIM_X].size*Grid[OH_DIM_Y].size;
npl = (dint)oh2_max_local_particles(npmax, maxfrac, minmargin) +
  2 * maxdensity * size;
if (npl>INT_MAX) mem_alloc_error("Particles", 0);
nOfLocalPLimitShadow = *maxlocalp = npl;
size =
  2 * maxdensity * Grid[OH_DIM_Z].size *
  ((Grid[OH_DIM_X].size+ext2)*(Grid[OH_DIM_Y].size+ext2)-
   Grid[OH_DIM_X].size*Grid[OH_DIM_Y].size);
BoundarySendBuf =
  (struct S_particle*)mem_alloc(sizeof(struct S_particle), size,
                                "BoundarySendBuf");
size = Grid[OH_DIM_X].size + ext2;
if (size<Grid[OH_DIM_Y].size)  size = Grid[OH_DIM_Y].size;
*cbufsize = 2 * maxdensity * Grid[OH_DIM_Z].size * size;
```

The next part is alomost equivalent to `init4p()`'s. That is, we initialize `PbufIndex` to be `NULL` to avoid the reference to it in `Check_Particle_Location()` before the first call of `transbound4s()`; allocate `NOfPGrid`[2][S][G] and `NOfPGridTotal`[2][S][G] by `Allocate_NOfPGrid()` after setting `GridDesc`[0] for the local node's primary subdomain by `set_grid_descriptor()`; calculate Γ to examine if $gidx(\delta_x^{\max}-1, \delta_y^{\max}-1, \delta_z^{\max}-1)$ is small enough to represent it by `int` when combined with the largest possible subdomain code; let `logGrid` and `gridMask` have Γ and $2^\Gamma - 1$ respectively; and call `adjust_field_descriptor()` to modify `FieldDesc`[$F-1$].{`bc`, `red`}`.size`[0].

A *difference* is that we also allocate level-4s's own `NOfPGridZ`[$\delta_z^{\max}$] by `mem_alloc()`. The other *difference* is in the abortion by `errstop()` with too large Γ with `OH_nid_t` being `int`. That is, codes for $D < 3$ cases are eliminated and the local grid size shown in the error message reflects the fact that the planes in exterior are $3e^g$ thick.

```
me = myRank;
```

```
PbufIndex = NULL;
set_grid_descriptor(0, me);
size = GridDesc[0].dw * GridDesc[0].h;
Allocate_NOfPGrid(npgdummy, NOfPGrid, dint, size, "NOfPGrid");
Allocate_NOfPGrid(npgtdummy, NOfPGridTotal, dint, size, "NOfPGridTotal");
NOfPGridZ = (dint*)mem_alloc(sizeof(dint), Grid[OH_DIM_Z].size,
                             "NOfPGridZ");

size = Coord_To_Index(Grid[OH_DIM_X].size-1,
                      If_Dim(OH_DIM_Y, Grid[OH_DIM_Y].size-1, 0),
                      If_Dim(OH_DIM_Z, Grid[OH_DIM_Z].size-1, 0),
                      GridDesc[0].w, GridDesc[0].dw);
for (loggrid=0; size; loggrid++,size>>=1);
idmax = (dint)(((nn+OH_NEIGHBORS)<<1)-1)<<loggrid;
if (idmax>INT_MAX && sizeof(OH_nid_t)==sizeof(int)) {
  const int ext6 = ext3<<1;
  errstop("local grid size (%d+%d)*(%d+%d)*(%d+%d) times number of nodes %d "
          "is too large for OH_nid_t=int and thus OH_BIG_SPACE should be "
          "defined.",
          GridDesc[0].w-ext6, ext6,
          GridDesc[0].d-ext6, ext6,
          GridDesc[0].h-ext6, ext6, nn);
}
logGrid = loggrid;  gridMask = (1 << loggrid) - 1;
adjust_field_descriptor(0);
```

The next targets of allocation and initialization are data structures of level-4s's own, i.e., `HPlane`[2][2], `VPlane`[$2N+6$], `ZBoundShadow`[$\delta_z^{\max}$] and `InteriorParts`[2][S]. First, we allocate $2 \times 2 \times 4 \times S$ elements of `int` for four arrays of S integers in each element of `HPlane`[2][2]. Then the pointer for each of arrays `nsend`, `nrecv`, `sbuf` and `rbuf` in each element of `HPlane`[][] is let point appropriate portion of S integers, while `nbor` is initialized to be `MPI_PROC_NULL` to indicate we have no transfer schedules for particle-associated array data in horizontal halo planes so that `oh4s_exchange_border_data()` will do nothing even if it is called before the first call of `oh4s_transbound()`.

Next we allocate $2N + 6$ elements of `S_vplane` structure for `VPlane`[], and let `VPlaneHead`[d][p][β] $= 0$ for all $d \in \{0, 1\}$, $p \in \{0, 1\}$ and $\beta \in \{0, 1\}$ as well as its last element [2][0][0] to mean we have no transfer schedules for particle-associated array data in vertical halo planes so that `oh4s_exchange_border_data()` will do nothing even if it is called before the first call of `oh4s_transbound()` again.

Next, we allocate `*zbound`[2][2] by `mem_alloc()` if `zbound` argument points `NULL`, and let `ZBoundShadow` point what `*zbound` points anyway. Then we initialize its elements as `ZBoundShadow`[0][] $= \{0, \delta_z(n)\}$ for the local node n to mean its primary subcuboid is the primary subdomain itself, and `ZBoundShadow`[1][] $= \{0, 0\}$ to mean n does not have secondary subcuboid, to allow the simulator body refers to the array before the first call of `oh4s_transbound()`. Note that the substance `ZBound`[][] is not allocated because it has its body on the declaration, and its elements are not initialized because they will not be accessed before the first call of `oh4s_transbound()`.

Next and at last of this part, we allocate `InteriorParts`[2][S] by `mem_alloc()`.

Note that we eliminate the allocation and initalization of data structures for hot-spots of level-4p, of course.

```
iptr = (int*)mem_alloc(sizeof(int), 2*2*4*nspec, "HPlane");
```

```
for (ps=0; ps<2; ps++)  for (i=OH_LOWER; i<=OH_UPPER; i++) {
  HPlane[ps][i].nsend = iptr;  iptr += nspec;
  HPlane[ps][i].nrecv = iptr;  iptr += nspec;
  HPlane[ps][i].sbuf  = iptr;  iptr += nspec;
  HPlane[ps][i].rbuf  = iptr;  iptr += nspec;
  HPlane[ps][i].nbor  = MPI_PROC_NULL;
}
size = 2*nn + 2*2 + 2;
VPlane = (struct S_vplane*)mem_alloc(sizeof(struct S_vplane), size,
                                     "VPlane");
VPlaneHead[0] = VPlaneHead[1] = VPlaneHead[2] = VPlaneHead[3] =
  VPlaneHead[4] = VPlaneHead[5] = VPlaneHead[6] = VPlaneHead[7] =
  VPlaneHead[8] = 0;

iptr = *zbound;
if (!iptr)  iptr = *zbound = mem_alloc(sizeof(int), 4, "ZBound");
ZBoundShadow = (int(*)[2])iptr;
ZBoundShadow[0][OH_LOWER] = 0;  ZBoundShadow[0][OH_UPPER] = GridDesc[0].z;
ZBoundShadow[1][OH_UPPER] = ZBoundShadow[1][OH_UPPER] = 0;

InteriorParts = mem_alloc(sizeof(struct S_interiorp), nspec*2,
                          "InteriorParts");
```

The next three lines are equivalent to those in `init4p()`; creating MPI data-type `T_Hgramhalf` by `MPI_Type_vector()` and `MPI_Type_commit()` for a slice $[p][*][m]$ in `NOfSend[][][]` and `NOfRecv[][][]`; and zero-clearing of all elements in `NOfSend[][][]` for the first call of `oh4s_transbound()`.

```
MPI_Type_vector(nspec, 1, nn, MPI_INT, &T_Hgramhalf);
MPI_Type_commit(&T_Hgramhalf);
for (n=0; n<nnns2; n++)  NOfSend[n] = 0;
```

The next allocation and initialization for data structures of neighboring information is very *different* from `init4p()`'s. That is, unlike level-4p nor other lower levels, `Neighbors`$[0][] =$ `DstNeighbors`$[]$ and `SrcNeighbors`$[]$ must be aware of system boundary conditions. That is, in level-4p and lower levels, a subdomain may have its neighbor beyond a non-periodic system boundary as if it is periodic, because no particle transfers eventually take place through the boundary. However in level-4p, the subdomain cannot have k-th neighbor if a system boundary separating them is non-periodic and thus it must be `Neighbors`$[k] = -(N+1)$, or a node responsible of the subdomain would try to send particles in its interior halo plane to the neighbor, while the neighbor correctly knows it has no such particles in its exterior halo plane because `exchange_population()` is aware of the system boundary condition.

Therefore we temporarily rebuild `DstNeighbors`$[k]$ and `SrcNeighbors`$[k'=3^D-k-1]$ for the local node n as follows, where n_k is the value of `DstNeighbors`$[k]$ given by `init1()`, $k = \sum_{d=0}^{D-1} \nu_d 3^d$, and $b_d^\beta =$ `Boundaries`$[n][d][\beta]$.

$$\mathit{periodic}(k) = \bigwedge_{d=0}^{D-1} (\nu_d = 1 \ \vee \ b_d^{\nu_d/2} = 0)$$

$$\texttt{DstNeighbors}[k] = \texttt{SrcNeighbors}[k'] = \begin{cases} n_k & \mathit{periodic}(k) \ \wedge \ n_k \geq 0 \\ -(n_k+1) & \mathit{periodic}(k) \ \wedge \ -N \leq n_k < 0 \\ -(N+1) & \neg \mathit{periodic}(k) \ \vee \ n_k = -(N+1) \end{cases}$$

That is, we let `DstNeighbors`$[k]$ = `SrcNeighbors`$[k']$ have non-negative subdomain identifier of the k-th neighbor if it originally exists and is not beyond non-periodic system boundary, or let them have $-(N+1)$ otherwise.

Then we modify `DstNeighbors`$[k]$ and `SrcNeighbors`$[k]$ so that they have the followings, where Δ_k and Σ_k are their original values, as done in `init1()` using `TempArray`$[m]$ ($m \in [0, N)$) for the occurence check.

$$\texttt{DstNeighbors}[k] = \begin{cases} -(N+1) & \Delta_k < 0 \\ \Delta_k & \Delta_k \geq 0 \ \wedge \ \forall l < k : (\Delta_l \neq \Delta_k) \\ -(\Delta_k + 1) & \Delta_k \geq 0 \ \wedge \ \exists l < k : (\Delta_l = \Delta_k) \end{cases}$$

$$\texttt{SrcNeighbors}[k] = \begin{cases} -(N+1) & \Sigma_k < 0 \\ \Sigma_k & \Sigma_k \geq 0 \ \wedge \ \forall l < k : (\Sigma_l \neq \Sigma_k) \\ -(\Sigma_k + 1) & \Sigma_k \geq 0 \ \wedge \ \exists l < k : (\Sigma_l = \Sigma_k) \end{cases}$$

At the same time, we let `FirstNeighbor`$[k]$ as follows as done in `init4p()` but using `TempArray`$[N+m]$ ($m \in [0, N)$) to remember minimum k such that $m = \Sigma_k$.

$$\texttt{FirstNeighbor}[k] = \begin{cases} k & \Sigma_k < 0 \\ \min\{l \,|\, l \leq k, \ \Sigma_l = \Sigma_k\} & \Sigma_k \geq 0 \end{cases}$$

In the loop doing above, we also add a very *level-4s own* initialization to let `PrimaryRLIndex`$[k] = k$ which are referred to by `make_send_sched()` and its callees when the next execution mode is primary as the trivial index of primary receiving, primary sending, secondary receiving and secondary sending blocks of `S_commlist` records, while the blocks are set in `update_neighbors()`.

The remaining part is literally equivalent to `init4p()`'s, but the call of `update_neighbors()` is semantically *different* because its initialization based on `Neighbors`[0][] is not only for `AbsNeighbors`[0][] and `GridOffset`[0][] but also for `PrimaryCommList`[0][]. On the other hand, other procedures, the allocation of `RealDstNeighbors`[2][2]`.nbor`$[N]$ and `RealSrcNeighbors`[2][2]`.nbor`$[N]$ by `mem_alloc()` and the call of `update_real_neighbors()` with the code `URN_PRI` to initialize their elements in [0][0] so that they have subdomain identifiers neighboring to the local node's primary subdomain, are perfectly equivalent to `init4p()`'s.

```
for (z=0,n=0; z<3; z++) {
  int (*bd)[OH_DIMENSION][2] = Boundaries;
  const int nonpz = z!=1 && bd[me][OH_DIM_Z][z>>1];
  for (y=0; y<3; y++) {
    const int nonpy = nonpz || (y!=1 && bd[me][OH_DIM_Y][y>>1]);
    for (x=0; x<3; x++,n++) {
      int dnbr = DstNeighbors[n];
      const int nrev = OH_NEIGHBORS - 1 - n;
      if (nonpy || (x!=1 && bd[me][OH_DIM_X][x>>1]))
        DstNeighbors[n] = SrcNeighbors[nrev] = -(nn+1);
      else if (dnbr<0 && dnbr>=-nn)
        DstNeighbors[n] = SrcNeighbors[nrev] = -(dnbr+1);
      else
        SrcNeighbors[nrev] = dnbr;
    }
  }
}
for (i=0; i<nn; i++)  TempArray[i] = 0;
```

```
  for (n=0; n<OH_NEIGHBORS; n++) {
    const int dnbr = DstNeighbors[n],  snbr = SrcNeighbors[n];
    int *sfirst = TempArray + nn;
    if (dnbr>=0) {
      if (TempArray[dnbr]&1)  DstNeighbors[n] = -(dnbr+1);
      else                    TempArray[dnbr] |= 1;
    }
    if (snbr>=0) {
      if (TempArray[snbr]&2) {
        SrcNeighbors[n] = -(snbr+1);
        FirstNeighbor[n] = sfirst[snbr];
      } else {
        FirstNeighbor[n] = sfirst[snbr] = n;
        TempArray[snbr] |= 2;
      }
    } else
      FirstNeighbor[n] = n;
    PrimaryRLIndex[n] = n;
  }
  update_neighbors(0);
  rnbr = (int*)mem_alloc(sizeof(int), nn*2*2*2, "RealNeighbors");
  for (tr=0; tr<2; tr++)  for (ps=0; ps<2; ps++,rnbr+=nn) {
    RealDstNeighbors[tr][ps].n = RealSrcNeighbors[tr][ps].n = 0;
    RealDstNeighbors[tr][ps].nbor = rnbr;
    RealSrcNeighbors[tr][ps].nbor = rnbr + nn*2*2;
  }
  update_real_neighbors(URN_PRI, 0, -1, -1);
```

The last part, in which we copy the contents of `bcond`$[D][2]$ to its substance `BoundaryCondition`$[][]$ by `memcpy()` if `SubDomainDesc` is `NULL` to mean regular process coordinate, is also equivalent to `init4p()`'s.

```
    if (!SubDomainDesc)
      memcpy(BoundaryCondition, bcond, sizeof(int)*OH_DIMENSION*2);
  }
```

4.13.7 `oh4s_particle_buffer()`

`oh4s_particle_buffer_()` `oh4s_particle_buffer()`

The API functions `oh4s_particle_buffer_()` for Fortran and `oh4s_particle_buffer()` for C associate particle buffer `pbuf`$[2P_{lim}]$ to `Particles`$[P_{lim}]$ and `SendBuf`$[P_{lim}]$ where P_{lim} = `maxlocalp`. The function is level-4s's own but performs what `init4p()` and `init2()` do as follows.

First, we confirm that P'_{lim} = `nOfLocalPLimitShadow` is non-negative and not greater than P_{lim} = `maxlocalp`, or in other words `oh4s_init()` has been called and its argument `maxlocalp` is passed (possibly after incremented) to `maxlocalp` of this function. If not, we stop the execution by `errstop()` with an appropriate error message. This part is very similar to the corresponding part of `init4p()` but the error messages are *different* reflecting the difference of the functions to calculate P'_{lim} and to check $P_{lim} > P'_{lim}$.

Next, as done in `init4p()`, we allocate `Particles`$[P_{lim}]$ and `SendBuf`$[P_{lim}]$ as a contiguous array of $[2P_{lim}]$ if `pbuf` points `NULL`. Otherwise, what `pbuf` points is set to the pointer `Particles`, and `SendBuf` is let point the head of the second half of `pbuf`.

Finally, we let `nOfLocalPLimit` $= P_{lim}$ and `totalParts` $= P_{lim}$ as done in `init2()`, because the assignments in `init2()` are meaningless with incorrect `maxlocalp` passed to it by `init4s()` through `init3()`.

```
void
oh4s_particle_buffer_(const int *maxlocalp, struct S_particle *pbuf) {
  oh4s_particle_buffer(*maxlocalp, &pbuf);
}
void
oh4s_particle_buffer(const int maxlocalp, struct S_particle **pbuf) {

  if (nOfLocalPLimitShadow<0)
    errstop("oh4s_particle_buffer() has to be called after oh4s_init()");
  else if (maxlocalp<nOfLocalPLimitShadow)
    errstop("argument maxlocalp %d given to oh4s_particle_buffer() is less "
            "than that calculated by oh4s_init() %d",
            maxlocalp, nOfLocalPLimitShadow);
  if (*pbuf)
    Particles = *pbuf;
  else
    Particles = *pbuf =
      (struct S_particle*)mem_alloc(sizeof(struct S_particle),
                                    maxlocalp<<1, "Particles");
  SendBuf = Particles + maxlocalp;
  nOfLocalPLimit = totalParts = maxlocalp;
}
```

4.13.8 `oh4s_per_grid_histogram()`

`oh4s_per_grid_histogram_()` `oh4s_per_grid_histogram()`

The API functions `oh4p_per_grid_histogram_()` for Fortran and `oh4p_per_grid_histogram()` for C associate the shadow per-grid histogram `NOfPGridOutShadow`[][][] and per-grid index `NOfPGridIndexShadow`[][][] to those in the simulator body given through the arguments `pghgram` and `pgindex`. The *differnces* between the functions and their counterparts `oh4p_per_grid_histogram[_]()` described in §4.10.8 are as follows; level-4s's has an additional argument `pgindex` for per-grid index; level-4s's associates arguments with shadow arrays because substance ones are accessed outside `oh4s_transbound()`, i.e., in `oh4s_exchange_border_data()`; and thus level-4s's allocates substance arrays.

As in the countepart of level-4p, the Fortran coded simulator must allocate 5-dimensional arrays whose leading 3-dimensional sizes are commonly specified in `fsizes`$[F-1]$[][][] given through the argument of `oh4s_init_()`, and give the origin element of the array $(0, 0, 0, 1, 1)$ through `pghgram` and `pgindex`. On the other hand, C coded simulator may let the function allocate the arrays by giving a double pointers to `NULL` to the arguments, or allocate the arrays by itself and give the double pointer to the array's origin element to arguments.

The function invokes the macro `Allocate_NOfPGrid()` to allocate the shadow arrays of $[2][S][G]$ where;

$$G = \texttt{GridDesc}[0]\texttt{.dw} \times \texttt{GridDesc}[0]\texttt{.h} = ((\delta_x^{\max} + 6e^g)(\delta_y^{\max} + 6e^g)) \times (\delta_z^{\max} + 6e^g)$$

and *returns* the pointer to the conceptual element [0][0][0][0][0] through `*pghgram` and `*pgindex` if they are `NULL`, to allocate the pointer array for them for the use in library functions, and to zero-clear their bodies. The macro is also used for the allocation and initialization of substance arrays `NOfPGridOut`$[2][S][G]$ and `NOfPGridIndex`$[2][S][G]$.

```
void
oh4s_per_grid_histogram_(int *pghgram, int *pgindex) {
  oh4s_per_grid_histogram(&pghgram, &pgindex);
}
void
oh4s_per_grid_histogram(int **pghgram, int **pgindex) {
  int *npgo=NULL, *npgi=NULL;
  const int size = GridDesc[0].dw*GridDesc[0].h;
  Allocate_NOfPGrid(npgo, NOfPGridOut, int, size, "NOfPGridOut");
  Allocate_NOfPGrid(*pghgram, NOfPGridOutShadow, int, size,
                    "NOfPGridOutShadow");
  Allocate_NOfPGrid(npgi, NOfPGridIndex, int, size, "NOfPGridIndex");
  Allocate_NOfPGrid(*pgindex, NOfPGridIndexShadow, int, size,
                    "NOfPGridIndexShadow");
}
```

4.13.9 `oh4s_transbound()` and `transbound4s()`

`oh4s_transbound_()` `oh4s_transbound()`

The API function `oh4s_transbound_()` for Fortran and `oh4s_transbound()` for C provide the simulator body calling them with the load-balanced particle transfer mechanism of level-4s and lower level libraries. The meanings of their two arguments, `currmode` and `stats`, and return value in $\{-1, 0, 1\}$ are perfectly equivalent to those of the level-1 to level-3 counterparts `oh1_transbound[_]()`, `oh2_transbound[_]()` and `oh3_transbound[_]()`. Also similarly to the counterparts, their bodies only have a simple call of `transbound4s()` but the third argument `level` is 4 to indicate the function is called from level-4s API functions.

```
int
oh4s_transbound_(int *currmode, int *stats) {
  return(transbound4s(*currmode, *stats, 4));
}
int
oh4s_transbound(int currmode, int stats) {
  return(transbound4s(currmode, stats, 4));
}
```

`transbound4s()`

The function `transbound4s()`, called from `oh4s_transbound[_]()`, is very similar to its level-4p counterpart `transbound4p()` described in §4.10.9. The *difference* between them are as follows; level-4s's initializes `ZBound[][]` at its beginning and copies the array into its shadow `ZBoundShadow[][]` at its end; and the range of `NOfPGrid[][][]` to be zero-cleared at the end of level-4s's is larger reflecting the fatter exterior.

Equivalently to `transbound4p()`, at first we call `transbound1()` to calculate `NOfPrimaries[][]`, `TotalPGlobal[]`, `nOfParticles` and `nOfLocalPMax` and to have `currmode` always, and to calculate `TotalP[][]`, `primaryParts` and `totalParts` on the first call. Then we perform level-4s's own operations to let `ZBound`$[p][\beta] = 0$ for all $p \in \{0, 1\}$ and $\beta \in \{0, 1\}$ to mean the local node has neither of primary nor secondary subcuboids, i.e., no particles at all, unless `exchange_particles4s()` finds some particles assigned to the node very likely but not necessarily. Then we call functions for the heart of balancing examination similarly to `transbound4p()` but there are *difference* that called functions are level-4s's own `try_primary4s()`, `try_stable4s()` and `rebalance4s()`.

```
static int
transbound4s(int currmode, int stats, const int level) {
  int ret=MODE_NORM_SEC;
  const int nn=nOfNodes, ns=nOfSpecies, ns2=ns<<1, nnns2=nn*ns2;
  struct S_particle *tmp;
  int i, ps, s, tp;
  Decl_For_All_Grid();

  stats = stats && statsMode;
  currmode = transbound1(currmode, stats, level);

  ZBound[0][OH_LOWER] = ZBound[0][OH_UPPER] = 0;
  ZBound[1][OH_UPPER] = ZBound[1][OH_UPPER] = 0;
  if (try_primary4s(currmode, level, stats))  ret = MODE_NORM_PRI;
  else if (!Mode_PS(currmode) || !try_stable4s(currmode, level, stats)) {
    rebalance4s(currmode, level, stats);  ret = MODE_REB_SEC;
  }
```

The next part is equivalent to `transbound4p()`'s. We allocate `PbufIndex`$[2][S]$ and its additional element $[2][0]$ by `mem_alloc()` if it is `NULL` to mean the first call of this function. Then we clear `NOfPLocal`$[][][]$; copy `TotalPNext`$[p][s]$ to its substance `TotalP`$[p][s]$ letting `PbufIndex`$[p][s]$ have the index of $pbuf(p,s)$; set `totalParts`, its shadow pointed by `totalLocalParticles` and `PbufIndex`$[2][0]$ to the sum of `TotalP`$[p][s]$ for all $p \in [0,1]$ and $s \in [0,S)$; and clear `InjectedParticles`$[0][][] = q^{\mathrm{inj}}(n)[][]$ and `nOfInjections` $= Q_n^{\mathrm{inj}}$.

```
  if (!PbufIndex)
    PbufIndex = (int*)mem_alloc(sizeof(int), ns2+1, "PbufIndex");
  for (i=0; i<nnns2; i++) NOfPLocal[i] = 0;
  for (s=0,tp=0; s<ns2; s++) {
    TotalP[s] = TotalPNext[s];  PbufIndex[s] = tp;  tp += TotalPNext[s];
  }
  PbufIndex[s] = totalParts = *totalLocalParticles = tp;  nOfInjections = 0;
  for (s=0; s<ns2; s++)  InjectedParticles[s] = 0;
```

The next part is almost equivalent to `transbound4p()`'s. It zero-clears elements of `NOfPGrid`$[p][s][gidx(x,y,z)]$ for $p = 0$ if the next execution mode is primary or $p \in [0,1]$ if secondary, for all $s \in [0,S)$ and all $(x,y,z) \in [-ke^g, \delta_x(m) + ke^g) \times [-ke^g, \delta_y(m) + ke^g) \times [-ke^g, \delta_z(m) + ke^g)$ where $m = n$ for $p = 0$ or $m = parent(n)$ for $p = 1$ for the local node n, and $k = 1$ for $p = 0$ or the helper-helpand tree is kept, or $k = 3$ being *different* from $k = 2$ in `transbound4p()` because of $3e^g$ thickness of the exterior otherwise, by `For_All_Grid()`.

```
  for (ps=0; ps<=Mode_PS(ret); ps++) {
    const int extio = (ps==1 && ret<0) ? OH_PGRID_EXT*3 : OH_PGRID_EXT;
    for (s=0; s<ns; s++) {
      dint *npg = NOfPGrid[ps][s];
      For_All_Grid(ps, -extio, -extio, -extio, extio, extio, extio)
        npg[The_Grid()] = 0;
    }
  }
```

Then we copy `ZBound`$[][]$ to its shadow `ZBoundShadow`$[][]$ to let it be referred to by the simulator body, as another level-4s's own operation. Finally and equivalently to

transbound4p(), we exchange the role of Particles[] and SendBuf[], and return to the caller with the return value defined in §4.3.10 letting currMode have its absolute value in order to replaced MODE_REB_SEC = −1 with MODE_NORM_SEC = 1.

```
    ZBoundShadow[0][0] = ZBound[0][0];    ZBoundShadow[0][1] = ZBound[0][1];
    ZBoundShadow[1][0] = ZBound[1][0];    ZBoundShadow[1][1] = ZBound[1][1];
    tmp = Particles;  Particles = SendBuf;  SendBuf = tmp;
    currMode = ret<0 ? -ret : ret;
    return(ret);
  }
```

4.13.10 try_primary4s()

try_primary4s()

The function try_primary4s(), called solely from transbound4s(), examines if we can stay in or turn to primary mode. If so, the local node gathers all the particles in its primary subdomain from other nodes, sort them according to their grid-position, and then gather halo particles from its neighbor nodes. The function has three arguments currmode, level and stats whose meanings are perfectly equivalent to those of its level-1 counterpart try_primary1().

The code structure of this function is completely *different* from its level-4p counterpart try_primary4p() described in §4.10.10, because it shares the particle transfer and sorting mechanisms implemented in exchange_particles4s() with try_stable4s() and rebalance4s(), while try_primary4p() has its own mechanisms for primary mode. The reason why we made the mechanisms common is that we need to have halo particle transfer and its scheduling which are easily implemented with S_commlist records even for primary mode, i.e., those in PrimaryCommList[].

In this function, first we call the level-1 counterpart try_primary1() to examine if the next execution mode is primary. If not, we simply return to its caller transbound4s() with the return value FALSE to indicate the mode will be secondary.

Otherwise, i.e., if we will be in primary mode, we call exchange_particles4s() with the arguments currmode, level and stats of this functions's own, and nextmode = 0 meaning the next execution mode is primary, reb = 0 meaning no rebalancing took place, oldp = *parent*(n) for the helpand of the local node in the last step being RegionId[1] before the call of try_primary1() if any, and newp = −1 meaning the local node will not have helpand of course.

After that, if we were in secondary mode, we call update_real_neighbors() with the operation code URN_PRI to reinitialize the elements RealDstNeighbors[0][0] and RealSrcNeighbors[0][0] so that they have subdomain identifiers neighboring to the local node's primary subdomains. Note that this call should be done *after* the call of exchange_particles4s() because the elements has been kept to send n's secondary particles to the nodes whose primary subdomains are neighbors of n's secondary subdomain, and to receive n's primary particles in the next step from the nodes whose primary or secondary subdomains are neighbors of n's primary subdomain.

Finally the function returns to transbound4s() with the return value of TRUE.

```
static int
try_primary4s(const int currmode, const int level, const int stats) {
  const int oldp = RegionId[1];

  if (!try_primary1(currmode, level, stats)) return(FALSE);
```

```
  exchange_particles4s(currmode, 0, level, 0, oldp, -1, stats);
  if (Mode_PS(currmode))  update_real_neighbors(URN_PRI, 0, -1, -1);
  return(TRUE);
}
```

4.13.11 `try_stable4s()`

`try_stable4s()` The function `try_stable4s()`, solely called from `transbound4s()`, examines if the current helpand-helper configuration sustains by `try_stable1()` and, if examination passes, performs particle transfer by `exchange_particles4s()`.

The code structure of this function is very similar to that of its level-4p counterpart `try_stable4p()` described in §4.10.11. That is, first we call the level-1 counterpart `try_stable1()` passing all arguments or negating `level` argument according to the accommodation pattern is anywhere or normal respectively. Then if `try_stable1()` returns `FALSE` we return to the caller `transbound4s()` with `FALSE` too, or we call `exchange_particles4s()` for particle transfer otherwise. The *differnce* is in the latter case because `exchange_particles4s()` is different from its counterpart `exchange_particles4p()` and it has an additional argument `nextmode` being 1 for this call to mean we will be in secondary mode in the next step. The other arguments, however, are same as those in `try_stable4p()`, and thus `reb` $= 0$ meaning no rebalancing took place and `oldp` $=$ `newp` $=$ `RegionId`$[1] = parent(n)$ meaning the helpand of the local node n is unchanged. After the call, as in `try_stable4p()`, we return to `transbound4s()` with `TRUE`.

```
static int
try_stable4s(const int currmode, const int level, const int stats) {
  if (!try_stable1(currmode, (Mode_Acc(currmode) ? level : -level), stats))
    return(FALSE);
  exchange_particles4s(currmode, 1, level, 0, RegionId[1], RegionId[1], stats);
  return(TRUE);
}
```

4.13.12 `rebalance4s()`

`rebalance4s()` The function `rebalance4s()`, solely called from `transbound4s()`, builds the new family tree to rebalance the load among nodes by `rebalance1()`, and then performs particle transfer by `exchange_particles4s()`. The code structure of this function is very similar to that of its level-4p counterpart `rebalance4p()` described in §4.10.12. That is, first we call the level-1 counterpart `rebalance1()` passing all arguments or negating `level` argument according to the accommodation pattern is anywhere or normal respectively. Then, if n^p_{old} and n^p_{new}, being the $parent(n)$ of the local node n in the last and next step respectiely, are different and we have anywhere accommodation, we modify `InjectedParticles`$[0][1][s]$ for all $s \in [0, S)$ so that it has the number of secondary particles injected to the new secondary subdomain n^p_{new} accidentally.

Then we call `exchange_particles4s()` *differently* from that in `rebalance4p()` because the function is different from its counterpart `exchange_particles4p()` and it has an additional argument `nextmode` being 1 for this call to mean we will be in secondary mode in the next step. The other arguments, however, are same as those in `rebalance4p()`, and thus `reb` $= 1$ meaning rebalancing took place, `oldp` $= n^p_{old}$, and `newp` $= n^p_{new}$.

Then after the call, as in `rebalance4p()`, we do the followings if we had normal accommodation; call `set_grid_descriptor()` to update `GridDesc`[1][] for the secondary subdomain; move `Neighbors`[2][k] for the neighbors of n^p_{new} to `Neighbors`[1][k] for all $k \in [0, 3^D)$; and finally call `update_neighbors()` telling it to update elements in `AbsNeighbors`[1][] and `GridOffset`[1][] for the secondary subdomain.

```
static void
rebalance4s(const int currmode, const int level, const int stats) {
  const int me=myRank, ns=nOfSpecies;
  const int oldp = RegionId[1],  amode = Mode_Acc(currmode);
  const int ninj = nOfInjections;
  int s, n, newp;

  rebalance1(currmode, (amode ? level : -level), stats);
  newp = amode ? Nodes[me].parentid : NodesNext[me].parentid;
  if (ninj && amode && oldp!=newp) {
    int *sinj = InjectedParticles + ns;
    const int sbase=specBase;
    int i;
    struct S_particle *p;
    Decl_Grid_Info();
    for (s=0; s<ns; s++)  sinj[s] = 0;
    if (newp>=0) {
      for (i=0,p=Particles+totalParts; i<ninj; i++,p++) {
        const OH_nid_t nid = p->nid;
        int sdid;
        if (Secondary_Injected(nid)) {
          Primarize_Id(p, sdid);  Secondarize_Id(p);
          if (sdid==newp)  sinj[Particle_Spec(p->spec-sbase)]++;
        }
      }
    }
  }
  exchange_particles4s(currmode, 1, level, 1, oldp, newp, stats);
  if (!amode) {
    set_grid_descriptor(1, newp);
    for (n=0; n<OH_NEIGHBORS; n++)  Neighbors[1][n] = Neighbors[2][n];
    update_neighbors(1);
  }
}
```

4.13.13 Macros `Parent_Old()`, `Parent_New()`, `Parent_New_Same()` and `Parent_New_Diff()`

`Parent_Old()`
`Parent_New()`
`Parent_New_Same()`
`Parent_New_Diff()`

The following four macros to examine the statuses of old and new parents n^p_{old} and n^p_{new} and its encoding in the local variable `pcode` $= \pi$ of `exchange_particles4s()` is perfectly equivalent to those in level-4p.

- `Parenet_Old`(π) is true iff $n^p_{old} \geq 0$.
- `Parent_New`(π) is true iff $n^p_{new} \geq 0$.
- `Parent_New_Same`(π) is true iff $n^p_{new} \geq 0$ and $n^p_{new} = n^p_{old}$.

- `Parent_New_Diff`(π) is true iff $n^p_{new} \geq 0$ and $n^p_{new} \neq n^p_{old}$.

These macros `Parent_Old()` and `Parent_New()` are used in `exchange_particles4s()` and `make_send_sched()`, while `Parent_New_Diff()` is solely used in `make_send_sched()`[81].

```
#define Parent_Old(PCODE)       ((PCODE) & 4)
#define Parent_New(PCODE)       ((PCODE) & 2)
#define Parent_New_Same(PCODE)  (((PCODE) & 3) == 3)
#define Parent_New_Diff(PCODE)  (((PCODE) & 3) == 2)
```

4.13.14 `exchange_particles4s()`

`exchange_particles4s()` The function `exchange_particles4s()`, called from `try_primary4s()`, `try_stable4s()` and `rebalance4s()`, performs an all-to-all type position-aware particle transfer including that for halo particles. Though the function has some similarity to its level-4p counterpart `exchange_particles4p()` described in §4.10.14, it has various aspects *different* from the counterpart as follows.

- This function is called not only from `try_stable4s()` and `rebalance4s()` but also from `try_primary4s()`, because halo particle transfer can be implemented easily by letting this function cover both primary and secondary modes in the next step, rather than having a mechanism specific to primary mode which `try_primary4p()` has. Therefore the function has an argument `nextmode` $= p'_n$ additional to those of `exchange_particles4p()` to indicate we will be in primary ($p'_n = 0$) or secondary mode in the next step. This difference also makes it unnecessary to have level-4s counterparts of level-4p functions `mpi_allreduce_wrapper()` and `move_and_sort_primary()`, while making the following callee functions responsible of primary mode too; `make_send_sched()`, `exchange_xfer_amount()`, `move_to_sendbuf_4s()`, `move_and_sort()`, and `xfer_particles()`.
- Since we have no hot-spots in level-4s, this function and its callees `make_recv_list()`, `make_send_sched()`, `move_to_sendbuf_4s()` and `move_and_sort()` are free from hot-spot-related operations. This also makes it unnecessary to have level-4s counterparts of level-4p functions `gather_hspot_recv()`, `gather_hspot_send()`, `gather_hspot_send_body()`, `scatter_hspot_send()`, `scatter_hspot_recv()` and `scatter_hspot_recv_body()`.
- Since a node is responsible of particles in subcuboids rather than a general contiguos set of grid-voxels, `make_recv_list()` and `make_send_sched()` works xy-plane-wise.
- Since we have to transfer halo particles, this function has calls of `make_bxfer_sched()` for scheduling, and `xfer_boundary_particles_v()` and `xfer_boundary_particles_h()` for transfer. This also lets the following functions take care the halo particle transfer; `make_send_sched()`, `sort_particles()`, `move_and_sort()` and `sort_received_particles()`
- Since we have the shadow per-grid histogram `NOfPGridOutShadow[][][]` of `NOfPGridOut[][][]` and the substance/shadow pair of per-grid index arrays `NOfPGridIndex[][][]` and `NOfPGridIndexShadow[][][]`, this function is responsible to let these new arrays have

[81] `Parent_New_Same()` is not used at all in level-4s but we keep this macro to make level-4s is similar to level-4p as much as possible.

appropriate values, and also works on `NOfPGridTotal[][][]` to let it have per-grid index while `sort_particles()`, `move_and_sort_primary()` and `move_and_sort_secondary()` do it in level-4p.

The arguments except for the new one `nextmode` shown above are equivalent to `exchange_particles4p()`. That is, the meanings of `currmode`, `level` and `stats` are same as those of the callers, and `reb`, `oldp` $= n^p_{old}$, `newp` $= n^p_{new}$ for the local node n are as follows, where $parent(n)$ means n's helpand in the last step.

- `try_primary4s()` gives `reb` $= 0$, $n^p_{old} = parent(n)$ and $n^p_{new} = -1$.
- `try_stable4s()` gives `reb` $= 0$, $n^p_{old} = n^p_{new} = parent(n)$.
- `rebalance4s()` gives `reb` $= 1$, $n^p_{old} = parent(n)$ and n^p_{new} being the new helpand of n.

As in `exchange_particles4p()`, at first in the variable declaration part, the function determines whether we have to take care of the transitional state of helpand-helper configuration, i.e., whetehr we have normal accommodation and rebalancing took place, and let its local variable `trans` be true iff so. It also sets the parent status code discussed in §4.10.13 and §4.13.13 according to the arguments n^p_{old} and n^p_{new}.

```
static void
exchange_particles4s(int currmode, const int nextmode, const int level,
                     int reb, int oldp, int newp, const int stats) {
  const int ns=nOfSpecies, exti=OH_PGRID_EXT;
  const int trans = !Mode_Acc(currmode) && reb ? 1 : 0;
  int pcode =
    (oldp>=0 ? 4 : 0) + (newp>=0 ? 2 : 0) + (oldp==newp ? 1 : 0);
  int ps, psold, psnew, s;
  int nacc[2], nsend, tp;
  struct S_commlist *rlist[2];
  int *rlidx[2];
  Decl_For_All_Grid();
```

If we have anywhere accommodation and $p'_n = 1$, we do the followings almost equivalently to what we do in `exchange_particles4p()`. First we call `exchange_particles()` as we do in `try_stable2()` or `rebalance2()` with anywhere accommodation to have primary and secondary particles of the local node without position-aware manner. Then if rebalanced, we call the followings; `update_descriptors()` to update elements in `FieldDesc[]` for n^p_{new} and to reinitialize `BorderExc[][1][][]` for n^p_{old}; `set_grid_descriptor()` to update `GridDesc[1]` for n^p_{new}; `update_neighbors()` to update `AbsNeighbors[1][]` and `GridOffset[1][]`; and `update_real_neighbors()` with the operation code `URN_SEC` to update `RealDstNeighbors[0][`p`]` and `RealSrcNeighbors[0][`p`]` for $p \in [0, 1]$. Then we reinitialize `NOfSend[][][]` by zero-clearing its all elements because it has been updated by `make_comm_count()` called in `try_stable1()` or `rebalance1()`.

On the other hand, the anywhere accommodation case with $p'_n = 0$ is similar to the corresponding part in `try_primary4p()` and thus has the calls of `move_to_sendbuf_primary()` and `exchange_primary_particles()` for non-position-aware particle transfer.

Then still as in `exchange_particles4p()` for $p'_n = 1$ and `try_primary4p()` for $p'_n = 0$, we call `count_population()` to build the local per-grid histogram in `NOfPGrid[][][]`, and then let `reb` $= 0$ and $n^p_{old} = n^p_{new}$ with corresponding setting of `pcode` because we do

not have to take care the helpand-helper reconfiguration. We also let primary/secondary mode indicator in `currmode` be `nextmode`, but keeping the accommodation pattern in it to be anywhere[82], so that the following process in this function assumes that we were in the execution mode in which will be in, because all particles are now accommodated by nodes as the next execution mode requires regardless of the mode we were in and the accommodation pattern we had.

On the other hand, the local per-grid histogram is then used *differently* from level-4p. If $p'_n = 1$, we will call `exchange_population()` afterward rather than `reduce_population()` because we need not only to sum up the local per-grid histograms in the local node's primary family but also exchange populations in halo planes with neighbor nodes. If $p'_n = 0$ on the other hand, we cannot sort partilces by `sort_particles()` because we need to build halo particle transfer schedule before the sorting. Therefore, the operations specific to anywhere accommodation is over here.

```
if (Mode_Acc(currmode)) {
  if (nextmode) {
    int i;
    const int nnns2 = nOfNodes * nOfSpecies * 2;
    if (reb) {
      exchange_particles(SecRList, SecRLSize, oldp, 0, currmode, stats);
      update_descriptors(oldp, newp);
      set_grid_descriptor(1, newp);
      update_neighbors(1);
      update_real_neighbors(URN_SEC, 0, -1, newp);
    }
    else
      exchange_particles(CommList+SLHeadTail[1], SecSLHeadTail[0], oldp, 0,
                         currmode, stats);
    for (i=0; i<nnns2; i++)  NOfSend[i] = 0;
  } else {
    move_to_sendbuf_primary(Mode_PS(currmode), stats);
    exchange_primary_particles(currmode, stats);
  }
  count_population(nextmode, (Parent_New(pcode) ? 1 : 0), 0);
  currmode = Mode_Set_Any(nextmode);
  reb = 0;  oldp = newp;  pcode = newp>=0 ? 7 : 0;
}
```

Now, regardless of the accommodation mode and p'_n, we have the local per-grid histogram in `NOfPGrid[][][]` and particles in `Particles[]` which will stay in the local node's primary or secondary subdomain or travel to one of their neighbors. Therefore with this common setting, we do the followings fairly *differently* from those in `exchange_particles4p()`. First we call `exchange_population()` always to have the complete per-grid histogram of the local node's primary subdomain and its halo planes in `NOfPGridTotal[0][][]` and per-plane histogram in `NOfPGridZ[]` for the subdomain's interior, giving it `currmode` to let it know whether the reduction on local per-grid histograms is required.

Next, as done in level-4p, we define two execution-mode indicators namely $p_c =$ `psold` and $p_n =$ `psnew` being 1 if the local node has or will have its secondary subdomain/particles in the last or next step respectively, or 0 otherwise. They are referred to when we need to know if a data structure for subdomain/particles of $p \in \{0, 1\}$ has the portion for $p = 1$, because, for example, $p_n = 1$ means $p'_n = 1$ but it can be $p'_n = 1$ and $p_n = 0$.

[82] So far, there are no reasons to remember the accommodation pattern, but at least keeping it is safe.

Next, if $p'_n = 1$, we call `make_recv_list()`, with arguments `currmode`, `level`, `reb`, `oldp`, `newp` and `stats` of this function itself or modified ones due to anywhere accommodation, to build the receiver-side particle transfer schudule in `CommList[]` together with the pointers to secondary receiving and alternative secondary receiving blocks in `SecRList` and `AltSecRList`, and indices of primary receiving/sending, secondary receiving/sending and alternative secondary receiving/sending blocks in `RLIndex[]`, `SecRLIndex[]` and `AltSecRLIndex[]` respectively. If $p'_n = 0$ on the other hand, the transfer schedule is trivial and `PrimaryCommList[`p`][]` has primary receiving/sending blocks ($p = 0$) and secondary receiving/sending ones ($p = 1$) while indices for neighbors in both cases are commonly in `PrimaryRLIndex[]`.

Next, with the transfer schedule above, we call `make_send_sched()` with other input arguments `currmode`, `reb`, `oldp` and `newp`. The function builds the per-receiver sending histogram in `NOfSend[][][]`, lets `NOfPGrid[][][]` act the second role shown in §4.12.3, lets `NOfPGridOut[][][]` have the local per-grid histogram at the beginning of the next step, and lets `ZBound[]` and `HPlane[][]` have the values defined in §4.12.3. The function also gives us the number of primary particles including halo ones to be accommodated by the local node in the local array element `nacc`[0], the sum of those numbers for primary and secondary particles in `nacc`[1][83], and the number of sending particles $P_n^{\rm send}$ in the local variable `nsend`, through its output arguments.

Finally we exchange `NOfSend[][][]` by a hand-made all-to-all communicaion in neighboring families to have `NOfRecv[][][]` by `exchange_xfer_amount()`, which is slightly *different* because it takes care of $p'_n = 0$ case in which we have no particles to send as other nodes's secondary particles.

```
exchange_population(currmode);
psold = Parent_Old(pcode) ? 1 : 0;
psnew = Parent_New(pcode) ? 1 : 0;
if (nextmode) {
  make_recv_list(currmode, level, reb, oldp, newp, stats);
  rlist[0] = CommList;  rlist[1] = SecRList;
  rlidx[0] = RLIndex;   rlidx[1] = SecRLIndex;
} else {
  rlist[0] = PrimaryCommList[0];  rlist[1] = PrimaryCommList[1];
  rlidx[0] = rlidx[1] = PrimaryRLIndex;
}
make_send_sched(reb, pcode, oldp, newp, rlist, rlidx, nacc, &nsend);
exchange_xfer_amount(trans, psnew, nextmode);
```

The next part is very level-4s's own, in which we copy the per-grid histogram from its substance `NOfPGridOut[`p`][][]` to its shadow `NOfPGridOutShadow[`p`][][]` for $p \in \{0, p_n\}$, in order to show it to the simulator body but keeping its original version from being (accidentally) tampered by the simulator body, by scannig the substance by `For_All_Grid()`. At the same time and also for $p \in \{0, p_n\}$, we build the per-grid index in `NOfPGridTotal[`p`][][]` for the use in `sort_particles()`, `move_and_sort()` and `sort_received_particles()`, the substance array `NOfPGridIndex[`p`][][]` and shadow one `NOfPGridIndexShadow[][][]` by accumulating the values of the per-grid histogram to let the former two array have the values defined in §4.12.3. One caution is that the shadow can have values greater by one than the substance for Fortran coded simulator body, i.e., when `specBase` = 1. The other caution is that the scanning of the per-grid histogram includes exterior halo planes of e^g thick,

[83] This is *different* from level-4p in which `nacc`[1] has the number of secondary particles rather than the sum.

and that on secondary subdomain is for the next step whose size is in `GridDesc`[2] when we have transitional state of helpand-helper configuration as given to the first argument of `For_All_Grid()`.

```
for (ps=0,tp=0; ps<=psnew; ps++) {
  const int psor2 = ps ? trans + 1 : 0;
  const int sb = specBase;
  for (s=0; s<ns; s++) {
    dint *npgt=NOfPGridTotal[ps][s];
    int *npgo=NOfPGridOut[ps][s], *npgos=NOfPGridOutShadow[ps][s];
    int *npgi=NOfPGridIndex[ps][s], *npgis=NOfPGridIndexShadow[ps][s];
    For_All_Grid(psor2, -exti, -exti, -exti, exti, exti, exti) {
      const int g = The_Grid(),  np = npgo[g];
      npgos[g] = np;  npgt[g] = npgi[g] = tp;  npgis[g] = tp + sb;  tp += np;
    }
  }
}
```

Now we start position-aware transfer of non-halo particles, similarly to `exchange_particles4s()` but differently from it in some details. If $Q_n + P_n^{\mathrm{send}} > P_{lim} =$ `nOfLocalPLimit`, where $Q_n =$ `nacc`[1], to mean we cannot move all particles in `Particles`[] to `SendBuf`[] with sorting, we perform a partially position-aware transfer only takeing care of the node to accommodate particles in each grid-voxel. In addition and *differently* from level-4p, we perform this type of transfer when helpand-helper reconfiguration takes place as disussed shortly. The partially position-aware transfer is at first done by `move_to_sendbuf_4s()` being the level-4s version of `move_to_sendbuf_sec4p()` but it works even on the case of $p'_n = 0$, while its argument set is almost equivalent to the level-4p counterpart except that we add `psnew` $= p_n$ to know the size of $rbuf(p, s)$ in $pbuf_i(p, s)$ and `nacc`[1] has the sum of primary and secondary paticles. Then we call `xfer_particles()`, which is slightly *different* from its level-4p version because it takes care of $p'_n = 0$ case in which we have no particles to send as other nodes's secondary particles, to have non-halo particles to accommodate in `Particles`[].

The next step is very level-4s's own and calls `make_bxfer_sched()` to build the transfer schedule of particles in vertical halo planes. Note that calling the function here means that `NOfPGrid`[][][] for `move_to_sendbuf_4s()` should have non-negative values less than 2^{32} indicating that particles in each grid-voxels stay in the local node or are sent to other node which the array element specifies. This fact is important when helpand-helper reconfiguration takes place, and thus a grid-voxel in the *old* secondary subdomain having particles for other nodes can be included in the vertical halo planes of *new* secondary subcuboid. That is, the particles in such grid-voxel are at first packed in `Particles`[] or moved to `SendBuf`[] properly by `move_to_sendbuf_4s()` as `NOfPGrid`[][][] indicates and then transferred, and after that the array elements for grid-voxels in vertical interior halo plane and exterior pillar are let have special values to copy particles in them to `BoundarySendBuf`[] by `sort_particles()`, being slightly *different* from its level-4p version because of this operation. This is the reason why we perform the partially position-aware transfer on helpand-helper reconfiguration.

On the other hand, if helpand-helper reconfiguration does not take place and $Q_n + P_n^{\mathrm{send}} \leq P_{lim}$, we at first call `make_bxfer_sched()` to let `NOfPGrid`[][][] for grid-voxels in vertical interior halo plane and exterior pillar have special values. Since particles in a grid-voxel in a vertical interior halo plane definitely stays in the local node, it is safe to let the element of `NOfPGrid`[][][] for the grid-voxel have a negative value because it should have been 0. Then we move particles in `Particles`[] to `SendBuf`[] sorting those staying

in the primary/secondary subcuboid by `move_and_sort()` being the level-4s version of `move_and_sort_secondary()` but it works even on the case of $p'_n = 0$, while its argument set is almost equivalent to the level-4p counterpart except for `nacc`[1] as discussed above with `move_to_sendbuf_4s()`. In addition, it copies particles in vertical interior halo planes to `BoundarySendBuf`[] as in `sort_particles()`. Then we transfer particles by `xfer_particles()` and then move received particles in $rbuf(p, s)$ to `SendBuf`[] sorting them by `sort_received_particles()`, which is slightly *different* from its level-4p version again because it also copies particles in vertical interior halo planes to `BoundarySendBuf`[].

```
if (trans || (dint)nacc[1]+(dint)nsend>(dint)nOfLocalPLimit) {
  move_to_sendbuf_4s(nextmode, psold, psnew, trans, oldp, nacc, nsend,
                     stats);
  xfer_particles(trans, psnew, nextmode, SendBuf);
  make_bxfer_sched(trans, psnew, rlist, rlidx);
  sort_particles(nextmode, psnew, stats);
} else {
  make_bxfer_sched(0, psnew, rlist, rlidx);
  move_and_sort(nextmode, psold, psnew, oldp, nacc, stats);
  xfer_particles(trans, psnew, nextmode, SendBuf+nacc[1]);
  sort_received_particles(nextmode, psnew, stats);
}
```

The last part of this function is very level-4s's own. We call `xfer_boundary_particles_v()` twice to transfer particles in vertical halo planes, for yz-planes with argument `d` $= 0$ at first and then for xz-planes with `d` $= 1$, *relaying* the particles in exterior pillars from west/east neighbor to south/north one. Then we call `xfer_boundary_particles_h()` to transfer those in horizontal halo planes including those received by the calls of `xfer_boundary_particles_v()`. The argument `psnew` $= p_n$ is commonly passed to them to specify whether secondary halo particles are transfered, and $t =$ `trans` is given to the former to specify that `GridDesc`$[t+1]$ is referred in `For_All_Grid()` to scan vertical exterior halo planes in per-grid histogram and per-grid index to find the locations for received secondary halo particles.

```
  xfer_boundary_particles_v(psnew, trans, 0);
  xfer_boundary_particles_v(psnew, trans, 1);
  xfer_boundary_particles_h(psnew);
}
```

Here we revisit the issue that this function should work well not only in the case of secondary mode in the next step as the level-4p counterpart works but also of primary mode, showing data structures for position-aware particle transfer have followings consistent with primary mode case.

- `NOfPGrid`[0][][] has been set according to `PrimaryCommList`[0][] so that all primary particles in the interior of n's primary subdomain stay in n while all in the exterior are sent to n's neighbors. As for those in vertical interior halo planes, they have indices of `BoundarySendBuf`[] for n's neighbors whose primary subdomains share vertical surfaces with n's ones. Those in exterior pillars also have indices of `BoundarySendBuf`[] for n's south/north neighbors to relay particles from west/east ones. If n has secondary particles, `NOfPGrid`[1][][] are set according to `PrimaryCommList`[1][] so that all of them are sent to n^p_{old} or its neighbors. Since `make_bxfer_sched()` does not modify anything in `NOfPGrid`[1][][], `move_and_sort()` should properly work on old secondary particles to send them to the nodes.

- `NOfPGridOut`[0][][] has been set according to the primary receiving block in `Primary CommList`[0][$\lfloor 3^D/2 \rfloor$] and thus has `NOfPGridTotal`[0][][] for all grid-voxels in n's primary subdomain and its horizontal exterior halo planes. On the other hand, `NOfPGridOut`[1][][] is meaningless because it has not been modified when n will not have secondary subdomain, but will not be referred to.

- `ZBound`[0][] has been set according to `PrimaryCommList`[0] and thus has $\{0, \delta_z(n)\}$, while `ZBound`[1][] has not been modified after its reinitialization and thus has $\{0, 0\}$.

- `HPlane`[0][] has been set according to `PrimaryCommList`[0] and thus has transfer schedules with n's neighbors below and above its primary subdomain. `HPlane`[1][] is meaningless because it has not been modified, but will not be referred to.

- `VPlane`[] has been set according to `PrimaryCommList`[0] and thus has transfer schedules with n's neighbors whose primary subdomains share vertical surfaces of n's one. `VPlaneHead`$[d][p][\beta] = h_{4d+2p+\beta}$ is also set properly according to the number of transfer schedules, $h_0 = 0$, $h_i = h_{i-1} + \{0, 1\}$ for $i \in \{1, 4\}$ according to the $\{0, 1, 2, 3\}$ for the first four elements and 4 for remaining 5 elements, in fact.

- `GridDesc`[0] has never been modified after the initilization for n's primary subdomain in `init4s()`, and, if n has secondary subdomain, `GridDesc`[1] has also been kept unchanged since the last helpand-helper reconfiguration having descriptors for the subdomain.

- `TotalPNext`[0][] has been set according to `PrimaryCommList`[0][$\lfloor 3^D/2 \rfloor$] and thus has the number of primary particles n will accommodate in the next step. `TotalPNext`[1][] has been zero-cleared and has not been modified after that.

- `NOfSend`[0][][] has been set according to `PrimaryCommList`[][] which tells us all particles sent to neighbors of n, and *parent*(n) if any, are primary for them. Therefore `NOfSend`[1][][] has not been modified after zero-clearing in the last step or in this function with anywhere accommodation. Since `RealDstNeighbors`[0][0] is kept unchanged even when we were in secondary mode, elements in `NOfSend`[0][] have been sent to the nodes whose primary subdomains are neighbors of n and *parent*(n) if any.

- `NOfRecv`[0][][] has been set according to `RealSrcNeighbors`[0][0] which is kept unchanged even when we were in secondary mode. Therefore, its elements have the number of n's primary particles currently accommodated by the nodes whose primary or secondary subdomains are neighbors of n's primary subdomain. `NOfRecv`[1][][] is meaningless because it has not been modified since the last step, but will not be referred to.

- `GridOffset`[0][] has never been modified after the initilization for n's primary subdomain in `init4s()`, and, if n has secondary subdomain, `GridOffset`[1][] has also been kept unchanged since the last helpand-helper reconfiguration having proper offsets for neighbors of the subdomain.

- `RealDstNeighbors`[0][0] and `RealSrcNeighbors`[0][0] has been kept unchanged since the last helpand-helper reconfiguration so that the former tells us the set of nodes whose primary subdomains are neighboring to n's primary and secondary subdomains while the latter does those whose primary or secondary subdomains are neighboring to n's primary subdomain. Other elements $[t][p]$ where $t \neq 0$ or $p \neq 0$ are meaningless and thus will not be referred to.

4.13.15 count_population()

count_population() The function count_population(), called solely from exchange_particles4s() with anywhere accmmodation, counts the particle population in each grid-voxel to have the per-grid histogram in NOfPGrid[][][] after we perform non-position-aware particle transfer. It also lets each of Particles[].nid have the subdomain code $\lfloor 3^D/2 \rfloor$ and the grid-position in the primary/secondary subdomain of the local node, copies TotalPNext[] to TotalP[], lets primaryParts and totalParts have the numbers of primary and all particles, and lets nOfInjection=0, as if we had the result of the non-position-aware particle transfer at the call of oh4s_transbound().

The function is perfectly equivalent to its level-4p counterpart described in §4.10.36. Only one *difference* is that the function is now called solely from exchange_particles4s() regardless of the execution mode in next step, because try_primary4s() does not have its own particle transfer mechanism which try_primary4p(), the other caller in level-4p, has[84].

```
static void
count_population(const int nextmode, const int psnew, const int stats) {
  int ps, s, t, i, j, tp;
  const int ns=nOfSpecies, exti=OH_PGRID_EXT;
  Decl_For_All_Grid();
  Decl_Grid_Info();

  if (stats) oh1_stats_time(STATS_TB_SORT, nextmode);
  for (ps=0,t=0,j=0,tp=0; ps<=psnew; ps++) {
    for (s=0; s<ns; s++,t++) {
      dint *npgs = NOfPGrid[ps][s];
      const int tpn = TotalP[t] = TotalPNext[t];
      tp += tpn;
      For_All_Grid(ps, -exti, -exti, -exti, exti, exti, exti)
        npgs[The_Grid()]=0;
      for (i=0; i<tpn; i++,j++) {
        const int g = Grid_Position(Particles[j].nid);
        npgs[g]++;
        Particles[j].nid = Combine_Subdom_Pos(OH_NBR_SELF, g);
      }
    }
    if (ps==0)  primaryParts = tp;
  }
  totalParts = tp;  nOfInjections = 0;
}
```

4.13.16 exchange_population()

exchange_population() The function exchange_population(), called solely from exchange_particles4s(), sums up local per-grid histograms in the local node's primary family if we have secondary mode configuration[85] indicated by currmode argument, and then gathers the per-grid histograms in neighbors' halo planes to have the complete per-grid histogram in NOfPGridTotal[0][][].

[84] Therefore, the argument stats is meaningless because it is always 0 and nextmode as well because it is meaningful only when stats is non-zero, but we keep them to make this function perfectly equivalent to the level-4p counterpart.

[85] That is, we were in secondary mode with normal accommodation or will be in secondary mode with anywhere accommodation.

The function is similar to its level-4p counterpart described in §4.10.15, but it has various *differnces* as follows.

- Since we always need per-grid histograms in neighbors' halo planes for halo particle transfer, this function is called regardless of the current and next execution mode and accommodation pattern to have the complete per-grid histogram in `NOfPGridTotal`[0][][][] always.
- Since we need per-plane histogram in `NOfPGridZ`[], this function calculate the particle population in each xy-plane of interior of the local node's primary subdomain.
- Since the receiving planes to recieve the per-grid histograms in neighbors' halo planes are $2e^g$ thick and they are added to those in local node's halo planes being e^g thick for each of interior and exterior ones, the arguments of `add_population()` are level-4s specific.
- Since the level-4s library is only for 3-dimentional simulations, dimension-dependent constructs are elminated.

```
static void
exchange_population(const int currmode) {
  const int ns=nOfSpecies;
  int s, zz;
  dint **npg = NOfPGridTotal[0];
  const int ct=nOfExc-1;
  const int ext = OH_PGRID_EXT,  ext2 = ext<<1,  ext3 = ext*3;
  const int x = GridDesc[0].x,  y = GridDesc[0].y,  z = GridDesc[0].z;
  const int w = GridDesc[0].w,  dw = GridDesc[0].dw;
  Decl_For_All_Grid();
```

At first, if we have secondary mode configuration, we sum up local per-grid histograms `NOfPGrid`[p][][][] in the local node's primary family, where $p = 0$ for the local node while $p = 1$ for its helpers, to have the sum in `NOfPGridTotal`[0][][][] by `reduce_population()`. On the other hand, if we have primary mode configuration, we copy elements `NOfPGrid`[0][s][$gidx(x, y, z)$] to the corresonding elements in `NOfPGridTotal`[][][][] using `For_All_Grid()` for all $s \in [0, S)$ and $(x, y, z) \in [-e^g, \delta_x(n)+e^g) \times [-e^g, \delta_y(n)+e^g) \times [-e^g, \delta_z(n)+e^g)$ for the local node n, because we need to keep `NOfPGrid`[][][][] unchanged for particle transfer[86]. Therefore, the *base* per-grid histogram is built in `NOfPGridTotal`[0][][][] always *unlike* in level-4p.

```
  if (Mode_PS(currmode))  reduce_population();
  else {
    for (s=0; s<ns; s++) {
      dint *npgs = NOfPGrid[0][s],  *npgt = npg[s];
      For_All_Grid(0, -ext, -ext, -ext, ext, ext, ext)
        npgt[The_Grid()] = npgs[The_Grid()];
    }
  }
```

[86] This copy cannot be done calling `reduce_population()` blindly because the `prime` element of `MyComm` is not meaningful with primary mode configuration and thus not necesarily be `MPI_COMM_NULL`

Now, for each $s \in [0, S)$, we gather sending planes of all $2D = 6$ neighbors to the local node's receiving planes by `oh3_exchange_borders()` giving it the base per-grid histogram `NOfPGridTotal`$[0][s][]$ and $C-1$ being the entry for per-grid histograms in `BorderExc`$[][][][]$. Its second argument for the secondary subdomain's array is `NULL` because we don't broadcast the receiving planes to the helpers as indicated its forth argument `bcast` $= 0$.

Then we add each of $2D$ receiving planes to each halo planes of d-th dimensional from $d = D - 1 = 2$ to 0 to have the complete per-grid histogram. The addition is performed by a series of calls of `add_population()` for each receiving/halo plane pair. More specifically, the d-th dimensional halo planes to which we add receiving planes are specified as $[\beta_0^l, \beta_0^u) \times [\beta_1^l, \beta_1^u) \times [\beta_2^l, \beta_2^u)$ where $[\beta_k^l, \beta_k^u)$ is specified as follows.

$$[\beta_k^l, \beta_k^u) = \begin{cases} [-3e^g, \delta_k(n) + 3e^g) & k < d \\ [-e^g, e^g) & k = d \text{ and lower} \\ [\delta_k(n) - e^g, \delta_k(n) + e^g) & k = d \text{ and upper} \\ [-e^g, \delta_k(n) + e^g) & k > d \end{cases}$$

The function is given the arguments for the base per-grid histogram, the lower and upper bound of the halo planes in each axis shown above, and the offset of the receiving planes from boundary planes namely;

$$gidx(\beta_0, \cdots, \beta_d \pm 2e^g, \cdots, \beta_{D-1}) - gidx(\beta_0, \cdots, \beta_d, \cdots, \beta_{D-1}) = \pm 2e^g \prod_{k=0}^{d-1} (\delta_k^{\max} + 6e^g)$$

where $-/+$ for lower/upper boundary. The method for the addition is same as that we used in the sample code's function `add_boundary_current()` shown in §3.13.

After the addition, we add $\sum_{y=0}^{\delta_y(n)} \sum_{x=0}^{\delta_x(n)} \mathcal{P}_T(0, s, gidx(x, y, z))$ to `NOfPGridZ`$[z]$ to have per-plane histogram $\mathcal{P}_Z(z)$ in it when we finish this function.

```
    for (zz=0; zz<z; zz++)  NOfPGridZ[zz] = 0;
    for (s=0; s<ns; s++) {
      dint *npgt = npg[s];
      oh3_exchange_borders(npgt, NULL, ct, 0);
      add_population(npgt, -ext3, x+ext3, -ext3, y+ext3, -ext, ext,  -dw*ext2);
      add_population(npgt, -ext3, x+ext3, -ext3, y+ext3, z-ext, z+ext, dw*ext2);
      add_population(npgt, -ext3, x+ext3, -ext, ext, -ext, z+ext, -w*ext2);
      add_population(npgt, -ext3, x+ext3, y-ext, y+ext, -ext, z+ext,  w*ext2);
      add_population(npgt, -ext,  ext,   -ext, y+ext, -ext, z+ext, -ext2);
      add_population(npgt, x-ext, x+ext, -ext, y+ext, -ext, z+ext,  ext2);

      For_All_Grid(0, 0, 0, 0, 0, 0, 0)
        NOfPGridZ[Grid_Z()] += npgt[The_Grid()];
    }
  }
```

4.13.17 `reduce_population()`

`reduce_population()` The function `reduce_population()`, called solely from `exchange_population()`, performs reduce communications in primary ($p = 0$) and secondary ($p = 1$) family members to sum up `NOfPGrid`$[p][][]$ to have the sum in `NOfPGridTotal`$[0][][]$.

The function works equivalently to its level-4p counterpart described in §4.10.18 when the counterpart is given `MPI_Reduce()` through its argument[87]. That is, the function performs red-black reduction as done in `oh1_reduce()` but keeping the source array `NOfPGrid`[][][] rather than overwriting it by `MPI_IN_PLACE` option, and if the `prime` element of `MyComm` is `MPI_COMM_NULL`, copies `NOfPGrid`[0][][] into `NOfPGridTotal`[0][][] explicitly by `memcpy()`. Also equivalent to the level-4p counterpart, the base index and the number of elements to be reduced are specified in `FieldDesc`[$F-1$]`.red.base` and its element `size`[p] for the per-grid histogram.

```
static void
reduce_population() {
  const int ft=nOfFields-1;
  const int base = FieldDesc[ft].red.base;
  const int *size = FieldDesc[ft].red.size;

  if (MyComm->black) {
    if (MyComm->prime!=MPI_COMM_NULL)
      MPI_Reduce(NOfPGrid[0][0]+base, NOfPGridTotal[0][0]+base, size[0],
                 MPI_LONG_LONG_INT, MPI_SUM, MyComm->rank, MyComm->prime);
    if (MyComm->sec!=MPI_COMM_NULL)
      MPI_Reduce(NOfPGrid[1][0]+base, NOfPGridTotal[1][0]+base, size[1],
                 MPI_LONG_LONG_INT, MPI_SUM, MyComm->root, MyComm->sec);
  } else {
    if (MyComm->sec!=MPI_COMM_NULL)
      MPI_Reduce(NOfPGrid[1][0]+base, NOfPGridTotal[1][0]+base, size[1],
                 MPI_LONG_LONG_INT, MPI_SUM, MyComm->root, MyComm->sec);
    if (MyComm->prime!=MPI_COMM_NULL)
      MPI_Reduce(NOfPGrid[0][0]+base, NOfPGridTotal[0][0]+base, size[0],
                 MPI_LONG_LONG_INT, MPI_SUM, MyComm->rank, MyComm->prime);
  }
  if (MyComm->prime==MPI_COMM_NULL)
    memcpy(NOfPGridTotal[0][0]+base, NOfPGrid[0][0]+base,
           size[0]*sizeof(dint));
}
```

4.13.18 `add_population()`

`add_population()` The function `add_population()`, called solely from `exchange_population()` but $2DS$ times, adds the elements in a receiving plane set in per-grid histogram for a species, specified by its argument `npd` being `NOfPGridTotal`[0][s][], to the halo plane (set) specified by arguments as [`xl`, `xu`) × [`yl`, `yu`) × [`zl`, `zu`), whose values are shown in §4.13.16. The function is perfectly equivalent to its level-4p counterpart described in §4.10.16 but its arguments given from the caller are *different* as discussed in §4.13.16.

```
static void
add_population(dint *npd, const int xl, const int xu, const int yl,
               const int yu, const int zl, const int zu,
               const int src) {
```

[87] Therefore, the function is very similar to the counterpart literally and thus have `NOfPGridTotal[1][0]+base` as the second argument of `MPI_Reduce()` for the reduction in secondary family knowing the argument is meaningless.

```
  dint *nps=npd+src;
  Decl_For_All_Grid();

  For_All_Grid_Abs(0, xl, yl, zl, xu, yu, zu)
    npd[The_Grid()] += nps[The_Grid()];
}
```

4.13.19 make_recv_list()

make_recv_list()

The function `make_recv_list()`, called solely from `exchange_particles4s()` when we will be in secondary mode in the next step, scans per-plane histogram to build primary receiving block, and then exchanges the block between neighbors to have primary sending block and broadcast them for secondary receiving/sending and alternative secondary receiving/sending blocks for helpers. Its arguments `currmode`, `level`, `reb` and `stats` are perfectly equivalent to those of the caller `exchange_particles4s()`, while `oldp` $= n^p_{old}$ and `newp` $= n^p_{new}$ are parents in the last and next simulation step.

The function is similar to its level-4p counterpart described in §4.10.19 but has various *differences* as follows.

- Since we scan per-plane histogram rather than per-grid histogram to build the receiving schedule, the arguments set of the callee `sched_recv()` and its `context` are different from those of the counterpart.
- Since we do not have hot-spots, we can be unaware of the possiblity the primary receiving block generated by `sched_recv()` has hot-spot records especially at its tail.
- Since we need primary sending and secondary sending blocks even with anywhere accommodation for halo paticle transfer scheduling, the function works without respect to the accommodation pattern.
- On the helpand-helper reconfiguration, level-4s's alternative secondary receiving block is followed by alternative secondary sending block being the copy of n^p_{new}'s primary sending block necessary for halo particle transfer.
- This function does not return anything because we do not have hot-spot sending block whose head pointer is returned by the counterpart.

```
static void
make_recv_list(const int currmode, const int level, const int reb,
               const int oldp, const int newp, const int stats) {
  const int me = myRank, ns=nOfSpecies, nn=nOfNodes, nnns=nn*ns;
  const int nn2 = nn<<1;
  struct S_node *nodes = reb ? NodesNext : Nodes;
  struct S_node *mynode = nodes + me;
  struct S_node *ch;
  struct S_recvsched_context
    context = {0, 0, 0, CommList};
  int rlsize, rlidx;
  const int ft=nOfFields-1;
  const int npgbase = FieldDesc[ft].bc.base;
  const int *npgsize = FieldDesc[ft].bc.size;
  const int zmax = GridDesc[0].z-1;
  struct S_commlist *lastrl;
```

```
int i;
```

First, the function builds primary receiving block by calling `sched_recv()` for the local node's helpers and then the node itself to determine the subcuboid assigned to each node. *Unlike* level-4p, we may scan the family members in either of helper-first or helpand-first order because of no hot-spots, but we followed the level-4s's convention, i.e., helper-first.

The arguments for the function are as follows, where *NN* is `Nodes[]` if `reb` is false, or `NodesNext[]` otherwise, for *new* helpands.

- *Unlike* level-4p, the function does not have the argument `currmode`, and `reb` is different from the counterpart because it is true iff we have normal accommodation[88], helpand-helper reconfiguration is taking place, and the call is for a helper. That is, `reb` is true when `get` has the expected number of particles the node will accommodate, or false the number is calculated by `get` + `stay` as discussed in §4.10.20.
- As in level-4p, `get` is $R_n^{\mathrm{get}} = NN[n]$`.get.prime` for the local node n, or $Q_m^{\mathrm{get}} = NN[m]$`.get.sec` for its helper m, to specify the baseline number of receiving (if positive) or sending (if negative) primary/secondary particles of n or m.
- As in level-4p, `stay` is $Q_n^n = NN[n]$`.stay.prime` for the local node n, or $Q_m^n = NN[m]$`.stay.sec` for its helper m, to specify the number of primary/secondary particles currently accommodated by n or m. The value is not useful if `reb` passed to the function is true as described above.
- As in level-4p, `nid` is the node identifier of the local node n or its helper m.
- As in level-4p, `tag` is 0 for the local node, or NS for its helpers, to distinguish helpand and helpers and to be set into `S_commlist` record element `tag`.
- *Unlike* level-4p, `context` is a `S_recvsched_context` structure whose *difference* from level-4p's counterpart is discussed in §4.12.4. Similarly to level-4p, however, its elements `z`, `nptotal` and `nplimit` are 0 at initial, while `cptr` is initialized to point the head of `CommList[]`.

Similarly to level-4p, we have primary receiving block by the sequence of calls, but its last record does not necessary has the largest z-coordinate of the local node n's primary subdomain, $\delta_z(n) - 1$, because the topmost xy-plane can have no particles. If so, we need to make the last record's `region` have $\delta_z(n) - 1$ but we cannot simply do it by overwriting the record in the case that the subdomain has no particles at all and thus the primary receiving block is empty. In this case, we add a record to assign all xy-planes to the local node n letting `region` element be $\delta_z(n) - 1$.

```
for (ch=mynode->child; ch; ch=ch->sibling)
  sched_recv(reb, ch->get.sec, ch->stay.sec, ch->id, nnns, &context);
sched_recv(0, mynode->get.prime, mynode->stay.prime, me, 0, &context);

rlidx = rlsize = context.cptr - CommList;  lastrl = context.cptr - 1;
if (rlsize==0) {
  struct S_commlist *rl = CommList;
  rl->rid = me;  rl->tag = 0;  rl->sid = 0;  rl->count = 0;
```

[88] Even in level-4p, it is assured that `reb` is false if we have anywhere accommodation but this fact is not exploited in the implementation.

```
  rl->region = zmax;
  rlidx = rlsize = 1;
} else
  lastrl->region = zmax;
```

Next, *unlike* level-4p, the local node n exchanges its primary receiving block between its neighbors to have primary sending blocks, *regardless* of the accommodation pattern because we need the blocks always for halo particle transfer. The procedure to do that is, however, exactly same as that shown in §4.10.19.

```
for (i=0; i<OH_NEIGHBORS; i++) {
  const int dst=DstNeighbors[i], src=SrcNeighbors[i];
  int rc;
  MPI_Status st;
  if (dst==me) {
    RLIndex[i] = 0;  continue;
  }
  if (src>=0) {
    RLIndex[i] = rlidx;
    if (dst>=0)
      MPI_Sendrecv(CommList, rlsize, T_Commlist, dst, 0,
                   CommList+rlidx, nn2, T_Commlist, src, 0, MCW, &st);
    else
      MPI_Recv(CommList+rlidx, nn2, T_Commlist, src, 0, MCW, &st);
    MPI_Get_count(&st, T_Commlist, &rc);  rlidx += rc;
  } else {
    if (dst>=0)
      MPI_Send(CommList, rlsize, T_Commlist, dst, 0, MCW);
    RLIndex[i] = (src<-nn) ? rlidx : RLIndex[FirstNeighbor[i]];
  }
}
```

The next and final step of the function, in which we broadcast primary receiving and primary sending blocks to helpers including those after the helpand-helper reconfiguration if it took place, is similar to level-4p. However there are two *differences* from level-4p. The major one is that alternative secondary receiving block of a node is followed by alternative secondary sending block being the copy of primary sending block of the new helpand. Therefore, we have `AltSecRLIndex`$[3^D + 1]$ whose element $[k]$ $(k < 3^D)$ is the index of the first record for the k-th neighbor of the helpand and $[3^D]$ is the size of the alternative secondary receiving/sending block, and broadcast this array to new helpers when the helpand-helper reconfiguration took place instead of the size of primary receiving block. The minor one is that this function does not return anything to its caller and thus it is unnecessary to keep track of the tail of `CommList`[].

```
RLIndex[OH_NEIGHBORS] = rlidx;  SecRLIndex[OH_NEIGHBORS] = 0;
AltSecRList = SecRList = CommList + rlidx;  AltSecRLIndex[OH_NEIGHBORS] = 0;
if (Mode_PS(currmode)) {
  oh1_broadcast(RLIndex, SecRLIndex, OH_NEIGHBORS+1, OH_NEIGHBORS+1,
                MPI_INT, MPI_INT);
  oh1_broadcast(CommList, SecRList, rlidx,
                SecRLIndex[OH_NEIGHBORS], T_Commlist, T_Commlist);
  AltSecRList += SecRLIndex[OH_NEIGHBORS];
}
```

```
    if (reb) {
      build_new_comm(currmode, -level, 2, stats);
      update_descriptors(oldp, newp);
      set_grid_descriptor(2, newp);
      update_real_neighbors(URN_TRN, Mode_PS(currmode), oldp, newp);
      oh1_broadcast(RLIndex, AltSecRLIndex, OH_NEIGHBORS+1, OH_NEIGHBORS+1,
                    MPI_INT, MPI_INT);
      oh1_broadcast(CommList, AltSecRList, RLIndex[OH_NEIGHBORS],
                    AltSecRLIndex[OH_NEIGHBORS], T_Commlist, T_Commlist);
    }
    oh1_broadcast(NOfPGridTotal[0][0]+npgbase, NOfPGridTotal[1][0]+npgbase,
                  npgsize[0], npgsize[1], MPI_LONG_LONG_INT, MPI_LONG_LONG_INT);
  }
```

4.13.20 `sched_recv()`

`sched_recv()` The function `sched_recv()`, called solely from `make_recv_list()` but $|F(n)|$ times for the local node n, scans per-plane histogram to determine the set of xy-planes, i.e., the subcuboid to be assigned to a node `nid` $= m_f$ being the f-th member of the local node's primary family, whose expected number of accommodating primary (`tag` $= 0$) or secondary (`tag` $= NS$) particles is determined by the argumetns `reb`, `get` and `stay`. The scanning and assignment context is kept in the `S_recvsched_context` structure argument `context` whose elements and their definitions were given in §4.13.19. This function is somewhat similar to its level-4p counterpart shown in §4.10.20 but much *simpler* than it because there are no hot-spots and the assignment unit is a xy-plane rather than a grid-voxel.

```
  static void
  sched_recv(const int reb, const int get, const int stay, const int nid,
             const int tag, struct S_recvsched_context *context) {
    const int z0=context->z;
    dint nptotal=context->nptotal;
    dint nplimit=context->nplimit;
    struct S_commlist *cptr=context->cptr;
    const int ns=nOfSpecies;
    int z;
    const int zz = GridDesc[0].z;
```

First, the function calculate $\mathcal{P}_\Lambda(f) = \mathcal{P}_\Lambda(f-1) + Q^n_{m_f} = \sum_{i=0}^{f} Q^n_{m_i}$ where $\mathcal{P}_\Lambda(f-1)$ is given through the `nplimit` element of `context`, just depending on the `reb` argument *unlike* the level-4p counterpart, but the calculation is logically equivalent to that shown in §4.10.20 because $Q^n_{m_f}$ = `get` if `reb` or $Q^n_{m_f}$ = `get` + `stay` otherwise, as discussed in §4.13.19.

```
    if (reb)
      nplimit += get;
    else
      nplimit += get + stay;
```

Then after writing $\mathcal{P}_\Lambda(f)$ back to `context`, we examine if $\mathcal{P}_\Sigma(z_0) = \sum_{z=0}^{z_0-1} \mathcal{P}_Z(z) \geq \mathcal{P}_\Lambda(f)$ where z_0 and $\mathcal{P}_\Sigma(z_0)$ are given through `z` and `nptotal` elements of `context`. If this

ineqaulity holds to mean that we don't have any xy-planes to assign m_f, we simply return from this function without adding `S_commlist` record[89].

Now we have some xy-planes to assign to m_f and thus set `S_commlist` record elements, `rid` to m_f and `tag` to that given as the argument[90].

```
    context->nplimit = nplimit;
    if (nptotal>=nplimit)  return;
    cptr->rid = nid;  cptr->tag = tag;  cptr->sid = 0;  cptr->count = 0;
```

Now we scan per-plane histogram entries until we find z such that $\mathcal{P}_\Sigma(z) \geq \mathcal{P}_\Lambda(f)$[91], to let `region` element of the `S_commlist` record be such z, and then return to the caller writing $z+1$, $\mathcal{P}_\Sigma(z)$ and the pointer to the next record back to `context`'s elements `z`, `nptotal` and `cptr` respectively. Note that z found here is not necessary to satisfy $z - z_0 + 1 \geq e^g$ when $e^g > 1$ to mean that the height of m_f's subcuboid can be less than e^g and thus m_f's horizontal exterior halo planes can be horizontal interior halo planes in two or more nodes. Therefore, this function should be modified if we cope with $e^g > 1$ cases with the second solution shown in §4.13.6.

```
    for (z=z0; z<zz; z++) {
      nptotal += NOfPGridZ[z];
      if (nptotal>=nplimit) {
        cptr->region = z;  context->z = z + 1;
        context->nptotal = nptotal;  context->cptr = cptr + 1;
        return;
      }
    }
    local_errstop("per-plane histogram total %d is less than the total particle "
                  "population %d up to node %d",
                  nptotal, nplimit, nid);
  }
```

4.13.21 `make_send_sched()`

`make_send_sched()` The function `make_send_sched()`, called solely from `exchange_particles4s()`, scans primary receiving/sending and secondary receiving/sending blocks in `PrimaryCommList[][]` if we will be in primary mode, or those in `CommList[]` possibly together with alternative secondary receiving/sending blocks otherwise, to determine the node to which the local node n send the particles in each grid-voxel and the transfer schedule of those in horizontal halo planes. The function is given the following arguments; `reb` being helpand-helper reconfiguration indicator; the parent status code `pcode`; the identifiers of the old and (possibly) new helpand `oldp` $= n^p_{old}$ and `newp` $= n^p_{new}$; the pointer pair to the head of primary receiving and secondary receiving blocks `rlist`[2] being {`PrimaryCommList`[0], `PrimaryCommList`[1]} or {`CommList`, `SecRList`}; the pair of index arrays `rlidx`[2] for primary receiving/sending and secondary receiving/sending blocks being {`PrimaryRLIndex`[], `PrimaryRLIndex`[]} or {`RLIndex`[], `SecRLIndex`[]}; an array `nacc`[2] to accumulate the number of primary particles ([0] $= Q^n_n$) and whole particles ([1] $= Q_n$) to be accommodated by the local node;

[89] If the local node's primary subdomain has no particles, this inequality holds at the first call of `sched_recv()` with $z = 0$ and $f = 0$, because $\mathcal{P}_\Sigma(z) = \mathcal{P}_\Lambda(f) = 0$. Therefore, the caller `make_recv_list()` of `sched_recv()` will have no `S_commlist` records in primary receiving block in this case as discussed in §4.13.19.

[90] The elements `sid` and `count` are meaningless but we let them be 0 to avoid to leave them undefined.

[91] Such z must be found because $\mathcal{P}_\Sigma(\delta_z(n)) = \mathcal{P}_\Lambda(|F(n)| - 1)$. Therefore, if not found, we abort the execution with `local_error_stop()`

and the pointer `nsendptr` to *return* the number of particles $P_n^{\rm send}$ to be sent from the local node.

What this function does is somewhat similar to the level-4p counterpart shown in §4.10.21, but its implementation is substantially *different* from the counterpart because we don't have hot-spots but have halo particle transfer.

```
static void
make_send_sched(const int reb, const int pcode, const int oldp,
                const int newp, struct S_commlist *rlist[2],
                int *rlidx[2], int *nacc, int *nsendptr) {
  const int psold = Parent_Old(pcode) ? 1 : 0;
  const int psnew = Parent_New(pcode) ? 1 : 0;
  const int ns = nOfSpecies, ns2 = ns<<1,  nn = nOfNodes;
  const int tagt = OH_NBR_TCC * ns,  tagb = OH_NBR_BCC * ns;
  const int tag1 = OH_NEIGHBORS * ns;
  int s, ps, n;
  int nsend=0;
```

First, after clearing all elements of `TotalPNext`[2][S] as done in the level-4p counterpart, we scan primary receiving/sending blocks in `rlist`[0] with indices in `rlidx`[0] always, and then secondary receiving/sending blocks in `rlist`[1] with `rlidx`[1] if the local node has helpand in the last step, i.e., if `Parent_Old(pcode)` is true, to determine the node to accommodate particles in each xy-plane and thus each grid-voxel, by the following mechanisms to visit each neighbor of the local node and its helpand by `make_send_sched_body()` also as done in the level-4p counterpart; the head index of a sub-block for k-th neighbor is `rlidx`[k'] where $k' = 3^D - 1 - k$ because it corresponds to `SrcNeighbors`[] rather than `Neighbors`[p][]; we visit a neighbor twice or more if it occurred multiple times in `Neighbors`[p][] unless the neighbor is the local node itself or its helpand; we explicitly skip inexistent neighbors such that `Neighbors`[p][k] $< -(N+1)$ to keep `make_send_sched_body()` from processing empty sub-block. On the other hand, the procedure in `make_send_sched_body()` is quite simpler than that in the level-4p counterpart as discussed later, because we don't have hot-spots and the almost all operations on grid-voxels in the local node's subcuboid are performed by other functions shown later together with those for halo particle transfer. Another *difference* is that the function returns the number of particles to be sent to other nodes, which is accumulated in $P_n^{\rm send}$.

```
  for (s=0; s<ns2; s++)  TotalPNext[s] = 0;
  for (ps=0; ps<=psold; ps++) {
    const int root = ps ? oldp : myRank;
    for (n=0; n<OH_NEIGHBORS; n++) {
      const int nrev = OH_NEIGHBORS - 1 - n;
      int sdid = Neighbors[ps][n];
      if (sdid<0)  sdid = -(sdid+1);
      if (sdid<nn && (n==OH_NBR_SELF || sdid!=root))
        nsend += make_send_sched_body(ps, n, sdid, rlist[ps]+rlidx[ps][nrev]);
    }
  }
```

The next part is very *level-4s's own* and is for subcuboids assigned to the local node and halo particle transfer. We perform the following for primary subdomain and particles ($p = 0$), and for secondary subdomain and particles ($p = 1$) in the next step if the local

node will have its helpand in the step, after initializing `nacc`[0] = `nacc`[1] = 0 as the base of accumulation[92].

First, we call `make_send_sched_self()` with arguments `psor2` = p', `rlist` = $\lambda + \chi[\lfloor 3^D/2 \rfloor]$ and `naccptr` pointing `nacc`[p]. If $p = 1$, $n^p_{new} \neq n^p_{old}$ and n^p_{new} exists, $p' = 2$ and λ = `AltSecRList`[] indexed by χ = `AltSecRLIndex`[] for the transitional state of the helpand-helper configuration. Otherwise, i.e., $p = 0$ or $p = 1$ but $n^p_{new} = n^p_{old}$ or n^p_{new} is inexistent, $p' = p$ and λ = `rlist`[p] whose indices are in χ = `rlidx`[p]. By this call, we obtain the followings by the scan of primary receiving, secondary receiving or alternative secondary receiving block and the per-grid histogram for interior grid-voxels; lower and upper boundary of primary or secondary subcuboid in `ZBound`[p][]; halo particle transfer schedule for the bottom/top surface of the subcuboid in `HPlane`[p][]; the number of particles the local nodes will accommodate as primary ones or as the whole in `nacc`[p], for each species in `TotalPNext`[p][], and for each species and grid-voxel in `NOfPGridOut`[p][][]. Note that we give $\lambda + \chi[\lfloor 3^D/2 \rfloor]$ instead of λ for the receiving block because it is not at the head of λ when λ is `PrimaryCommList`[p].

Also Note that the local node can have no primary/secondary particles at all and, if so, `ZBound`[p][β] = $\{0, 0\}$, `HPlane`[p][β]`.nbor` = `MPI_PROC_NULL`[93], `nacc`[p] = $\{0,$ `nacc`[0]$\}$[p] and `TotalPNext`[p][s] = 0 for all $\beta \in \{0, 1\}$ and $s \in [0, S)$ because they remain unchanged, while `NOfPGridOut`[p][s][g] is explicitly let be 0 for all g in the local node's primary/ secondary subdomain including its exterior. If this emptiness happens, we skip the following procedures for halo particle transfer because the local node does not do anything for it.

Next we check if `HPlane`[p][β]`.nbor` = N to mean the bottom/top surface of the local node's subcuboid is that of its subdomain. If it holds for the bottom surface ($\beta = 0$) and the corresponding true-bottom neighbor m_b at the index $k_b = 3^1 + 3^0$ exsits, we replace the `HPlane`[p][0]`.nbor` with the `rid` element m'_b of the last `S_commlist` record, whose `region` is $\delta_z(m_b) - 1$, of the sub-block for k_b of the primary sending, secondary sending or alternative secondary sending block in λ indexed by $\chi[k'_b]$. We let `HPlane`[p][0]`.stag` be $(p' \cdot 3^D + k_b)S$ according to the `tag` element of the record being 0 ($p' = 0$) or not (NS, $p' = 1$) to indicate that particles in horizontal interior halo plane of the local node's subcuboid are sent to m'_b as its primary/secondary particles respectively. If such neighbor does not exist, on the other hand, because the bottom of the local node's subdomain is also the non-periodic bottom boundary of the system domain, we replace `HPlane`[p][0]`.nbor` with `MPI_PROC_NULL` to mean no halo particle transfer takes place through the bottom surface[94]. Similarly, if `HPlane`[p][1]`.nbor` = N holds for the top surface, we let its `nbor` and `stag` have values shown above, but with neighbor indices $k_t = 2 \cdot 3^2 + 3^1 + 3^0$, and referring to the first `S_commlist` record of the k_t's sub-block in primary sending, secondary sending or alternative secondary sending block.

The last operation of the loop for p is to let `nacc`[1] = `nacc`[0] if $p = 0$ to give `nacc`[1] the base of accumulation, which can be its eventual value when the local node will not have helpand in the next step.

```
nacc[0] = nacc[1] = 0;
for (ps=0; ps<=psnew; ps++) {
```

[92] Letting `nacc`[1] = 0 is necessary because the last operation of the loop for p to let `nacc`[1] be `nacc`[0] can be skipped if the local node does not have primary subcuboid.

[93] As set by `make_send_sched_self()`. Although `MPI_PROC_NULL` can be N with some MPI implementation, no confusion should happen because iff `ZBounhd`[p][1] = 0 then `HPlane`[p][β]`.nbor` = `MPI_PROC_NULL` means no communmication will take place for particles in horizontal halo planes rather than that the local node's subcuboid is the bottom/top one in its subdomain.

[94] And let `.stag` be k_bS knowing it is not referred to but to avoid confusions in debugging etc.

```
  int psor2;
  int *nbors, *ri;
  struct S_commlist *rl;
  struct S_hplane *hp = HPlane[ps];
  if (ps && Parent_New_Diff(pcode)) {
    psor2 = 2;  nbors = Neighbors[2];  rl = AltSecRList;  ri = AltSecRLIndex;
  } else {
    psor2 = ps;  nbors = Neighbors[ps];  rl = rlist[ps];  ri = rlidx[ps];
  }
  make_send_sched_self(psor2, rl+ri[OH_NBR_SELF], nacc+ps);
  if (ZBound[ps][OH_UPPER]==0)  continue;
  if (hp[OH_LOWER].nbor==nn) {
    int sdid = nbors[OH_NBR_BCC];
    if (sdid<0)  sdid = -(sdid+1);
    if (sdid<nn) {
      const int zmax = (SubDomains[sdid][OH_DIM_Z][OH_UPPER] -
                        SubDomains[sdid][OH_DIM_Z][OH_LOWER]) - 1;
      struct S_commlist *rlb = rl + ri[OH_NEIGHBORS-1-OH_NBR_BCC];
      int rlz = rlb->region;
      while (rlz<zmax)  rlz = (++rlb)->region;
      hp[OH_LOWER].nbor = rlb->rid;
      hp[OH_LOWER].stag = (rlb->tag) ? tagb + tag1 : tagb;
    } else {
      hp[OH_LOWER].nbor = MPI_PROC_NULL;
      hp[OH_LOWER].stag = tagb;
    }
  }
  if (hp[OH_UPPER].nbor==nn) {
    int sdid = nbors[OH_NBR_TCC];
    struct S_commlist *rlt = rl + ri[OH_NEIGHBORS-1-OH_NBR_TCC];
    if (sdid<0)  sdid = -(sdid+1);
    if (sdid<nn) {
      hp[OH_UPPER].nbor = rlt->rid;
      hp[OH_UPPER].stag = (rlt->tag) ?  tagt + tag1 : tagt;
    } else {
      hp[OH_UPPER].nbor = MPI_PROC_NULL;
      hp[OH_UPPER].stag = tagt;
    }
  }
  if (!ps)  nacc[1] = nacc[0];
 }
```

Finally, we *return* P_n^{send} through the argument pointer `nsendptr`.

```
  *nsendptr = nsend;
}
```

4.13.22 Macros `For_All_Grid_Z()`, `For_All_Grid_XY()`, `Grid_Exterior_Boundary()` and `Grid_Interior_Boundary()`

Here we show four macros used in the functions called from `make_send_sched()`.

`For_All_Grid_Z()`
`For_All_Grid_XY()`

The macro pair of `For_All_Grid_Z`$(p, x_0, y_0, z_0, x_1, y_1, z_1)$ and `For_All_Grid_XY`(p, x_0, y_0, x_1, y_1) constracts triply nested for-loops as `For_All_Grid()` does, but the outermost z-loop

and inner double xy-loops are constructed seperatedly so that we have some codes special to each z coordinate. The implementation of the macros is just a cut-and-paste of the body of `For_All_Grid()`; its first `For_Z()` is in the former and the remaining `For_Y()` and `for` construct are in the latter. The macro pair is used in `make_send_sched_body()` (the latter is through the macro `Make_Send_Sched_Body()`), `make_send_sched_self()`, `make_bsend_sched()`, `make_brecv_sched()`, `xfer_boundary_particles_v()` and `exchange_border_data_v()`.

```
#define For_All_Grid_Z(PS, X0, Y0, Z0, X1, Y1, Z1)\
  For_Z((fag_zidx=(Z0), fag_x1=GridDesc[PS].x+(X1),\
         fag_y1=GridDesc[PS].y+(Y1), fag_z1=GridDesc[PS].z+(Z1),\
         fag_w=GridDesc[PS].w, fag_dw=GridDesc[PS].dw,\
         fag_gz=Coord_To_Index(X0,Y0,Z0,fag_w,fag_dw)),\
        (fag_zidx<fag_z1), (fag_zidx++,fag_gz+=fag_dw))
#define For_All_Grid_XY(PS, X0, Y0, X1, Y1)\
  For_Y((fag_yidx=(Y0), fag_gy=fag_gz),\
        (fag_yidx<fag_y1), (fag_yidx++,fag_gy+=fag_w))\
    for (fag_xidx=(X0),fag_gx=fag_gy; fag_xidx<fag_x1; fag_xidx++,fag_gx++)
```

`Grid_Exterior_Boundary()`
`Grid_Interior_Boundary()`

`Grid_Exterior_Boundary`$(\nu_d, \delta_d(n'), x^l_d, x^u_d)$ and `Grid_Interior_Boundary`$(\nu_d, \delta_d(n'), x^l_d, x^u_d)$ give d-th dimensional lower (x^l_d) and upper (x^u_d) bounds of an exterior or interior region repectively of the local node n's primary/secondary subdomain whose d-th dimensional size is $\delta_d(n')$ where $n' = n$ or $n' = \mathit{parent}(n)$, shared with a neighbor whose d-th dimensional process coordinate relative to the subdomain is $\nu_d - 1 \in \{-1, 0, 1\}$. Note that the upper bound x^u_d is relative to the subdomain's upper bound $\delta_d(n')$. Also note that the former macro may gives the bounds of the subdomain itself rather than its exterior.

The lower and upper bounds x^l_d and $x^l_u + \delta_d(n')$ for an exterior resion are;

$$(x^l_d,\ x^u_d + \delta_d(n')) = \begin{cases} (-e^g,\ 0) & \nu_d - 1 = -1 \\ (0,\ \delta_d(n')) & \nu_d - 1 = 0 \\ (\delta_d(n'),\ \delta_d(n') + e^g) & \nu_d - 1 = 1 \end{cases}$$

while thier interior couterparts are;

$$(x^l_d,\ x^u_d + \delta_d(n')) = \begin{cases} (0,\ e^g) & \nu_d - 1 = -1 \\ (0,\ \delta_d(n')) & \nu_d - 1 = 0 \\ (\delta_d(n') - e^g,\ \delta_d(n')) & \nu_d - 1 = 1 \end{cases}$$

The macro `Grid_Exterior_Boundary()` is used in `make_send_sched_body()`, `make_brecv_sched()`, `xfer_boundary_particles_v()` and `exchange_border_data_v()`, while `Grid_Interior_Boundary()` is used in `make_bsend_sched()` and `exchange_border_data_v()`.

```
#define Grid_Exterior_Boundary(N, GS, PL, PU) {\
  const int e = OH_PGRID_EXT;\
  if (N==0)      { PL = -e;    PU = -(GS); }\
  else if (N==1) { PL = 0;     PU = 0; }\
  else           { PL = (GS);  PU = e; }\
}
#define Grid_Interior_Boundary(N, GS, PL, PU) {\
```

```
  const int e = OH_PGRID_EXT;\
  if (N==0)      { PL = 0;       PU = -(GS)+e; }\
  else if (N==1) { PL = 0;       PU = 0; }\
  else           { PL = (GS)-e;  PU = 0; }\
}
```

4.13.23 make_send_sched_body()

Make_Send_Sched_Body()

Prior to discussing the function `make_send_sched_body()`, we show a macro `Make_Send_Sched_Body`(μ) used solely in the function. This macro scans `NOfPGrid`$[p][s][g]$ for all $s \in [0, S)$ and $g \in \mathcal{S}_z = [x_l, x_u) \times [y_l, y_u) \times \{z\}$ in a xy-subplane at z in the exterior or interior specified by (x_l, y_l) and (x_u, y_u) of the local node n's primary ($p = 0$) or secondary ($p = 1$) subdomain, where p, z, x_l, x_u, y_l and y_u are given from the invoker function implicitly as its local variable `ps`, (invisible) `fag_zidx`, `xl`, `xu`, `yl` and `yu`. For each s and g, we let `NOfPGrid`$[p][s][g] = 0$ if the macro's argument μ is true to mean the xy-subplane is in n's subcuboid.

Otherwise, since all particles in the subplane are sent to a node m_f being the f-th member of a neighbor family, we accumulate the sum of `NOfPGrid`$[p][s][g]$ so that `NOfSend`$[p_f][s][m_f] =$ `NOfSend`$[(p_f S + s)N + m_f]$ has the number of particles to be sent to the node m_f as its primary ($p_f = 0$) or secondary($p_f = 1$) particles, where $p_f SN + m_f$ is given from the invoker function implicitly as its local variable `nofsbase`, and so that $P'_{\rm send}$, given implicitly as `nsend` too, has the total number of particles sent from n to nodes in the neighbor family. Then we let `NOfPGrid`$[p][s][g] = (p_f S + s)N + m_f + 1$ so that functions referring to it finds `NOfSend`$[p_f][s][m_f]$ quickly.

The reason why we define this macro is to avoid explicitly having two versions of similar codes for $\mu = 0$ and $\mu = 1$ in `make_send_sched_body()` and at the same time to avoid examining μ for each s and g. That is, the function just has two invocations of this macro with $\mu = 0$ and $\mu = 1$ expecting that the two versions are eventually produced by macro expansion and then unnecessary conditional construct examining μ for each s and g is eliminated by compilers because μ in the construct is a constant 0 or 1.

```
#define Make_Send_Sched_Body(MYSELF) {\
  int s, nofsidx=nofsbase;\
  for (s=0; s<ns; s++,nofsidx+=nn) {\
    dint *npg = NOfPGrid[ps][s];\
    int nsendofs=0;\
    For_All_Grid_XY(ps, xl, yl, xu, yu) {\
      const int g = The_Grid();\
      if (MYSELF)  npg[g] = 0;\
      else {\
        nsendofs += npg[g];  npg[g] = nofsidx + 1;\
      }\
    }\
    nsend += nsendofs;  NOfSend[nofsidx] += nsendofs;\
  }\
}
```

make_send_sched_body()

The function `make_send_sched_body()`, called solely from `make_send_sched()` but up to $2 \cdot 3^D$ times, scans a sub-block $\lambda =$ `rlist` of primary receiving/sending ($p =$ `ps` $= 0$)

or secondary receiving/sending ($p = 1$) block in `PrimaryCommList[][]` or `CommList[]`. The sub-block is for a neighbor subdomain $m =$ `sdid` having index $k =$ `n` of the local node n's primary ($p = 0$) or secondary ($p = 1$) subdomain $n^p = \{n, parent(n)\}[p]$. The function also scans the per-grid histogram elements in the exterior region of n^p being a part of its neighbor subdomain m, or in n^p itself, to find the node in which particles in each grid-voxel are accommodated and to return the number of particles in the region to be sent from n.

The function implements a part of what its level-4p counterpart shown in §4.10.22 does, but the implementation is quite *different* from the counterpart.

```
static int
make_send_sched_body(const int ps, const int n, const int sdid,
                     struct S_commlist *rlist) {
  const int me=myRank, ns=nOfSpecies, nn=nOfNodes;
  const int nx = n % 3, ny = n/3 % 3, nz = n/9;
  int xl, xu, yl, yu, zl, zu, zn;
  int rlz = rlist->region, rid, ridp=-1, ridn=-1, nofsbase;
  int nsend = 0;
  const int zmax = (SubDomains[sdid][OH_DIM_Z][OH_UPPER] -
                    SubDomains[sdid][OH_DIM_Z][OH_LOWER]) - 1;
  Decl_For_All_Grid();
```

At first, we determine the exterior or interior region of the subdomain n^p to be scanned, $\mathcal{S} = [x_l, x_u) \times [y_l, y_u) \times [z_l, z_u)$ for the neighbor m having index $k = \sum_{d=0}^{D-1} \nu_d 3^d$ invoking macro `Grid_Exterior_Boundary()` for each dimension $d \in [0, D)$. Then we skip `S_commlist` records in λ until we find the record whose `region` element $\zeta^u_{p_f}(m_f) - 1$ satisfies the following where m_f being its `rid` element and p_f is 0 or 1 according to its `tag` element being 0 or NS respectively.

$$\zeta^u_{p_f}(m_f) - 1 \geq z'_l = \begin{cases} \delta_z(m) - e^g & \nu_z - 1 = -1 \\ 0 & \nu_z - 1 \in \{0, 1\} \end{cases}$$

That is, if m is located *below* n^p, we start from the node m_f whose subcuboid contains m's top-side horizontal interior halo plane[95]. Otherwise, we simply start the node whose subcuboid is the lowest in the family of m. Note that such record for the particle in the xy-plane at z_l should be found because the last record has $\delta_z(m) - 1 \geq z'_l$.

```
  Grid_Exterior_Boundary(nx, GridDesc[ps].x, xl, xu);
  Grid_Exterior_Boundary(ny, GridDesc[ps].y, yl, yu);
  Grid_Exterior_Boundary(nz, GridDesc[ps].z, zl, zu);
  zn = (nz==0) ? zmax + 1 - OH_PGRID_EXT : 0;
  while (rlz<zn)  rlz = (++rlist)->region;
  rid = rlist->rid;  nofsbase = rlist->tag + rid;
```

Now we scan each xy-plane at $z \in [z_l, z_u)$ in the subdomain n^p and $z' = (z - z_l) + z'_l$ of the subdomain m invoking `Make_Send_Sched_Body()` with $\mu = 1$ if $k = \lfloor 3^D/2 \rfloor$ and $m_f = n$ to mean the plane is in n's subcuboid, or $\mu = 0$ otherwise. When $\mu = 1$, the macro lets `NOfPGrid`$[p][s][g] = 0$ for all $s \in [0, S)$ and g in the plane to mean that particles in grid-voxels stay in n.

[95] The node m_f is the last one if $e^g = 1$, but can be non-last when $e^g > 1$ and must be the one whose subcuboid is the lowest one among those having intersection with the planes.

When $\mu = 0$, on the other hand, the macro accumulates the sum of `NOfPGrid`$[p][s][g]$ for each s and for all g so that, at the end of this function, `NOfSend`$[p_f][s][m_f]$ has the number of particles of species s to be sent to m_f as its primary ($p_f = 0$) or secondary ($p_f = 1$) particles, and so that $P'_{\rm send}$ being the return value to the caller `make_send_sched()` has the total number of particles to be sent from n to the family members of m. The macro also lets `NOfPGrid`$[p][s][g] = (p_f S + s)N + m_f + 1$ after the accumulation so that functions referring to it quickly find the entry `NOfSend`$[p_f][s][m_f] =$ `NOfSend`$[(p_f S + s)N + m_f]$ when they move the particles in the grid-voxel to `SendBuf`[].

Then we visit the next `S_commlist` record for m_{f+1} if $z' = \zeta^u_{p_f}(m_f) - 1$ and $z' < \delta_z(m) - 1$, or in other words $z' + 1 > \zeta^u_{p_f}(m_f) - 1$ and $z' + 1 \le \delta_z(m) - 1$, updating p_f and m_f according to the record.

```
    For_All_Grid_Z(ps, xl, yl, zl, xu, yu, zu) {
      if (n==OH_NBR_SELF && rid==me) {
        Make_Send_Sched_Body(1);
      }
      else {
        Make_Send_Sched_Body(0);
      }
      if (++zn>rlz && zn<=zmax) {
        rlz = (++rlist)->region;  rid = rlist->rid;  nofsbase = rlist->tag + rid;
      }
    }
    return(nsend);
  }
```

4.13.24 `make_send_sched_self()`

`make_send_sched_self()` The function `make_send_sched_self()`, called solely from `make_send_sched()` but up to twice, scans primary receiving (`psor2` $= p' = p = 0$), secondary receiving ($p' = p = 1$) or alternative secondary receiving ($p' = 2$, $p = 1$) block in $\lambda =$ `rlist` being a part of `PrimaryCommList`[][] or `CommList`[], in order to have the lower/upper bound of the local node n's subcuboid in `ZBound`$[p][\beta] = \zeta^\beta_p(n')$ where $n' = \{n, n^p_{old}, n^p_{new}\}[p']$, and to have the schedule of halo particle transfer through the horizontal surface of the subcuboid in `HPlane`$[p][\beta]$. It also scans `NOfPGridTotal`$[p][s][g]$ for all $s \in [0, S)$ and g in n's subdomain including its exterior, so that `NOfPGridOut`$[p][s][g]$, `TotalPNext`$[p][s]$ and $Q = \{Q^n_n, Q_n\}[p]$ pointed by `naccptr` have appropriate values.

This function is very *level-4s own* and thus has no counterpart in level-4p.

```
  static void
  make_send_sched_self(const int psor2, struct S_commlist *rlist, int *naccptr) {
    const int me=myRank, nn=nOfNodes, ns=nOfSpecies;
    const int tag1 = OH_NEIGHBORS * ns;
    const int tagt = OH_NBR_TCC * ns,  tagb = OH_NBR_BCC * ns;
    const int ps = psor2==0 ? 0 : 1,  rtag = ps ? tag1 : 0;
    const int exti = OH_PGRID_EXT;
    const int zmax = GridDesc[psor2].z - 1;
    int rlz = -1, rid = nn, ridp = -1, ridn, stag = 0;
    struct S_hplane *hp = HPlane[ps];
    int *zb = ZBound[ps];
    int np = *naccptr,  *tpn = TotalPNext + (ps ? ns : 0);
```

```
int s;
Decl_For_All_Grid();
```

At first we initialize `HPlane`$[p][\beta]$`.nbor` = `MPI_PROC_NULL` for both $\beta \in \{0, 1\}$ in case n's subcuboid is empty and thus no halo particle transfer takes place. Then we scan each xy-plane at $z \in [-e^g, \delta_z(n') + e^g)$, with initial setting $m_{-1} = N$, $m_{-2} = -1$ and $\zeta^u_{p_{-1}}(m_{-1}) - 1 = -1$ to mean that bottom exterior of n's subdomain up to $z = -1$ is not assigned to any n's family members but to a family member of the subdomain below it if exist ($m_{-1} = N$) and we don't care about the second-top member of the bottom-side subdomain ($m_{-2} = -1$).

In the scanning loop body, we first examine if $z = \zeta^u_{p_f}(m_f) - 1$ ($f = -1$ at initial) to mean we reach to the top surface of m_f's subcuboid. If so, we let m_{f+1} be `rid` element of the record we will visit next if $z < \delta_z(n') - 1$, or N to mean the subcuboid above the surface is for a node in the family for the subdomain above n's one. Otherwise, we let $m_{f+1} = -1$ to mean we don't yet care about the subcuboid above that we are now visiting.

Next we examine $m_{f-1} = n$ or $m_{f+1} = n$. If the former holds to mean we are now visiting the bottom surface of a subcuboid of m_f just above n's subcuboid, we let `ZBound`$[p][1] = \zeta^u_p(n) = z$ and `HPlane`$[p][1]$`.nbor` $= m_f$ to mean the particles in the upper horizontal halo planes are exchanged with m_f. We also let `stag` and `rtag` elements of `HPlane`$[p][1]$ be $(p_f \cdot 3^D + (2 \cdot 3^2 + 3^1 + 3^0))S$ and $(p \cdot 3^D + (3^1 + 3^0))S$ so that the sending/receiving communmication for species s is associated with a unique point in a conceptual 5-dimensional space of $[2][3][3][3][S]$ being $[p_f][2][1][1][s]$ for the former and $[p][0][1][1][s]$ for latter respectively, where p_f is 0 or 1 according to `tag` element of currently visiting record being 0 or NS respectively, or 0 if $m_f = N$ to mean the current xy-plane is in upper exterior of n's subdomain.

On the other hand, if $m_{f+1} = n$ holds to mean we are now visiting the top surface of a subcuboid of m_f just below n's subcuboid, we let `ZBound`$[p][0] = \zeta^l_p(n) = z + 1$ and `HPlane`$[p][0]$`.nbor` $= m_f$ to mean the particles in the lower horizontal halo plane are exchanged with m_f. We also let `stag` and `rtag` elements of `HPlane`$[p][0]$ be $(p_f \cdot 3^D + (2 \cdot 3^2 + 3^1 + 3^0))S$ and $(p \cdot 3^D + (3^1 + 3^0))S$ to associate communicatetions for s with $[p_f][0][1][1][s]$ and $[p][2][1][1][s]$ respectively. Note that $p_f = 0$ if $m_f = N$ too but this means the current xy-plane is in lower exterior of n's subdomain.

```
hp[OH_LOWER].nbor = hp[OH_UPPER].nbor = MPI_PROC_NULL;
For_All_Grid_Z(psor2, -exti, -exti, -exti, exti, exti, exti) {
  const int z = Grid_Z();
  ridn = (z==rlz) ? (z<zmax ? rlist->rid : nn) : -1;
  if (ridp==me) {
    zb[OH_UPPER] = z;  hp[OH_UPPER].nbor = rid;
    hp[OH_UPPER].stag = stag + tagt;
    hp[OH_UPPER].rtag = rtag + tagb;
  } else if (ridn==me) {
    zb[OH_LOWER] = z + 1;  hp[OH_LOWER].nbor = rid;
    hp[OH_LOWER].stag = stag + tagb;
    hp[OH_LOWER].rtag = rtag + tagt;
  }
```

Then we examine if $m_f = n$ to mean we are visiting a xy-plane in n's subcuboid. If so we further examine if $m_{f-1} \geq 0$ or $m_{f+1} \geq 0$ to mean the plane is the bottom or top surface of the subcuboid and, if either of them holds, call `make_send_sched_hplane()` giving p', z, the pointer to Q, and the element arrays `nsend` and `sbuf` of `HPlane`$[p][\beta]$ where $\beta = 0$

for the former and $\beta = 1$ for the latter. By this call, besides it lets $\mathcal{P}_O(p,s,g) = \mathcal{P}_T(p,s,g)$ for all $s \in [0,S)$ and $g \in \mathcal{S}_z$ where

$$\mathcal{S}_z = [-e^g,\, \delta_x(n')+e^g) \times [-e^g,\, \delta_y(n')+e^g) \times \{z\}$$

we have the followings.

$$\begin{aligned}
\texttt{nsend}[s] = \mathcal{P}'_Z(p,s,z) &= \sum_{g \in \mathcal{S}_z} \mathcal{P}_T(p,s,g) \\
z_0 &= \zeta^l_p(n') - 1 \\
\texttt{TotalPNext}[p][s] = \mathcal{Q}(p,s,z) &= \sum_{z'=z_0}^{z} \mathcal{P}'_Z(p,s,z') \\
\texttt{sbuf}[s] &= \mathcal{Q}(p,s,z-1) \\
Q &= \{0, Q^n_n\}[p] + \sum_{s=0}^{S-1} \mathcal{Q}(p,s,z)
\end{aligned}$$

Note that $\texttt{sbuf}[s]$ above is so far the offset to the head of $hbuf^s_h(p,\beta,s)$ from the head of $pbuf(p,s)$ in $\texttt{SendBuf}[]$.

Also note that $m_{f-1} \geq 0$ and $m_{f+1} \geq 0$ may hold at the same time if n's subcuboid is one-grid thick. Even if so, we call `make_send_sched_hplane()` just once for $\beta = 0$ and then copy all elements in `nsend` and `sbuf` elements of $\texttt{HPlane}[p][0]$ into those of $\texttt{HPlane}[p][1]$ because the amount of particles to be sent to m_{f-1} and m_{f+1} and their locations in the particle buffer are equivalent.

If neither $m_{f-1} \geq 0$ nor $m_{f+1} \geq 0$ holds, on the other hand, we call `make_send_sched_hplane()` too but giving `NULL` instead of `nsend` and `sbuf` just for updating $\texttt{NOfPGridOut}[p][][]$, $\texttt{TotalPNext}[p][]$ and Q.

```
if (rid==me) {
  if (ridp>=0) {
    make_send_sched_hplane(psor2, z, naccptr,
                           hp[OH_LOWER].nsend, hp[OH_LOWER].sbuf);
    if (ridn>=0) {
      for (s=0; s<ns; s++) {
        hp[OH_UPPER].nsend[s] = hp[OH_LOWER].nsend[s];
        hp[OH_UPPER].sbuf[s] = hp[OH_LOWER].sbuf[s];
      }
    }
  }
  else if (ridn>=0)
    make_send_sched_hplane(psor2, z, naccptr,
                           hp[OH_UPPER].nsend, hp[OH_UPPER].sbuf);
  else
    make_send_sched_hplane(psor2, z, naccptr, NULL, NULL);
```

If $m_f \neq n$, again we examine if $m_{f-1} = n$ or $m_{f+1} = n$ to mean the xy-plane we are visiting is just above or below the subcuboid of n and thus upper or lower horizontal exterior halo plane. If either of them holds, we call `make_send_sched_hplane()` to have $\mathcal{P}'_Z(p,s,z)$ in $\texttt{nrecv}[\text{s}]$ and $\mathcal{Q}(p,s,z-1)$ in $\texttt{rbuf}[\text{s}]$ of $\texttt{HPlane}[p][\beta]$ where $\beta = 1$ if $m_{f-1} = n$ and $\beta = 0$ $m_{f+1} = n$, because n receives halo particles into $hbuf^r_h(p,\beta,s)$ and thus they are accommodated by n.

If neither $m_{f-1} = n$ nor $m_{f+1} = n$ holds, on the other hand, we simply let `NOfPGridOut`$[p][s][g] = 0$ for all $s \in [0, S)$ and g in the plane, because n does not have any particles in the plane.

```
    } else {
      if (ridp==me)
        make_send_sched_hplane(psor2, z, naccptr,
                               hp[OH_UPPER].nrecv, hp[OH_UPPER].rbuf);
      else if (ridn==me)
        make_send_sched_hplane(psor2, z, naccptr,
                               hp[OH_LOWER].nrecv, hp[OH_LOWER].rbuf);
      else {
        for (s=0; s<ns; s++) {
          int *npgo=NOfPGridOut[ps][s];
          For_All_Grid_XY(psor2, -exti, -exti, exti, exti)
            npgo[The_Grid()] = 0;
        }
      }
    }
```

At the end of the scanning loop body, we examine if $z = \zeta^u_{p_f}(m_f) - 1$ again and, if so to mean the next xy-plane is for the bottom of m_{f+1}'s subcuboid, we let $m_{f-1} = m_f$, and step to the next f letting m_f be `rid` element of the record in λ and p_f be 0 or 1 according to its `tag` element being 0 or NS respectively if $z < \delta_z(n') - 1$, or let $m_f = N$ and $p_f = 0$ otherwise. If $z \neq \zeta^u_{p_f}(m_f) - 1$, on the other hand, the next xy-plane is still in m_f's subcuboid and thus we let $m_{f-1} = -1$.

```
    ridp = -1;
    if (z==rlz) {
      ridp = rid;
      if (z<zmax) {
        rlz = rlist->region;  rid = rlist->rid;
        stag = rlist->tag ? tag1 : 0;  rlist++;
      } else {
        rlz++;  rid = nn;  stag = 0;
      }
    }
  }
```

After the scanning loop, `HPlane`$[p][\beta]$`.{sbuf,rbuf}`$[s]$ has the offset from the head of $pbuf(p, s)$ whose index in `SendBuf`[] is $Q^0 + \sum_{t=0}^{s-1}$ `TotalPNext`$[p][s]$ where $Q^0 = \{0, Q^n_n\}[p]$ and is Q at the begining of this function. Therefore, we calculate the index of $pbuf(p, s)$ for each s and add it to `HPlane`$[p][\beta]$`.`$b[s]$ for each $\beta \in \{0, 1\}$ and $b \in$ `{sbuf,rbuf}` so that they have the indices of `SendBuf`[] for the send/receive buffers, $hbuf^s_h(p, \beta, s)$ and $hbuf^r_h(p, \beta, s)$ respectively.

```
  for (s=0; s<ns; s++) {
    hp[OH_LOWER].sbuf[s] += np;  hp[OH_LOWER].rbuf[s] += np;
    hp[OH_UPPER].sbuf[s] += np;  hp[OH_UPPER].rbuf[s] += np;
    np += tpn[s];
  }
}
```

4.13.25 make_send_sched_hplane()

For_All_Grid_XY_At_Z() Prior to discussing make_send_sched_hplane(), we show the macro For_All_Grid_XY_At_Z $(p, x_0, y_0, x_1, y_1, z)$ solely used in the function. This macro is a relative of For_All_Grid_XY() but the z-coordinate value of the xy-plane to be scanned is given explicitly by z. Therefore, this macro is equivalent to For_All_Grid$(p, x_0, y_0, z, x_1, y_1, z')$ where $z' = z + 1 - \delta_z(\{n, parent(n)\}[p])$ for the local node n.

```
#define For_All_Grid_XY_At_Z(PS, X0, Y0, X1, Y1, Z0)\
  For_Z((fag_zidx=(Z0), fag_x1=GridDesc[PS].x+(X1),\
         fag_y1=GridDesc[PS].y+(Y1), fag_z1=(Z0)+1,\
         fag_w=GridDesc[PS].w, fag_dw=GridDesc[PS].dw,\
         fag_gz=Coord_To_Index(X0,Y0,Z0,fag_w,fag_dw)),\
        (fag_zidx<fag_z1), (fag_zidx++,fag_gz+=fag_dw))\
    For_Y((fag_yidx=(Y0), fag_gy=fag_gz),\
          (fag_yidx<fag_y1), (fag_yidx++,fag_gy+=fag_w))\
      for (fag_xidx=(X0),fag_gx=fag_gy; fag_xidx<fag_x1; fag_xidx++,fag_gx++)
```

make_send_sched_hplane() The function make_send_sched_hplane(), called solely from make_send_sched_self() but possibly for each xy-plane in the local node n's subdomain and its exterior, scans $\mathcal{P}_T(p, s, g) =$ NOfPGridTotal$[p][s][g]$ for all $s \in [0, S)$ and

$$g \in \mathcal{S}_z = [-e^g,\, \delta_x(n')+e^g) \times [-e^g,\, \delta_y(n')+e^g) \times \{z = \mathtt{z}\}$$

where $p = \{0, 1, 1\}[p' =$ psor2$]$ and $n' = \{n, n^p_{old}, n^p_{new}\}[p']$, and copies each of them into $\mathcal{P}_O(p, s, g) =$ NOfPGridOut$[p][s][g]$ because z is in the subcuboid of n' or its horizontal exterior halo planes. It also calculate $\mathcal{P}'_Z(p, s, z) = \sum_{g \in \mathcal{S}_z} \mathcal{P}_T(p, s, g)$ for each s to let HPlane$[p][\beta]$.{nsend, nrecv}$[s]$ be the sum if the argument np is not NULL and thus points it. The sum is also added to $\mathcal{Q}(p, s, z)$ being the so-far value of TotalPNext$[p][s]$ and to Q being Q^n_n or Q_n pointed by naccptr. In addition, if the argument buf is not NULL and thus points HPlane$[p][\beta]$.{sbuf, rbuf}$[s]$, it is let be $\mathcal{Q}(p, s, z-1)$ or in other words TotalPNext$[p][s]$ before the addition and thus have the offset from $pbuf(p, s)$ to the send/receive buffer of halo particles, i.e., $hbuf^s_h(p, \beta, s)$ or $hbuf^r_h(p, \beta, s)$ respectively.

This function is very *level-4s's own* and thus has no counterpart in level-4p.

```
static void
make_send_sched_hplane(const int psor2, const int z, int *naccptr,
                       int *np, int *buf) {
  const int ns=nOfSpecies, exti=OH_PGRID_EXT;
  const int ps = psor2==0 ? 0 : 1,  nsor0 = ps ? ns : 0;
  int nacc = *naccptr, s;
  Decl_For_All_Grid();

  for (s=0; s<ns; s++) {
    dint *npgt = NOfPGridTotal[ps][s];
    int *npgo = NOfPGridOut[ps][s];
    int npofs = 0;
    if (buf)  buf[s] = TotalPNext[nsor0+s];
    For_All_Grid_XY_At_Z(psor2, -exti, -exti, exti, exti, z) {
      const int g = The_Grid();
      npofs += (npgo[g] = npgt[g]);
```

```
    }
    nacc += npofs;  TotalPNext[nsor0+s] += npofs;
    if (np)  np[s] = npofs;
  }
  *naccptr = nacc;
}
```

4.13.26 update_descriptors()

update_descriptors()

The function update_descriptors(), called from exchange_particles4s() when we had anywhere accommodation and from make_recv_list() otherwise, reinitializes BorderExc[][1][][].{send, recv} for the old secondary subdomain given through the argument n^p_{old} = oldp by clear_border_exchange(), and update FieldDesc[].{bc,red}.size[1] for the new secondary subdomain given through the argument n^p_{new} = newp by calling set_field_descriptors() giving it arrays FieldTypes[][and SubDomains[m][][]. It also calls adjust_field_descriptor() to modify FieldDesc[$F-1$].{bc,red}.size[1] for per-grid histograms giving it ps = 1.

Note that we do above if the old and new parents are different, and call clear_border_exchange() if the old one exists, while other two functions are called if the new one exists. Also note that this function is perfectly equivalent to its level-4p counterpart shown in §4.10.29

```
static void
update_descriptors(const int oldp, const int newp) {
  int n;

  if (oldp!=newp) {
    if (oldp>=0)  clear_border_exchange();
    if (newp>=0) {
      set_field_descriptors(FieldTypes, SubDomains[newp], 1);
      adjust_field_descriptor(1);
    }
  }
}
```

4.13.27 update_neighbors()

Neighbor_Grid_Offset()

The macro Neighbor_Grid_Offset(p, ν_d-1, m, d, c), used three times in the function update_neighbors() solely as discussed later in this section, calculates

$$x^0_d(m, n^p) = \delta^l_d(m) - \delta^l_d(n^p) = \begin{cases} \delta^l_d(m) - \delta^u_d(m) = -\delta_d(m) & \nu_d = 0 \\ \delta^l_d(n^p) - \delta^l_d(n^p) = 0 & \nu_d = 1 \\ \delta^u_d(n^p) - \delta^l_d(n^p) = \delta_d(n^p) & \nu_d = 2 \end{cases}$$

where $n^p = \{n, parent(n)\}[p]$ for the local node n, to update GridOffset[p][k] where $k = \sum_{d=0}^{D-1} \nu_d 3^d$, as discussed in §4.9.5 and in §4.10.30 for the level-4p counterpart perfectly equivalent to this macro.

```
#define Neighbor_Grid_Offset(PS, N, SD, D, XYZ)\
  (N==0 ? 0 : (N<0 ? SubDomains[SD][D][OH_LOWER]-SubDomains[SD][D][OH_UPPER] :\
                     GridDesc[ps].XYZ))
```

update_neighbors()

The function `update_neighbors()` is called from `init4s()` with $p = \texttt{ps} = 0$, and from `rebalance4s()` or `exchange_particles4s()` with $p = 1$ when we had normal or anywhere accommodation respectively. The function initializes/updates `AbsNeighbors`$[p][k]$ for all $k \in [0, 3^D)$ to let it have;

$$\texttt{AbsNeighbors}[p][k] = m_k = \begin{cases} \texttt{Neighbors}[p][k] & \texttt{Neighbors}[p][k] \geq 0 \\ -(\texttt{Neighbors}[p][k] + 1) & \texttt{Neighbors}[p][k] < 0 \end{cases}$$

and lets `GridOffset`$[p][k] = gidx(x_0^0(m_k, n^p), \ldots)$ where $n^p = \{n, parent(n)\}[p]$ and $x_d^0(m_k, n^p)$ is given by `Neighbor_Grid_Offset()`, as discussed in §4.9.5 and in §4.10.30 for the level-4p coutnerpart of the function. However in addition to those initializations, the function has *level-4s's own* ones for `PrimaryCommList`$[p][k']$ where $k' = 3^D - 1 - k$ for each k to let its elements `rid` $= m_k$, `tag` $= 0$, and `region` $= \delta_z(m_k) - 1$[96]. Another small *difference* from the counterpart is in the outermost two loops whose coding is simplified exploiting the fact $D = 3$.

```
static void
update_neighbors(const int ps) {
  int n, nx, ny, nz;
  const int nn = nOfNodes;
  struct S_commlist *cl = PrimaryCommList[ps];

  for (nz=-1,n=0; nz<2; nz++) {
    for (ny=-1; ny<2; ny++) {
      for (nx=-1; nx<2; nx++,n++) {
        int nbr = Neighbors[ps][n];
        const int nrev = OH_NEIGHBORS - 1 - n;
        nbr = AbsNeighbors[ps][n] = nbr<0 ? -(nbr+1) : nbr;
        cl[nrev].rid = nbr;  cl[nrev].tag = cl[nrev].sid = cl[nrev].count = 0;
        if (nbr>=nn) {
          GridOffset[ps][n] = 0;  cl[nrev].region = 0;
        } else {
          GridOffset[ps][n] =
            Coord_To_Index(Neighbor_Grid_Offset(ps, nx, nbr, OH_DIM_X, x),
                           Neighbor_Grid_Offset(ps, ny, nbr, OH_DIM_Y, y),
                           Neighbor_Grid_Offset(ps, nz, nbr, OH_DIM_Z, z),
                           GridDesc[0].w, GridDesc[0].dw);
          cl[nrev].region = SubDomains[nbr][OH_DIM_Z][OH_UPPER] -
                            SubDomains[nbr][OH_DIM_Z][OH_LOWER] - 1;
        }
      }
    }
  }
}
```

4.13.28 `set_grid_descriptor()`

set_grid_descriptor()

The function `set_grid_descriptor()` is called from `init4s()` with `idx` $= i = 0$ and `nid` $= m = n$ arguments for the local node n for the initialization, from `rebalance4s()` or

[96] We also let unused elements `sid` and `count` be 0 to avoid leaving them undefined.

`exchange_particles4s()` with $i = 1$ and $m = parent(n)$ when we had normal or anywhere accommodation respectively, and from `make_recv_list()` with $i = 2$ and $m = parent(n)$, when helpand-helper reconfiguration is taking place assigning m to the local node as its secondary subdomain. The function lets `GridDesc`$[i]$ have the shape information of the per-grid histogram for the subdomain m. Note that `GridDesc`[2] is used to have the shape information of *new* $parent(n)$ due to helpand-helper reconfiguration while [1] keeps that of *old* $parent(n)$.

The function is very similar to its level-4p counterpart shown in §4.10.31, but is *different* from it because the per-grid histogram has 3-grid thick exterior, one for sending and two for receiving as discussed in §4.12.3. Therefore, the elements `w`, `d` and `h` of `GridDesc`$[i]$ is let be $\delta_d^{\max} + 6e^g = \texttt{Grid}[d]\texttt{.size} + 6 \cdot \texttt{OH\_PGRID\_EXT}$ with $d = 0$, 1 and 2 respectively for the physical array size, and `dw` be `d` $\times$ `w`, if $D = 3$. Similarly, the elements of `x`, `y` and `z` for non-existent secondary subdomain are let be $-6e^g$.

```
static void
set_grid_descriptor(const int idx, const int nid) {
  const int exti6 = OH_PGRID_EXT*6;
  const int w = GridDesc[idx].w = Grid[OH_DIM_X].size+(exti6);
  const int d = GridDesc[idx].d =
                If_Dim(OH_DIM_Y, Grid[OH_DIM_Y].size+(exti6), 1);

  GridDesc[idx].h = If_Dim(OH_DIM_Z, Grid[OH_DIM_Z].size+(exti6), 1);
  GridDesc[idx].dw = d * w;
  if (nid>=0) {
    GridDesc[idx].x = SubDomains[nid][OH_DIM_X][OH_UPPER] -
                      SubDomains[nid][OH_DIM_X][OH_LOWER];
    GridDesc[idx].y = If_Dim(OH_DIM_Y,
                             SubDomains[nid][OH_DIM_Y][OH_UPPER] -
                             SubDomains[nid][OH_DIM_Y][OH_LOWER], 0);
    GridDesc[idx].z = If_Dim(OH_DIM_Z,
                             SubDomains[nid][OH_DIM_Z][OH_UPPER] -
                             SubDomains[nid][OH_DIM_Z][OH_LOWER], 0);
  } else {
    GridDesc[idx].x = GridDesc[idx].y = GridDesc[idx].z = -exti6;
    /* to ensure, e.g., x+3*(OH_PGRID_EXT)<=-3*(OH_PGRID_EXT) */
  }
}
```

4.13.29 `adjust_field_descriptor()`

`adjust_field_descriptor()` The function `adjust_field_descriptor()` is called from `init4s()` with argument `ps` $= p = 0$ for the initialization of the local node n's primary subdomain, and from `update_descriptors()` with $p = 1$ for n's secondary subdomain newly assigned to it by helpand-helper reconfiguration. The function is perfectly equivalent to its level-4p counterpart and thus modifies `FieldDesc`$[F-1]\texttt{.}\{\texttt{bc},\texttt{red}\}\texttt{.size}[p]$ as discussed in §4.10.32.

```
static void
adjust_field_descriptor(const int ps) {
  const int f = nOfFields - 1,  ns = nOfSpecies;
  int d, fs;
```

```
  for (d=0,fs=1; d<OH_DIMENSION; d++)  fs *= FieldDesc[f].size[d];
  fs *= ns-1;
  FieldDesc[f].bc.size[ps] += fs;    FieldDesc[f].red.size[ps] += fs;
}
```

4.13.30 update_real_neighbors()

update_real_neighbors()

The function update_real_neighbors(), called from init4s(), try_primary4s(), exchange_particles4s() and make_recv_list(), updates RealDstNeighbors[][] and RealSrcNeighbors[][] according to its mode argument to specify the elements to be updated, dosec to specify whether nodes have helpers or not, and oldp and newp being old and new *parent*(n) of the local node n on helpand-helper reconfiguration.

This function is perfectly equivalent to its level-4p counterpart shown in §4.10.33

```
static void
update_real_neighbors(const int mode, const int dosec, const int oldp,
                      const int newp) {
  const int me=myRank, nn=nOfNodes, nn4=nn<<2;
  const int dosec0 = mode != URN_PRI;
  int i, nbridx, ps, *doccur[2], *soccur[2];

  for (i=0; i<nn4; i++)  TempArray[i] = 0;
  doccur[0] = TempArray;       doccur[1] = doccur[0] + nn;
  soccur[0] = doccur[1] + nn;  soccur[1] = soccur[0] + nn;

  if (mode==URN_TRN) {
    int *tmp = RealSrcNeighbors[1][0].nbor;
    RealSrcNeighbors[1][0].n = RealSrcNeighbors[0][0].n;
    RealSrcNeighbors[1][0].nbor = RealSrcNeighbors[0][0].nbor;
    RealSrcNeighbors[0][0].nbor = tmp;
  }
  RealDstNeighbors[0][0].n = RealDstNeighbors[0][1].n = 0;
  RealSrcNeighbors[0][0].n = RealSrcNeighbors[0][1].n = 0;
  upd_real_nbr(me, 0, 1, 0, dosec0, Nodes, RealDstNeighbors[0], doccur);
  upd_real_nbr(me, 0, 0, 0, dosec0, Nodes, RealSrcNeighbors[0], soccur);
  if (mode==URN_PRI)  return;

  nbridx = mode==URN_TRN ? 2 : 1;
  upd_real_nbr(newp, 0, 1, nbridx, 1, Nodes, RealDstNeighbors[0], doccur);
  upd_real_nbr(newp, 1, 1, nbridx, 1, Nodes, RealSrcNeighbors[0], soccur);
  if (mode!=URN_TRN)  return;

  for (ps=0; ps<2; ps++) {
    const int nd = RealDstNeighbors[0][ps].n;
    const int ns = RealSrcNeighbors[0][ps].n;
    for (i=0; i<nd; i++)  doccur[ps][RealDstNeighbors[0][ps].nbor[i]] = 0;
    for (i=0; i<ns; i++)  soccur[ps][RealSrcNeighbors[0][ps].nbor[i]] = 0;
  }
  RealDstNeighbors[1][0].n = RealDstNeighbors[1][1].n = 0;
  RealSrcNeighbors[1][1].n = 0;
  upd_real_nbr(me,   0, 1, 0, 1,     Nodes,     RealDstNeighbors[1], doccur);
  upd_real_nbr(oldp, 0, 1, 1, 1,     Nodes,     RealDstNeighbors[1], doccur);
  upd_real_nbr(newp, 1, 1, 2, dosec, NodesNext, RealSrcNeighbors[1], soccur);
```

```
}
```

4.13.31 upd_real_nbr()

upd_real_nbr() The function upd_real_nbr(), called solely from update_real_neighbors() but up to seven times, to add members to RealDstNeighbors[][] or RealSrcNeighbors[][]. The function is perfectly equivalent to its level-4p counterpart shown in §4.10.34.

```
static void
upd_real_nbr(const int root, const int psp, const int pss,
             const int nbr, const int dosec, struct S_node *nodes,
             struct S_realneighbor rnbrptr[2], int *occur[2]) {
  const int me=myRank;
  struct S_realneighbor *pnbr = rnbrptr+psp,  *snbr = rnbrptr+pss;
  int *poccur = occur[psp],  *soccur = occur[pss];
  int i;

  if (root<0)  return;
  if (root!=me && !poccur[root]) {
    pnbr->nbor[pnbr->n++] = root;  poccur[root] = 1;
  }
  if (dosec) {
    struct S_node *ch;
    for (ch=nodes[root].child; ch; ch=ch->sibling) {
      const int nid = ch->id;
      if (nid!=me && !soccur[nid]) {
        snbr->nbor[snbr->n++] = nid;  soccur[nid] = 1;
      }
    }
  }
  for (i=0; i<OH_NEIGHBORS; i++) {
    const int nid = Neighbors[nbr][i];
    struct S_node *ch;
    if (nid<0 || nid==root)  continue;
    if (!poccur[nid]) {
      pnbr->nbor[pnbr->n++] = nid;  poccur[nid] = 1;
    }
    if (dosec) {
      for (ch=nodes[nid].child; ch; ch=ch->sibling) {
        const int cid = ch->id;
        if (!soccur[cid]) {
          snbr->nbor[snbr->n++] = cid;  soccur[cid] = 1;
        }
      }
    }
  }
}
```

4.13.32 exchange_xfer_amount()

exchange_xfer_amount() The function exchange_xfer_amount(), called solely from exchange_particles4s(), exchanges NOfSend[][][] in the nodes responsible of a subdomain and its neighbors as the

nodes' primary/secondary subdomain to have `NOfRecv[][][]` for position-aware particle transfer. The function is very similar to its level-4p counterpart shown in §4.10.35, but slightly *different* from it because this function is called not only when we will be in secondary mode but also for primary mode. Therefore, the function is given an argument p'_n = `nextmode` to indicate we will be in primary ($p'_n = 0$) or secondary ($p'_n = 1$) mode in addition to two arguments equivalent to the counterpart; `trans` $= t \in \{0, 1\}$ to mean we have stable ($t = 0$) or transitional ($t = 1$) state of helpand-helper configuration; and `psnew` $= p_n \in \{0, 1\}$ being 1 iff the local node will have a secondary subdomain and thus will receive some particles.

```
static void
exchange_xfer_amount(const int trans, const int psnew, const int nextmode) {
  const struct S_realneighbor *snbr = RealSrcNeighbors[trans];
  const struct S_realneighbor *dnbr = RealDstNeighbors[trans];
  const int nnns = nOfNodes * nOfSpecies;
  int ps, tag, req;
```

The first part of this function is perfectly equivalent to that in the level-4p counterpart, and thus posts `MPI_Irecv()` to receive `NOfRecv`$[p][s][m]$ from all nodes $m \in$ `RealSrcNeighbors`$[t][p]$`.nbor[]` for $p \in \{0, p_n\}$.

```
  for (ps=0,tag=0,req=0; ps<=psnew; ps++,tag+=nnns) {
    const int n = snbr[ps].n;
    const int *nbor = snbr[ps].nbor;
    int i,  *nrbase = NOfRecv + tag;
    for (i=0; i<n; i++,req++) {
      const int nid = nbor[i];
      MPI_Irecv(nrbase+nid, 1, T_Hgramhalf, nid, tag, MCW, Requests+req);
    }
  }
```

The next part is slightly *different* from that of the counterpart because it sends local node n's `NOfSend`$[p][s][m]$ to all nodes $m \in$ `RealDstNeighbors`$[t][p]$`.nbor[]` for $p \in \{0, p'_n\}$, instead of $p \in \{0, 1\}$, by `MPI_Isend()`.

```
  for (ps=0,tag=0; ps<=nextmode; ps++,tag+=nnns) {
    const int n = dnbr[ps].n;
    const int *nbor = dnbr[ps].nbor;
    int i,  *nsbase = NOfSend + tag;
    for (i=0; i<n; i++,req++) {
      const int nid = nbor[i];
      MPI_Isend(nsbase+nid, 1, T_Hgramhalf, nid, tag, MCW, Requests+req);
    }
  }
```

The last part is perfectly equivalent to that of the counterpart again, and thus confirms the completion of all `MPI_Irecv()` and `MPI_Isend()` calls recorded in `Requests[]` by `MPI_Waitall()` to have their completion status in `Statuses[]` (but not referring to).

```
  MPI_Waitall(req, Requests, Statuses);
}
```

4.13.33 make_bxfer_sched()

make_bxfer_sched() The function make_bxfer_sched(), called solely from exchange_particles4s(), scans primary receiving/sending and secondary receiving/sending blocks in PrimaryCommList[][] if we will be in primary mode, or those in CommList[] possibly together with alternative secondary receiving/sending blocks otherwise, to build the halo particle transfer schedule for those in vertical halo planes. The function is given the following arguments; trans $= t \in \{0,1\}$ is 1 iff we have helpand-helper reconfiguration with normal accommodation; psnew $= p_n \in \{0,1\}$ is 1 iff the local node will have helpand in the next step; the pointer pair to the head of primary receiving and secondary receiving blocks rlist[2] $= \lambda[2]$ being {PrimaryCommList[0], PrimaryCommList[1]} or {CommList, SecRList}; and the pair of index arrays rlidx[2] $= \chi[2]$ for primary receiving/sending and secondary receiving/sending blocks being {PrimaryRLIndex[], PrimaryRLIndex[]} or {RLIndex[], SecRLIndex[]}.

This function is very *level-4s own* and thus has no counterpart in level-4p.

```
static void
make_bxfer_sched(const int trans, const int psnew, struct S_commlist *rlist[2],
                 int *rlidx[2]) {
  const int nn = nOfNodes;
  int d, ps, du;
  int (*vph)[2][2] = (int(*)[2][2])VPlaneHead;
  int nsendib=0, nrecveb=0, vpidx=0;
```

This function calls make_bsend_sched() for sending schedule and make_brecv_sched() for receiving schedule up to eight times for each in a triply nested loop for all $d \in \{0,1\}$, $p \in \{0, p_n\}$ and $\beta \in \{0,1\}$ in this order, to have the transfer schedule for j-th node responsible of contacting subcuboid in a subdomain being lower ($\beta = 0$) or upper ($\beta = 1$) neighbor of the local node n's primary ($p = 0$) or secondary ($p = 1$) subdomain along x ($d = 0$) or y ($d = 1$) axis in VPlane[$i(d,p,\beta,j)$]. The functions are called if the local node has primary/secondary subcuboid and the neighbor exists, and commonly given the following arguments.

- psor2 $= p' = 2$ if $p = 1$ and $t = 1$, or $p' = p$ otherwise.
- nx $= \nu_x = \{2\beta, 1\}[d]$, ny $= \nu_y = \{1, 2\beta\}[d]$, and n $= k = 3^2 + \nu_y \cdot 3^1 + \nu_x \cdot 3^0$ for neighbor index.
- rlist $= \lambda[p][\chi[p][k']]$ if $p' \neq 2$, or AltSecRList[AltSecRLIndex[k']] otherwise, where $k' = 3^D - k - 1$.

The function make_bsend_sched() is also given pointers nsendptr and vpptr pointing local variables to let it know the index H_s of BoundarySendBuf[] for $hbuf_v^s(d,p,\beta,m_0)$ and the index V of VPlane[] being $i(d,p,\beta,0)$, and to let it add the total size of the buffers and the number of contacting nodes in the neighbor family to the variables. Similarly, make_brecv_sched() is given a pointer nrecvptr pointing a local varible of Particles[]'s index H_r for $hbuf_v^r(d,p,\beta,m_0)$ and for the addition of total size to it, but the other argument vpidx simply carries $i(d,p,\beta,0)$.

In addition to build the schedules in VPlane[], make_bsend_sched() lets NOfPGrid[][][] for grid-voxels in vertical interior halo planes have negative values representing indices of BoundarySendBuf[], while make_brecv_sched() lets array elements for grid-voxels in exterior pillars have values greater than 2^{32} for indices of the buffer.

In addition to calling two functions, this function lets `VPlaneHead`$[d][p][\beta] = i(d, p, \beta, 0)$ for all $d \in \{0, 1\}$, $p \in \{0, 1\}$ and $\beta \in \{0, 1\}$, as well as its last element $[2][0][0]$. This assignment is done even for p such that $p > p_n$ or `ZBound`$[p][1] = 0$ to mean inexistent subcuboid, as well as for inexistent neighbors. Therefore, we can keep `xfer_boundary_particles_v()` and `exchange_border_data_v()` from scanning grid-voxels in a vertical exterior halo plane for copying particles from $hbuf_v^r(d, p, \beta, m)$ for efficiency, and keep the latter from scanning those in vertical interior halo plane for copying to $hbuf_v^s(p, d, \beta, m)$ for logical correctness, because these functions recognizes that the particle copying is unnecessary/inhibitive by the fact `VPlaneHead`$[j]$ = `VPlaneHead`$[j + 1]$ where $j = 4d + 2p + \beta$ for $[d][p][\beta]$.

```
    for (d=OH_DIM_X; d<=OH_DIM_Y; d++) {
      for (ps=0; ps<2; ps++) {
        const int psor2 = ps ? trans+1 : 0;
        struct S_commlist *rl;
        int *ri;
        if (ps>psnew || ZBound[ps][OH_UPPER]==0) {
          vph[d][ps][0] = vph[d][ps][1] = vpidx;
          continue;
        }
        if (psor2==2) {
          rl = AltSecRList;  ri = AltSecRLIndex;
        } else {
          rl = rlist[ps];  ri = rlidx[ps];
        }
        for (du=OH_LOWER; du<=OH_UPPER; du++) {
          const int vpisave = vpidx;
          const int nx = (d==OH_DIM_X) ? du<<1 : 1;
          const int ny = (d==OH_DIM_X) ? 1 : du<<1;
          const int n = 3*3 + 3*ny + nx;
          const int nrev = OH_NEIGHBORS - 1 - n;
          int nbor = Neighbors[psor2][n];
          vph[d][ps][du] = vpidx;
          if (nbor<0)  nbor = -(nbor+1);
          if (nbor<nn) {
            make_bsend_sched(psor2, n, nx, ny, rl+ri[nrev], &nsendib, &vpidx);
            make_brecv_sched(psor2, n, nx, ny, rl+ri[nrev], &nrecveb, vpisave);
          }
        }
      }
    }
    vph[2][0][0] = vpidx;
  }
```

4.13.34 Macros `Add_Pillar_Voxel()`, `Is_Pillar_Voxel()`, `Pillar_Lower()` and `Pillar_Upper()`

`Add_Pillar_Voxel()`
`Is_Pillar_Voxel()`
`Pillar_Lower()`
`Pillar_Upper()`

Here we show macros related to a value v in `NOfPGrid`$[][][]$ for a grid-voxel in an interior pillar or exterior pillar. As shown in §4.12.3, v for a grid-voxel can have $-(i+1) \cdot 2^{32} - (j+1) \cdot \leq -2^{32}$ when it is in an interior pillar, or $(i+1) \cdot 2^{32} + \sigma \geq 2^{32}$ when in an exterior pillar, to represent the index i of `BoundarySendBuf`$[]$ to which the particle in the grid-voxel is copied, together with another index j of the buffer or σ of `NOfSend`$[]$.

The macro `Add_Pillar_Voxel`(l) gives $l \cdot 2^{32}$ so that we have $(i+1) \cdot 2^{32}$ with $l = i+1$ to subtract it from $v = -(j+1)$ or add it to $v = \sigma$, or have 2^{32} to subtract it from v to *increment* its -2^{32} component. The macro `Is_Pillar_Voxel`(v) is true iff $v \geq 2^{32}$ to indicate $-v$ or v has $(i+1)$ component. The macro `Pillar_Lower`(v) gives $v \bmod 2^{31}$ and thus $(j+1)$ component of $-v$ or σ component of v. The macro `Pillar_Upper`(v) gives $\lfloor v/2^{32} \rfloor$ and thus $(i+1)$ component of $-v$ or v.

All of four macros are used in `Move_Or_Do()` when it is invoked from `move_and_sort()`, and in `Sort_Particle()` used in `sort_particles()` and `sort_received_particles()`. The macro `Add_Pillar_Voxel()` is also used in `make_bsend_sched()`, while `Is_Pillar_Voxel()` and `Pillar_Upper()` are also used in `xfer_boundary_particles_v()`.

```
#define Add_Pillar_Voxel(I)     (((dint)(I))<<32)
#define Is_Pillar_Voxel(V)      (V>=((dint)1)<<32)
#define Pillar_Lower(V)         (V&INT_MAX)
#define Pillar_Upper(V)         ((V)>>32)
```

4.13.35 `make_bsend_sched()`

`make_bsend_sched()` The function `make_bsend_sched()`, called solely from `make_bxfer_sched()` but up to $2 \times 2 \times 2 = 8$ times, scans $\mathcal{P}_O(p,s,g) =$ `NOfPGridOut`$[p][s][g]$ in a vertical interior halo plane of the subcuboid in the subdomain $n' = \{n, n^p_{old}, n^p_{new}\}[p']$ ($p' =$ `psor2`) assigned to the local node n as its primary ($p = p' = 0$) or secondary ($p = 1$, $p' \in \{1,2\}$) one, in order to build the sending schedule for halo particles in the plane to be sent to the nodes in the k-th ($k =$ `n`) neighbor family of n' according to k'-th ($k' = 3^D - 1 - k$) sub-block $\lambda =$ `rlist` in primary sending ($p' = 0$), secondary sending ($p' = 1$) or alternative secondary sending ($p' = 2$) blocks. Note that $k = 3^2 + \nu_y \cdot 3^1 + \nu_x \cdot 3^0$, where $\nu_x =$ `nx` $= \{2\beta, 0\}[d]$ and $\nu_y =$ `ny` $= \{0, 2\beta\}[d]$ for d-th dimensional lower ($\beta = 0$) or upper ($\beta = 1$) neighbor. The function also accumulates H_s pointed by `nsendptr` being the numbder of halo particles to be sent, and V pointed by `vpptr` being the number of `VPlane`[] enties consumed for bulding sending schedules.

This function is very *level-4s's own* and thus has no counterpart in level-4p.

```
static void
make_bsend_sched(const int psor2, const int n, const int nx, const int ny,
                 struct S_commlist *rlist, int *nsendptr, int *vpptr) {
  const int ns=nOfSpecies;
  const int ps = psor2==0 ? 0 : 1;
  const int tag1 = OH_NEIGHBORS;
  const int stag = n;
  const int rtag = ((ps ? OH_NEIGHBORS : 0) + (OH_NEIGHBORS - 1 - n));
  struct S_commlist *rl = rlist;
  int rlz = rl->region;
  int nsend = *nsendptr, nsendsave = nsend,  vpidx = *vpptr;
  int xl, xu, yl, yu;
  const int zbl = ZBound[ps][OH_LOWER];
  const int zbu = ZBound[ps][OH_UPPER] - GridDesc[psor2].z;
  const int zmax = ZBound[ps][OH_UPPER] - 1;
  const int xtop = GridDesc[psor2].x;
  int s;
  Decl_For_All_Grid();
```

At first, we specify $\mathcal{S}_{xy} = [x_l, x_u) \times [y_l, y_u)$ being each xy-subplane (or a line segment perpendicular to z-axis in usual cases with $e^g = 1$) in the vertical interior halo plane. The values y_l and y_u are always determined by `Grid_Interior_Boundary()`, and x_l and x_u are as well if $\nu_y = 1$ for west or east vertical interior halo plane. However $x_l = -e^g$ and $x_u = \delta_x(n') + e^g$ if $\nu_y \neq 1$ for south or north vertical interior halo plane, because it includes exterior pillars at its west and east ends. Then we do the followings after keeping H_s in H_s^0.

First we skip records in λ until we find the first family member m_f such that $\zeta_{p_f}^u(m_f) \geq \zeta_p^l(n)$ being the node which receives the particles in the bottom xy-plane of n's subcuboid. Then, we let elements of `VPlane`$[V]$ have the followings; m_f in `nbor`; $p_f \cdot 3^D + k$ in `stag`; $p \cdot 3^D + (3^D - 1 - k)$ in `rtag`; and H_s in `sbuf` because H_s has the so-far total number of halo particles to be sent and thus the head index of the send buffer $hbuf_v^s(d, p, \beta, m_f)$ in `BoundarySendBuf`[] into which particles sent to m_f are copied from corresponding grid-voxels in `SendBuf`[]. The assignments to `stag` and `rtag` are to make the communication between m_f and n for halo particles in the plane and its exterior counterpart unique, with a concept similar to that discussed in §4.13.24 but reflecting that particles of all species are transferred at once.

Next, we visit each xy-subplane at z for all $z \in [\zeta_p^l(n), \zeta_p^u(n))$ to scan $\mathcal{P}_O(p, s, g) =$ `NOfPGridOut`$[p][s][g]$ for all $s \in [0, S)$ and $g \in \mathcal{S}_{xy} \times \{z\}$ adding it to H_s. We also modify $v =$ `NOfPGrid`$[p][s][g]$ as follows.

1. If $g = gidx(x, y, z)$ is in an exterior pillar, i.e., $x < 0$ or $x \geq \delta_x(n')$ and thus $d = 1$ definitely, we modify $v = \sigma$ as $v = (H_s + 1) \cdot 2^{32} + \sigma$ to indicate that the halo particles in the grid-voxel received from a west/east neighbor should be copied into $hbuf_v^s(d, p, \beta, m_f)$ starting from `BoundarySendBuf`$[H_s]$ so that they are *relayed* to a south/north neighbor. We keep σ in new v because it is necessary for `move_and_sort()` to send particles originally in the grid-voxel to other node.

2. Otherwise and v is non-negative, we let $v = -(H_s + 1)$ so that particles in the grid-voxel are copied into $hbuf_v^s(d, p, \beta, m_f)$ starting from `BoundarySendBuf`$[H_s]$. Note that keeping the original v is unnecessary because it is definitely 0 for `move_and_sort()` (and `sort_received_particles()`) or it is no longer meaningful for `sort_particles()`.

3. Otherwise, i.e., v is negative to mean v has $-(H'+1)$ with some H' given in the case 2, the grid-voxel has already been visited and thus is in an interior pillar. Therefore, we let $v = -(H_s+1) \cdot 2^{32} - (H'+1)$ so that particles in the grid-voxel are copied into both of $hbuf_v^s(d, p, \beta, m_f)$ and $hbuf_v^s(d', p', \beta', m_f')$ starting from `BoundarySendBuf`$[H_s]$ and `BoundarySendBuf`$[H']$ respectively.

Then each time we compelete the scan of a xy-plane for all $s \in [0, S)$, we examine if $z = \zeta_{p_f}^u(m_f) - 1$ to mean z is at the top surface of m_f's subcuboid. If so, we let `VPlane`$[V]$`.nsend` $= H_s - H_s^0$ being the number of halo particles to be sent to m_f, i.e., the size of $hbuf_v^s(d, p, \beta, m_f)$. Now the buffer $hbuf_v^s(d, p, \beta, m_f)$ is formed as a conceptual 5-dimensional array whose elements $[z][s][y][x][i]$ are for $z \in [\zeta_p^l(n), \zeta_p^u(n)) \cap [\zeta_{p_f}^l(m_f), \zeta_{p_f}^u(m_f))$, $s \in [0, S)$, $y \in [y_l, y_u)$, $x \in [x_l, x_u)$ and $i \in [0, \mathcal{P}_O(p, s, gidx(x, y, z)))$. Then, if $z < \zeta_p^u(n) - 1$, we step to the next m_f fetching the next record in λ and incrementing V, and then let `nbor`, `stag`, `rtag` and `sbuf` elements of `VPlane`$[V]$ be m_f, $p_f \cdot 3^D + k$, $p \cdot 3^D + (3^D - 1 - k)$ and H_s respectively, updating H_s^0 to keep H_s.

At the end of the loop for $z \in [\zeta_p^l(n), \zeta_p^u(n))$, the series of the buffers $hbuf_v^s(d, p, \beta, m_f)$ for $f \in \{0, \ldots\}$ is formed as a conceptual 5-dimensional array as a whole whose elements $[z][s][y][x][i]$ are for $z \in [\zeta_p^l(n), \zeta_p^u(n))$, $s \in [0, S)$, $y \in [y_l, y_u)$, $x \in [x_l, x_u)$ and $i \in [0, \mathcal{P}_O(p, s, gidx(x, y, z)))$, since $\zeta_{p_f}^u(m_f) = \zeta_{p_{f+1}}^l(m_{f+1})$ for all f. Therefore, the whole series of $hbuf_v^s(d, p, k, m_f)$ for $d \in \{0, 1\}$, $p \in \{0, p_n\}$ and $\beta \in \{0, 1\}$ established by the series of the calls of this function from `make_bxfer_sched()` is formed as a 8-dimensional array whose elements $[d][p][\beta][z][s][y][x][i]$ are for those shown above but y_l, y_u, x_l and x_u are dependent on d, p and β. Moreover, the index of `BoundarySendBuf[]` for the element $[d][p][\beta][z][s][y][x][0]$ is given by `NOfPLocal`$[p][s][gidx(x, y, z)]$.

Finally, we *return* H_s and $V + 1$ to the caller `make_bxfer_sched()` for the successive calls of this function itself, after letting `VPlane`$[V]$`.nsend` $= H_s - H_s^0$. Note that since V has, at the end of the loop, the index of the last entry consumed by the function, we return $V + 1$ rather than V.

```
if (ny==1) {
  Grid_Interior_Boundary(nx, GridDesc[psor2].x, xl, xu);
} else {
  xl = -OH_PGRID_EXT;  xu = OH_PGRID_EXT;
}
Grid_Interior_Boundary(ny, GridDesc[psor2].y, yl, yu);

while (rlz<zbl)  rlz = (++rl)->region;
VPlane[vpidx].nbor = rl->rid;
VPlane[vpidx].stag = (rl->tag ? tag1 : 0) + stag;
VPlane[vpidx].rtag = rtag;
VPlane[vpidx].sbuf = nsend;
For_All_Grid_Z(psor2, xl, yl, zbl, xu, yu, zbu) {
  const int z = Grid_Z();
  for (s=0; s<ns; s++) {
    dint *npg = NOfPGrid[ps][s];
    int *npgo = NOfPGridOut[ps][s];
    For_All_Grid_XY(psor2, xl, yl, xu, yu) {
      const int g = The_Grid();
      const dint dst = npg[g];
      if (Grid_X()<0 || Grid_X()>=xtop)
        npg[g] += Add_Pillar_Voxel(nsend+1);
      else if (dst>=0)
        npg[g] = -(nsend+1);
      else
        npg[g] -= Add_Pillar_Voxel(nsend+1);
      nsend += npgo[g];
    }
  }
  if (z==rlz && z<zmax) {
    VPlane[vpidx++].nsend = nsend - nsendsave;
    rlz = (++rl)->region;
    VPlane[vpidx].nbor = rl->rid;
    VPlane[vpidx].stag = (rl->tag ? tag1 : 0) + stag;
    VPlane[vpidx].rtag = rtag;
    VPlane[vpidx].sbuf = nsendsave = nsend;
  }
}
VPlane[vpidx].nsend = nsend - nsendsave;
```

```
  *nsendptr = nsend;  *vpptr = vpidx + 1;
}
```

4.13.36 make_brecv_sched()

make_brecv_sched()

The function make_brecv_sched(), called solely from make_bxfer_sched() but up to $2 \times 2 \times 2 = 8$ times, scans $\mathcal{P}_O(p,s,g) =$ NOfPGridTotal$[p][s][g]$ in a vertical exterior halo plane surrounding the subcuboid in the subdomain $n' = \{n, n^p_{old}, n^p_{new}\}[p']$ ($p' =$ psor2) assigned to the local node n as its primary ($p = p' = 0$) or secondary ($p = 1$, $p' \in \{1,2\}$) one, in order to build the receiving schedule for halo particles in the plane to be received from the nodes in the k-th ($k =$ n) neighbor family of n' according to the k'-th ($k' = 3^D - 1 - k$) sub-block $\lambda =$ rlist in primary sending ($p' = 0$), secondary sending ($p' = 1$) or alternative secondary sending ($p' = 2$) blocks. Note that $k = 3^2 + \nu_y \cdot 3^1 + \nu_x \cdot 3^0$, where $\nu_x =$ nx $= \{2\beta, 0\}[d]$ and $\nu_y =$ ny $= \{0, 2\beta\}[d]$ for d-th dimensional lower ($\beta = 0$) or upper ($\beta = 1$) neighbor. The function also accumulates H_r pointed by nrecvptr being the number of particles to be received, and the index $V =$ vpidx of VPlane[] from which make_bsend_sched() has built the sending schedule for the corresponding vertical interior halo plane and thus this function will do as well.

This function is very *level-4s's own* and thus has no counterpart in level-4p.

```
static void
make_brecv_sched(const int psor2, const int n, const int nx, const int ny,
                 struct S_commlist *rlist, int *nrecvptr, int vpidx) {
  const int ns=nOfSpecies;
  const int ps = psor2==0 ? 0 : 1;
  int nrecv = *nrecvptr,  nrecvsave = nrecv;
  struct S_commlist *rl = rlist;
  int rlz = rl->region;
  int xl, xu, yl, yu;
  const int zbl = ZBound[ps][OH_LOWER];
  const int zbu = ZBound[ps][OH_UPPER] - GridDesc[psor2].z;
  const int zmax = ZBound[ps][OH_UPPER] - 1;
  int s;
  Decl_For_All_Grid();
```

At first, we specify $\mathcal{S}_{xy} = [x_l, x_u) \times [y_l, y_u)$ being each xy-subplane (or a line segment perpendicular to z-axis in usual cases with $e^g = 1$) in the vertical exterior halo plane. The values y_l and y_u are always determined by Grid_Exterior_Boundary(), and x_l and x_u are as well if $\nu_y = 1$ for west or east vertical exterior halo plane. However $x_l = -e^g$ and $x_u = \delta_x(n') + e^g$ if $\nu_y \neq 1$ for south or north vertical exterior halo plane, because it includes pillars, being the intersection of west/east and south/north vertical exterior halo planes at its west and east ends. Then we do the followings after keeping H_r in H^0_r.

First we skip records in λ until we find the first family member m_f such that $\zeta^u_{p_f}(m_f) \geq \zeta^l_p(n)$ being the node which sends the particles in the bottom xy-plane of n's subcuboid. Then, we let VPlane$[V]$.rbuf $= H_r$ because H_r has the so-far total number of halo particles to be received and thus the head index of the receive buffer $hbuf^r_v(d,p,\beta,m_f)$ in Particles[] from which particles received from m_f are copied to corresponding grid-voxels in SendBuf[].

Next, we visit each xy-subplane at z for all $z \in [\zeta^l_p(n), \zeta^u_p(n))$ to scan $\mathcal{P}_O(p,s,g) =$ NOfPGridOut$[p][s][g]$ for all $s \in [0,S)$ and $g \in \mathcal{S}_{xy} \times \{z\}$ adding it to H_r. Then each time we compelete the scan of a xy-plane, we examine if $z = \zeta^u_{p_f}(m_f) - 1$ to mean z is

at the top surface of m_f's subcuboid. If so, we let `VPlane`$[V]$`.nrecv` $= H_r - H_r^0$ being the number of halo particles to be received from m_f, i.e., the size of $hbuf_v^r(d, p, \beta, m_f)$. Now the buffer $hbuf_v^r(d, p, \beta, m_f)$ is formed as a conceptual 5-dimensional array whose elements $[z][s][y][x][i]$ are for $z \in [\zeta_p^l(n), \zeta_p^u(n)) \cap [\zeta_{p_f}^l(m_f), \zeta_{p_f}^u(m_f))$, $s \in [0, S)$, $y \in [y_l, y_u)$, $x \in [x_l, x_u)$ and $i \in [0, \mathcal{P}_O(p, s, gidx(x, y, z)))$. Then, if $z < \zeta_p^u(n) - 1$, we step to the next m_f fetching the next record in λ and incremening V, and then let `VPlane`$[V]$`.rbuf` $= H_r$, as we did before the loop, updating H_r^0 to keep H_r.

At the end of the loop for $z \in [\zeta_p^l(n), \zeta_p^u(n))$, the series of the buffers $hbuf_v^r(d, p, \beta, m_f)$ for $f \in \{0, \ldots\}$ is formed as a conceptual 5-dimensional array as a whole whose elements $[z][s][y][x][i]$ are for $z \in [\zeta_p^l(n), \zeta_p^u(n))$, $s \in [0, S)$, $y \in [y_l, y_u)$, $x \in [x_l, x_u)$ and $i \in [0, \mathcal{P}_O(p, s, gidx(x, y, z)))$, since $\zeta_{p_f}^u(m_f) = \zeta_{p_{f+1}}^l(m_{f+1})$ for all f. Therefore, the whole series of $hbuf_v^r(d, p, \beta, m)$ for $d \in \{0, 1\}$, $p \in \{0, p_n\}$ and $\beta \in \{0, 1\}$ established by the series of the calls of this function from `make_bxfer_sched()` is formed as a 8-dimensional array whose elements $[d][p][\beta][z][s][y][x][i]$ are for those shown above but y_l, y_u, x_l and x_u are dependent on d, p and β.

Finally, we *return* H_r to the caller `make_bxfer_sched()` for the successive calls of this function itself, after letting `VPlane`$[V]$`.nrecv` $= H_r - H_r^0$.

```
    if (ny==1) {
      Grid_Exterior_Boundary(nx, GridDesc[psor2].x, xl, xu);
    } else {
      xl = -OH_PGRID_EXT;  xu = OH_PGRID_EXT;
    }
    Grid_Exterior_Boundary(ny, GridDesc[psor2].y, yl, yu);

    while (rlz<zbl)  rlz = (++rl)->region;
    VPlane[vpidx].rbuf = nrecv;
    For_All_Grid_Z(psor2, xl, yl, zbl, xu, yu, zbu) {
      const int z = Grid_Z();
      for (s=0; s<ns; s++) {
        int *npgo = NOfPGridOut[ps][s];
        For_All_Grid_XY(psor2, xl, yl, zu, yu)
          nrecv += npgo[The_Grid()];
      }
      if (z==rlz && z<zmax) {
        VPlane[vpidx++].nrecv = nrecv - nrecvsave;
        rlz = (++rl)->region;
        VPlane[vpidx].rbuf = nrecvsave = nrecv;
      }
    }
    VPlane[vpidx].nrecv = nrecv - nrecvsave;  *nrecvptr = nrecv;
  }
```

4.13.37 Macros `Local_Grid_Position()` and `Move_Or_Do()`

`Local_Grid_Position()`

The macro `Local_Grid_Position`$(g, k{\cdot}2^\Gamma + g, p)$, used in the macro `Move_Or_Do()` and the function `oh4s_remove_mapped_particle()` directly, transforms a particle's grid-position g in the k-th neighbor of the local node n's primary ($p = 0$) or secondary ($p = 1$) subdomain into its corresponding index g' in the n's subdomain by $g +$ `GridOffset`$[p][k]$. The macro is perfectly equivalent to its level-4p counterpart shown in §4.13.37.

```
#define Local_Grid_Position(G, NID, PS)  ((G) + GridOffset[PS][NID>>loggrid])
```

`Move_Or_Do()` The macro `Move_Or_Do`$(\pi, p, n', \mu, a, \eta)$, used in the particle scanning loop in `move_to_sendbuf_4s()`, `move_to_sendbuf_uw4s()` and `move_to_sendbuf_dw4s()` with $\eta = 0$, and `move_and_sort()` with $\eta = 1$, examines a primary ($p = 0$) or secondary ($p = 1$) particle π of species s in `Particles`[] and moves/copies the particle in `Particles`[], to `SendBuf`[] and/or to `BoundarySendBuf`[]. This macro is somewhat similar to its level-4p counterpart shown in §4.10.40 but significantly *different* from it because we have no hot-spots but have halo particles.

The first part is, however, quite similar to the counterpart. That is, If π.`nid` < 0 to mean the particle was eliminated, we skip the iteration of the loop by `continue`. Otherwise we obtain its subdomain identifier m by `Neighbor_Subdomain_Id()` and grid-position g by `Grid_Position()` if $m = n'$, or by transforming what the macro gives into the local coordinate g by `Local_Grid_Position()` otherwise, where n' is the primary/secondary subdomain of the local node.

```
#define Move_Or_Do(P, PS, MYSD, TOSB, ACT, PIL) {\
  const OH_nid_t nid = P->nid;\
  int g = Grid_Position(nid);\
  int sdid;\
  dint dst;\
  if (nid<0)  continue;\
  sdid = Neighbor_Subdomain_Id(nid, PS);\
  if (sdid!=(MYSD)) g = Local_Grid_Position(g, nid, PS);\
```

Then if $v =$ `npg`$[g] =$ `NOfPGrid`$[p][s][g] = 0$, where the pointer to `NOfPGrid`$[p][s][0]$ is given through the implicit argument `npg`, to mean that the particle should stay in the local node, we perform the operation a specified by the macro invoker, which is to move π in `Particles`[] or to `SendBuf`[], except for the invocation in a loop of `move_to_sendbuf_uw4s()` skipping particles.

If $v > 0$, on the other hand, it means that $v \bmod 2^{32} = \sigma = ((p'S + s)N + m') + 1$ representing the one-dimensional index of $[p'][s][m']$ of `NOfSend`$[2][S][N]$ and π is to be sent to m' as its (not the local node's) primary ($p' = 0$) or secondary ($p' = 1$) particle. Therefore, we move π to `SendBuf`$[\beta +$ `NOfSend`$[p'][s][m']]$ and then increment `NOfSend`$[p'][s][m']$ for the next particle to be sent to m' if $\mu \neq 0$. Note that `SendBuf`$[\beta]$ is given through the implicit argument `sb`, and $\beta = Q_n$ if the macro is invoked in `move_and_sort()`, or $\beta = 0$ otherwise. Also note that it can be $v \geq 2^{32}$ only when $\eta \neq 0$ requiring us to extract σ by `Pillar_Lower()`, and it cannot be $\sigma = 0$ if $v \geq 2^{32}$ because the grid-voxel having π is in a vertical exterior halo plane[97]. In addition, the conditionals examining η and μ should be eliminated by compilers because they are constant in functions using this macro.

Otherwise, $v < 0$ to mean that π is in a vertical interior halo plane and definitely $\eta \neq 0$. Since π stays in the local node, the operation a takes place. Then if $v > -2^{32}$ having $-(i + 1)$, we copy π to `BoundarySendBuf`$[i]$, and then increment i by decrementing v for the particle next to π. Otherwise, i.e., $v = -(i + 1) \cdot 2^{32} - (j + 1) \leq -2^{32}$ to mean g is in an interior pillar, we copy π to `BoundarySendBuf`$[i]$ and `BoundarySendBuf`$[j]$, and then increment i and j by decrementing v by $2^{32} + 1$ for the particle next to π.

[97] This does not means a grid-voxel cannot have $v = (i + 1)2^{32}$ with $\eta \neq 0$. In fact, if a neighbor of n' is n' itself, grid-voxels in an exterior pillar in a vertical exterior halo plane for halo particles received from the self neighbor definitely has such value since the grid-voxels do not have any particles. However, since no particles in such grid-voxels, they are not accessed in `Move_Or_Do()`.

```
    dst = npg[g];\
    if (dst==0)  { ACT; }\
    else if (!PIL) {\
      if (TOSB)  sb[NOfSend[dst-1]++] = *P;\
    }\
    else if (dst>0)\
      sb[NOfSend[Pillar_Lower(dst)-1]++] = *P;\
    else {\
      ACT;\
      const dint bsbidx = -dst;\
      if (!Is_Pillar_Voxel(bsbidx)) {\
        BoundarySendBuf[bsbidx-1] = *P;  npg[g] = dst - 1;\
      }\
      else {\
        BoundarySendBuf[Pillar_Lower(bsbidx)-1] =\
          BoundarySendBuf[Pillar_Upper(bsbidx)-1] = *P;\
        npg[g] = dst - (Add_Pillar_Voxel(1) + 1);\
      }\
    }\
  }
```

4.13.38 `move_to_sendbuf_4s()`

`move_to_sendbuf_4s()` The function `move_to_sendbuf_4s()`, called solely from `exchange_particles4s()` when it finds $Q_n + P_n^{\text{send}} > P_{lim}$ or we have helpand-helper reconfiguration to mean we have to transfer particles among neighbors before sorting, is the level-4s counterpart of `move_to_sendbuf_sec4p()` shown in §4.10.41, to move particles to be sent to `SendBuf[]` and pack those to stay in the local nodes in `Particles[]`. The function is literally very similar to the counterpart but has the following *differences* from it.

- Since `exchange_particles4s()` works not only in the case that we will be in secondary mode but also in primary mode, this function is called both cases as well. Therefore, the function is given `nextmode` $= p'_n$ being 0 or 1 to indicate the mode we will be in. However, the mode in the next step does not affect the mechanism of the function almost at all, because data structures referred to in this function have values consistent with primary mode in the next step as discussed in §4.13.14 and thus the mechanism can depend almost only on the mode in the last step. Therefore, p'_n is used only as an argument of `oh1_stats_time()` and `set_sendbuf_disps4s()`.
- We need another mode indicator argument for the next step, namely `psnew` $= p_n \in \{0, 1\}$, being 1 iff the local node will have secondary subdomain and thus secondary particles. This argument is used when we scan `NOfRecv`$[p][][]$ for $p \in \{0, p_n\}$ to know the size of $rbuf(p, s)$ in $pbuf_i(p, s)$ for each $s \in [0, S)$ so that `InteriorParts`$[p][s]$`.size` has it as the initial value.
- Since we have no hot-spots but halo particles, what `Move_Or_Do()` used in the function does is significantly *different* from its level-4p counterpart as discussed in §4.13.37, though the literal difference is just that the macro has an additional and last argument η being 0 in this function to mean that copying halo particles to `BoundarySendBuf[]` does not take place in this function (but in `sort_particles()`).

- The funcitons `set_sendbuf_disps4s()`, `move_to_sendbuf_uw4s()` and `move_to_sendbuf_dw4s()` called in this function have level-4s specific names and small *differences* from their level-4p counterparts.

- Since the argument `nacc`[2] has $\{Q_n^n, Q_n\}$ instead of $\{Q_n^n, Q_n^{n^p_{new}}\}$, we simply passes `nacc`[1] to the second call of `move_to_sendbuf_dw4s()` without calculating Q_n.

On the other hand, the arguments other than `nextmode`, `psnew` and `nacc` are equivalent to those of the counterpart as follows; `psold` $= p_c = 1$ if the local node had a secondary subdomain or 0 otherwise; `trans` $= t = 1$ iff we have transitional state of helpand-helper configuration; `oldp` $= n^p_{old}$ being the local node n's helpand in the last step; `nsend` $= P_n^{\rm send}$; and `stats` $\neq 0$ if we have to start new timing measurement. The reason why n^p_{old} is given instead of n^p_{new} is also as same as that shown in §4.10.41 for the counterpart.

```
static void
move_to_sendbuf_4s(const int nextmode, const int psold, const int psnew,
                   const int trans, const int oldp, const int *nacc,
                   const int nsend, const int stats) {
  const int me=myRank, ns=nOfSpecies, nn=nOfNodes, sbase=specBase;
  const int ninj=nOfInjections, nplim=nOfLocalPLimit;
  int ps, s, t, i;
  int *nofr;
  int ninjp=0, ninjs=nplim;
  struct S_particle *sb = SendBuf,  *p;
  Decl_Grid_Info();
```

As done in the level-4p counterpart, after starting the new timing measurement with the key `STATS_TB_MOVE` if required by `stats` $\neq 0$, we call `set_sendbuf_disps4s()` to build the index array of $sbuf(p,s,m)$ in `NOfSend`$[p][s][m]$ for $m \in$ `RealDstNeighbors`$[t][p]$ with $p \in \{0, p'_n\}$ giving it p'_n and t as its argument. The argument p'_n (not p_n) is *level-4s's own* to keep the function from working on `NOfSend`[1][][] referring to meaningless values in `RealDstNeighbors`$[0][1]$[98].

Next, as a very *level-4s's own* operation, we let `InteriorParts`$[p][s]$`.size` have the following for all $p \in \{0,1\}$ and $s \in [0,S)$.

$$\mathcal{N}_S(p) = \texttt{RealSrcNeighbors}[t][p]\texttt{.nbor}[]$$

$$\texttt{InteriorParts}[p][s]\texttt{.size} = \begin{cases} \displaystyle\sum_{m\in\mathcal{N}_s(p)} \texttt{NOfRecv}[p][s][m] & p \le p_n \\ 0 & p > p_n \end{cases}$$

That is `InteriorParts`$[p][s]$`.size` is let have the size of $rbuf(p,s)$, which is 0 for $p = 1$ if $p_n = 0$. This value is the *base* of the size of $pbuf_i(p,s)$ being added to the number of particles staying and thus being packed into $pbuf_i(p,s)$ by `move_to_sendbuf_uw4s()` and `move_to_sendbuf_dw4s()`.

Then, again as done in the counterpart, we scan all injected particles invoking `Move_Or_Do()` for each to move those staying in the local node to the tail region of `SendBuf`[] temporalily, and to do others to corresponding $sbuf(p,s,m)$ in `SendBuf`[]. Since we tell the macro that halo particles are not taken care of in this function by $\eta = 0$, the mechanism of this part is almost equivalent to the level-4p counterpart for particles in non-hot-spot

[98] Though letting the function work on $p = 1$ is safe.

grid-voxels. The *difference* is, however, that we need to enlarge $rbuf(p,s)$ by increasing `InteriorParts`$[p][s]$`.size` for each injected-and-staying particle because it will be moved back into the buffer. Note that we can be unaware the possiblity that we have some secondary injected particles and turn the mode from secondary to primary, because, if so, any `NOfPGrid`[][][] for them tell us that they should send to other node, old helpand in fact and thus definitely we will have none of them at the tail of `SendBuf`[], as happening in the case of helpand-helper reconfiguration changing helpand.

```
if (stats) oh1_stats_time(STATS_TB_MOVE, nextmode);
set_sendbuf_disps4s(nextmode, trans);

for (ps=0,t=0,nofr=NOfRecv; ps<2; ps++) {
  const int nnbr = RealSrcNeighbors[trans][ps].n;
  const int *rnbr = RealSrcNeighbors[trans][ps].nbor;
  if (ps<=psnew) {
    for (s=0; s<ns; s++,t++,nofr+=nn) {
      int n, nrec;
      for (n=0,nrec=0; n<nnbr; n++)  nrec += nofr[rnbr[n]];
      InteriorParts[t].size = nrec;
    }
  } else
    for (s=0; s<ns; s++,t++)  InteriorParts[t].size = 0;
}
for (i=0,p=Particles+totalParts; i<ninj; i++,p++) {
  const int s = Particle_Spec(p->spec-sbase);
  const OH_nid_t nid = p->nid;
  const int ps = Secondary_Injected(nid) ? 1 : 0;
  dint *npg = NOfPGrid[ps][s];
  if (nid<0) continue;
  if (ps) {
    Primarize_Id_Only(p);
    Move_Or_Do(p, ps, oldp, 1,
               (sb[--ninjs]=*p, InteriorParts[ns+s].size++), 0);
  } else
    Move_Or_Do(p, ps, me, 1,
               (sb[nsend+ninjp++]=*p, InteriorParts[s].size++), 0);
}
```

The next part is almost logically equivalent to that of the level-4p counterpart for the case the local node does not have any hot-spots in its subdomains, though the code is a little bit *different* as discussed at the beginning of this section. That is, we call `move_to_sendbuf_uw4s()` and `move_to_sendbuf_dw4s()` once or twice for each according to p_c being 0 or 1 respectively, to pack particles to stay in the local node in `Particles`[], to move others to `SendBuf`[], and, as a *level-4s's own* operation, to let `InteriorParts`$[p][s]$ has the index and size of $pbuf_i(p,s)$. In addition, if $p_c = 0$ we let `RecvBufBases`$[1][s]$ point the head of $pbuf(1,s)$ and, as another *level-4s's own* operation, let `InteriorParts`$[1][s]$`.head` have its index so that $rbuf(1,s) = pbuf_i(1,s)$ because its `size` element has $|rbuf(1,s)|$, knowing that this is meaningless with $p_n = 0$ but safe because they will not be referred to (and have meaningful common vaules $Q_n^n = Q_n$).

```
move_to_sendbuf_uw4s(0, me, 0, 0);
if (psold) {
  move_to_sendbuf_uw4s(1, oldp, primaryParts, nacc[0]);
  move_to_sendbuf_dw4s(1, oldp, totalParts, nacc[1]);
```

```
  } else {
    struct S_particle *rbb=Particles+nacc[0];
    int s;
    for (s=0; s<ns; s++) {
      RecvBufBases[ns+s] = rbb;  InteriorParts[ns+s].head = rbb - Particles;
      rbb += TotalPNext[ns+s];
    }
  }
  move_to_sendbuf_dw4s(0, me, primaryParts, nacc[0]);
```

The last part, to move back injected and staying particles from `SendBuf[]` to `Particles[]` and to let `primaryParts` and its shadow pointed by `secondaryBase` be Q_n^n in the next step, is literally and logically equivalent to that of the level-4p counterpart.

```
  for (i=0,p=SendBuf+nsend; i<ninjp; i++,p++)
    *(RecvBufBases[Particle_Spec(p->spec-sbase)]++) = *p;
  for (i=ninjs,p=SendBuf+ninjs; i<nplim; i++,p++)
    *(RecvBufBases[Particle_Spec(p->spec-sbase)+ns]++) = *p;

  primaryParts = *secondaryBase = nacc[0];
}
```

4.13.39 `move_to_sendbuf_uw4s()`

`move_to_sendbuf_uw4s()` The function `move_to_sendbuf_uw4s()`, called solely from `move_to_sendbuf_4s()` once or twice, scans particles in the local node n's primary (`ps` $= p = 0$) or secondary ($p = 1$) subdomain `mysd` $= m$ from `Particles`$[b_0^-]$ where b_0^- = `cbase` argument for $pbuf(p, 0)$ in the last step. It moves scanned particles to be sent to other nodes to `SendBuf[]`, and packs those to stay in the local node *upward*, i.e., to the direction of smaller indices of `Particles[]` and to the region beginning `Particles`$[b_0^+]$ where b_0^+ = `nbase` argument for the packed $pbuf(p, 0)$.

This function is almost logically equivalent to no hot-spot case of its level-4p counterpart `move_to_sendbuf_uw4p()` shown in §4.10.42 but has the following *differences* from it, where $b_s^- = b_{s-1}^- +$ `TotalP`$[p][s-1]$ and $b_s^+ = b_{s-1}^+ +$ `TotalPNext`$[p][s-1]$.

- In the cases of $b_s^+ \le b_s^-$ or $b_{s+1}^+ \le b_{s+1}^-$, i.e. the cases in which old $pbuf(p, s)$ is scanned by this function, we let `InteriorParts`$[p][s]$`.head` $= b_s^+$ being the head index of new $pbuf(p, s)$ because at its top $pbuf_i(p, s)$ is placed. We also add the number of packed staying particles to `InteriorParts`$[p][s]$`.size` which had the size of $rbuf(p, s)$ placed at the bottom of $pbuf_i(p, s)$.
- The case of $b_s^+ > b_s^- \ \wedge \ b_{s+1}^+ > b_{s+1}^-$ is eliminated from this function because we can do nothing for the case including the placement of $rbuf(p, s)$ which is now placed at the top of $pbuf_i(p, s)$ rather than that of $pbuf(p, s)$.
- For all four invocations of the macro `Move_Or_Do()` in the loop body, we add the last argument $\eta = 0$ to mean halo particles are not taken care of in this function. Therefore, in the first invocation with $b_s^+ \le b_s^-$, and the third invocation scanning the bottom half of $pbuf(p, s)$ with $b_s^+ > b_s^-$ and $b_{s+1}^+ \le b_{s+1}^-$, the macro works equivalently to no hot-spot case of its level-4p counterpart.

- In the case of $b_s^+ > b_s^-$ and $b_{s+1}^+ \le b_{s+1}^-$, we give $\mu = 0$ to the macro `Move_Or_Do()` in the first scan of the top half of $pbuf(p, s)$, and then give $\mu = 1$ in the second scan. This means that the first scan just counts the number of staying particles without moving any particles, while the second scan moves them to `SendBuf[]` or in `Particles[]` for packing. Since there are no hot-spots, the fate of all particles in a grid-voxel are common and thus the operations are correct.

```
static void
move_to_sendbuf_uw4s(const int ps, const int mysd, const int cbase,
                     const int nbase) {
  const int ns=nOfSpecies;
  const int nsor0 = ps ? ns : 0;
  const int *ctp = TotalP + nsor0,  *ntp = TotalPNext + nsor0;
  struct S_interiorp *ip = InteriorParts + nsor0;
  struct S_particle *p,  **rbb = RecvBufBases + nsor0,  *sb = SendBuf;
  int s, c, d, cn, dn;
  Decl_Grid_Info();

  for (s=0,c=cbase,d=nbase; s<ns; s++,c=cn,d=dn) {
    dint *npg = NOfPGrid[ps][s];
    cn = c + ctp[s];  dn  = d + ntp[s];
    ip[s].head = d;
    if (d<=c) {
      for (p=Particles+c; c<cn; c++,p++)
        Move_Or_Do(p, ps, mysd, 1, (Particles[d++]=*p), 0);
      rbb[s] = Particles + d;  ip[s].size += d - ip[s].head;
    } else if (dn<=cn) {
      const int cb = c;
      int cm, dm;
      for (p=Particles+c; c<d; c++,p++)
        Move_Or_Do(p, ps, mysd, 0, (d++), 0);
      cm = c - 1;  dm = d - 1;
      for (p=Particles+c; c<cn; c++,p++)
        Move_Or_Do(p, ps, mysd, 1, (Particles[d++]=*p), 0);
      rbb[s] = Particles + d;  ip[s].size += d - ip[s].head;
      for (c=dm,d=dm,p=Particles+c; c>=cb; c--,p--)
        Move_Or_Do(p, ps, mysd, 1, (Particles[d--]=*p), 0);
    }
  }
}
```

4.13.40 `move_to_sendbuf_dw4s()`

`move_to_sendbuf_dw4s()` The function `move_to_sendbuf_dw4s()`, called solely from `move_to_sendbuf_4s()` once or twice, scans particles in the local node n's primary (`ps` $= p = 0$) or secondary ($p = 1$) subdomain `mysd` $= m$ from `Particles`$[b_S^- - 1]$ where b_S^- = `ctail` argument for the element following $pbuf(p, S-1)$ in the last step. It moves scanned particles to be sent to other nodes to `SendBuf[]`, and packs those to stay in the local node *downward*, i.e., to the direction of greater indices of `Particles[]` and to the region ending `Particles`$[b_S^+ - 1]$ where b_S^+ = `ntail` argument for the element following the packed $pbuf(p, S-1)$.

This function is almost logically equivalent to no hot-spot case of its level-4p counterpart `move_to_sendbuf_dw4p()` shown in §4.10.43 but has the following *differences*.

- We give $\eta = 0$ to the macro `Move_Or_Do()` in the particle scanning loop telling it that halo particles are not taken care of. Therefore, the function is logically equivalent to no hot-spot case of the counterpart.
- After the partcile scanning loop for $pbuf(p,s)$ from `Particles`$[b^-_{s+1} - 1]$ to `Particles`$[b^-_s]$ and to move those staying into $pbuf_i(p,s)$ whose tail is at `Partilce`$[b^+_{s+1} - 1]$, we let

$$\begin{aligned}
\texttt{InteriorParts}[p][s]\texttt{.head} &= i - |rbuf(p,s)| \\
\texttt{InteriorParts}[p][s]\texttt{.size} &= b^+_{s+1} - i + |rbuf(p,s)| \\
\texttt{RecvBufBases}[p][s] &= \texttt{Particles} + i - |rbuf(p,s)|
\end{aligned}$$

  where $|rbuf(p,s)|$ is kept in `InteriorParts`$[p][s]$`.size` until its update above, i is the index of `Particles`[] for the last particle moved into $pbuf_i(p,s)$, $b^-_s = b^-_{s+1} -$ `TotalP`$[p][s{+}1]$ and $b^+_s = b^+_{s+1} -$ `TotalPNext`$[p][s{+}1]$, so that $pbuf_i(p,s)$ is placed at the bottom of $pbuf(p,s)$ and $rbuf(p,s)$ at the top of $pbuf_i(p,s)$ (rather than the top of $pbuf(p,s)$).

```
static void
move_to_sendbuf_dw4s(const int ps, const int mysd, const int ctail,
                     const int ntail) {
  const int ns=nOfSpecies;
  const int nsor0 = ps ? ns : 0;
  const int *ctp = TotalP + nsor0,  *ntp = TotalPNext + nsor0;
  struct S_interiorp *ip = InteriorParts + nsor0;
  struct S_particle *sb = SendBuf,  *p,  **rbb = RecvBufBases + nsor0;
  int s, c, d, cn, dn;
  Decl_Grid_Info();

  cn = ctail;  dn = ntail;
  for (s=ns-1,c=cn-1,d=dn-1; s>=0; s--,c=cn-1,d=dn-1) {
    dint *npg = NOfPGrid[ps][s];
    const int dd = d;
    cn -= ctp[s];  dn -= ntp[s];
    if (c>=d || cn>=dn)  continue;
    for (p=Particles+c; c>=cn; c--,p--)
      Move_Or_Do(p, ps, mysd, 1, (Particles[d--]=*p), 0);
    ip[s].head = d - ip[s].size + 1;  ip[s].size += dd - d;
    rbb[s] = Particles + ip[s].head;
  }
}
```

4.13.41 Macro `Sort_Particle()`

`Sort_Particle()` The *level-4s's own* macro `Sort_Particle`(π), used in `sort_particles()` and `sort_received_particles()`, moves local node n's primary ($p = 0$) or secondary ($p = 1$) particle π of species s at $g =$ `Grid_Position`(π`.nid`) in n's subcuboid to `SendBuf`$[$`NOfPGridTotal`$[p][s][g]]$ for sorting and increments `NOfPGridTotal`$[p][s][g]$ for the next particle in g. Then if $v =$ `NOfPGrid`$[p][s][g] < 0$ to mean g is in a interior halo plane, π is copied to `BoundarySendBuf`$[i]$ if $v = -(i+1) > -2^{32}$ incrementing i by decrementing v, or to `BoundarySendBuf`$[i]$ and `BoundarySendBuf`$[j]$ otherwise to mean

$v = -(i+1) \cdot 2^{32} - (j+1) \leq -2^{32}$ incrementing i and j by subtracting $2^{32}+1$ from v, as done in `Move_Or_Do()`. Note that `NOfPGridTotal`$[p][s][]$ and `NOfPGrid`$[p][s][]$ are given through implicit arguments `npgt` and `npg`.

```
#define Sort_Particle(P) {\
  const int g = Grid_Position(P->nid);\
  const dint dst = npg[g];\
  SendBuf[npgt[g]++] = *P;\
  if (dst<0) {\
    const dint bsbidx = -dst;\
    if (!Is_Pillar_Voxel(bsbidx)) {\
      BoundarySendBuf[bsbidx-1] = *P;  npg[g] = dst - 1;\
    }\
    else {\
      BoundarySendBuf[Pillar_Lower(bsbidx)-1] =\
        BoundarySendBuf[Pillar_Upper(bsbidx)-1] = *P;\
      npg[g] = dst - (Add_Pillar_Voxel(1) + 1);\
    }\
  }\
}
```

4.13.42 `sort_particles()`

`sort_particles()` The function `sort_particles()`, called solely from `exchange_particles4s()`, moves particles in `Particles[]` to `SendBuf[]` with sorting after non-position-aware particle transfer due to $Q_n + P_n^{\text{send}} > P_{lim}$ or helpand-helper reconfiguration. The function is almost logically equivalent to its level-4p counterpart shown in §4.10.37, but literally *different* from it significantly because of the followings.

- The caller is solely `exchange_particles4s()` regardless of the execution mode in the next step, and the index array for sorting is always `NOfPGridTotal[][][][]`.
- The role changing of `NOfPGridTotal[][][][]` from per-grid histogram to per-grid index has been done in `exchange_particles4s()` and thus the code for it is eliminated from this function.
- Since we cannot scan whole of $pbuf(p,s)$ because it includes halo particles not yet received, we scan its subset $pbuf_i(p,s)$ of non-halo ones referring to `InteriorParts`$[p][s]$ for its head index and size.
- Since we have to take care of particles in interior halo planes, we use `Sort_Particle()` to move a particle.

The function is given `nextmode` being 0 or 1 according to the mode in the next step is primary or secondary respectively for timing measurement, $p_n =$ `psnew` $\in \{0,1\}$ being 1 iff the local node n has secondary subdomain in the next step, and `stats` $\neq 0$ iff we have to start new timing measurement.

After starting the new timing measurement with the key `STATS_TB_SORT` if required by `stats` $\neq 0$, we scan all particles in $pbuf_i(p,s)$ (not $pbuf(p,s)$), whose head index and size are in `InteriorParts`$[p][s]$, for all $p \in \{0, p_n\}$ and $s \in [0, S)$. For each particle, we invoke `Sort_Particle()` for each of them to move it to `SendBuf[]` with sorting. Note that `NOfPGrid`$[p][s][]$ and `NOfPGridTotal`$[p][s][]$ are passed to the macro implicitly.

```
static void
sort_particles(const int nextmode, const int psnew, const int stats) {
  const int ns=nOfSpecies;
  struct S_particle *p;
  int ps, s, t, i;
  Decl_Grid_Info();

  if (stats) oh1_stats_time(STATS_TB_SORT, nextmode);
  for (ps=0,t=0; ps<=psnew; ps++) {
    for (s=0; s<ns; s++,t++) {
      dint *npg = NOfPGrid[ps][s], *npgt = NOfPGridTotal[ps][s];
      const int ips = InteriorParts[t].size;
      for (i=0,p=Particles+InteriorParts[t].head; i<ips; i++,p++)
        Sort_Particle(p);
    }
  }
}
```

4.13.43 `move_and_sort()`

`move_and_sort()` The function `move_and_sort()`, called solely from `exchange_particles4s()` when it finds $Q_n + P_n^{\text{send}} \leq P_{lim}$ without helpand-helper reconfiguration to mean we can move particles staying in and leaving from the local node together from `Particles[]` to `SendBuf[]` with sorting. It is given the following arguments; `nextmode` $= p'_n$ being 0 or 1 according to the mode in the next step being primary or secondary respectively; `psold` $= p_c = 1$ iff the local node had secondary particles; `psnew` $= p_n = 1$ iff it will have secondary particles; `oldp` $= n^p_{old}$ being the local node n's helpand in the last step; `nacc`$[2] = \{Q^n_n, Q_n\}$; and `stats` $\neq 0$ iff we have to start new timing measurement. The reason why this function needs to have n^p_{old} instead of n^p_{new} is same as what we discussed in §4.10.41.

The function is almost logically equivalent to its level-4p counterpart `move_and_sort_secondary()` shown in §4.10.44, but literally *different* from it somewhat because of the followings.

- The caller `exchange_particles4s()` works not only when we will be in secondary mode in the next step but also in primary mode, this function works with both modes as well. Therefore, the function is given `nextmode` $= p'_n$ but it does not affect the mechanism of the function almost at all, because data structures referred to in this function have values consistent with primary mode in the next step as discussed in §4.13.14 and thus the mechanism can depend almost only on the mode in the last step. Therefore, p_n is used only as an argument of `oh1_stats_time()` and `set_sendbuf_disps4s()`.
- Since this function is not called if we are in the transitional state of helpand-helper reconfiguration, the argument `trans` to indicate that is eliminated.
- The role changing of `NOfPGridTotal[][][]` from per-grid histogram to per-grid index has been done in `exchange_particles4s()` and thus the code for it is eliminated from this function.
- Since we have to take care of particles in interior halo planes, we pass $\eta = 1$ to the macro `Move_Or_Do()` as its last argument.

```
static void
move_and_sort(const int nextmode, const int psold, const int psnew,
              const int oldp, const int *nacc, const int stats) {
  const int me=myRank, ns=nOfSpecies, nn=nOfNodes, sbase=specBase;
  const int mysubdom[2] = {me, oldp},  ninj = nOfInjections;
  struct S_particle *p,  *rbb,  *sb = SendBuf + nacc[1];
  int *nofr;
  int ps, s, t, i;
  Decl_For_All_Grid();
  Decl_Grid_Info();
```

The first part for timing measurement, building the index array of $sbuf(p, s, m)$ in `NOfSend`$[p][s][m]$, and building the pointer array of $rbuf(p, s)$ in `RecvBufBases`$[p][s]$ has a few *differences* from that of the level-4p counterpart; p'_n is passed to `oh1_stats_time()` and `set_sendbuf_disps4s()`; 0 is passed to `set_sendbuf_disps4s()` through its second argument `trans` and is used as the first dimensional index of `RealSrcNeighbors`$[][]$ because we cannot be in transitional state of helpand-helper reconfiguration; and the role changing of `NOfPGridTotal`$[][][][]$ is eliiminated.

```
  if (stats) oh1_stats_time(STATS_TB_MOVE, nextmode);
  set_sendbuf_disps4s(nextmode, 0);
  for (ps=0,t=0,nofr=NOfRecv,rbb=Particles; ps<=psnew; ps++) {
    const int nnbr = RealSrcNeighbors[0][ps].n;
    const int *rnbr = RealSrcNeighbors[0][ps].nbor;
    for (s=0; s<ns; s++,t++,nofr+=nn) {
      int n, nrec;
      for (n=0,nrec=0; n<nnbr; n++)  nrec += nofr[rnbr[n]];
      RecvBufBases[t] = rbb;  rbb += nrec;
    }
  }
  RecvBufBases[t] = rbb;
```

The second part to scan all particles in $pbuf(p, s)$ for all $p \in \{0, p_c\}$ and $s \in [0, S)$, and the third part to scan all injected primary ($p = 0$) or secondary ($p = 1$) particles of species s are almost equivalent to those in the level-4p counterpart, *except* that we give $\eta = 1$ to the macro `Move_Or_Do()` to copy each particle in a vertical interior halo plane to `BoundarySendBuf`$[]$. As for the last part to let `primaryParts` and its shadow pointed by `secondaryBase` be Q_n^n, it is equivalent to the counterpart.

```
  for (ps=0,p=Particles,t=0; ps<=psold; ps++) {
    const int mysd = mysubdom[ps];
    for (s=0; s<ns; s++,t++) {
      dint *npg = NOfPGrid[ps][s],  *npgt = NOfPGridTotal[ps][s];
      const int itail = TotalP[t];
      for (i=0; i<itail; i++,p++)
        Move_Or_Do(p, ps, mysd, 1, (SendBuf[npgt[g]++]=*p), 1);
    }
  }
  for (i=0; i<ninj; i++,p++) {
    const int s = Particle_Spec(p->spec-sbase);
    const OH_nid_t nid = p->nid;
```

```
    const int ps = Secondary_Injected(nid) ? 1 : 0;
    const int mysd = mysubdom[ps];
    dint *npg = NOfPGrid[ps][s],  *npgt = NOfPGridTotal[ps][s];
    if (nid<0) continue;
    if (ps)  Primarize_Id_Only(p);
    Move_Or_Do(p, ps, mysd, 1, (SendBuf[npgt[g]++]=*p), 1);
  }
  primaryParts = *secondaryBase = nacc[0];
}
```

4.13.44 sort_received_particles()

sort_received_particles()

The function sort_received_particles() is solely called from exchange_particles4s(), to sort particles received from other nodes when it finds $Q_n + P_n^{\mathrm{send}} \leq P_{lim}$ without helpand-helper reconfiguration. The function is almost equivalent to its level-4p counterpart shown in §4.10.39, but has a *difference* that we use Sort_Particle() to move a particle from $rbuf(p, s)$ to take care of the case that it is in a vertical interior halo plane. This difference let us refer to NOfPGrid$[p][s][]$ in addition to NOfPGridTotal$[p][s][]$ so that they are passed to the macro implicitly.

```
static void
sort_received_particles(const int nextmode, const int psnew, const int stats) {
  const int ns=nOfSpecies;
  int ps, s;
  struct S_particle *p = Particles, **rbb = RecvBufBases+1;
  Decl_Grid_Info();

  if (stats) oh1_stats_time(STATS_TB_SORT, nextmode);
  for (ps=0; ps<=psnew; ps++) {
    for (s=0; s<ns; s++,rbb++) {
      dint *npg = NOfPGrid[ps][s], *npgt = NOfPGridTotal[ps][s];
      const struct S_particle *rbtail = *rbb;
      for (; p<rbtail; p++)  Sort_Particle(p);
    }
  }
}
```

4.13.45 set_sendbuf_disps4s()

set_sendbuf_disps4s()

The function set_sendbuf_disps4s(), called from move_to_sendbuf_4s() and move_and_sort() prior to their particle scan, is almost equivalent to its level-4p counterpart set_sendbuf_disps4p() shown in §4.10.45 to build the index array for SendBuf[] in NOfSend[][][] based on the sending counts in itself. The only one *difference* is that this function is given an additional argument nextmode $= p_n$ being 0 or 1 according to the mode in the next step being primary or secondary respectively, so that the function scans RealDstNeighbors$[t][p]$, where t is the argument trans, for $p \in \{0, p_n\}$ instead of $p \in \{0, 1\}$ in order to keep it from referring to meaningless values in RealDstNeighbors$[0][1]$ when $p_n = 0$ (with $t = 0$ definitely) because the callers work not only with $p_n = 1$ but also with $p_n = 0$.

```
static void
```

```
set_sendbuf_disps4s(const int nextmode, const int trans) {
  const int nn=nOfNodes, ns=nOfSpecies;
  int ps, s, i, np, *sbd;

  for (ps=0,sbd=NOfSend,np=0; ps<=nextmode; ps++) {
    const int n = RealDstNeighbors[trans][ps].n;
    const int *nbor = RealDstNeighbors[trans][ps].nbor;
    for (s=0; s<ns; s++,sbd+=nn) {
      for (i=0; i<n; i++) {
        const int nid = nbor[i];
        const int nsend = sbd[nid];
        sbd[nid] = np;  np += nsend;
      }
    }
  }
}
```

4.13.46 xfer_particles()

xfer_particles()

The function `xfer_particles()`, called solely from `exchange_particles4s()` regardless of $Q_n + P_n^{\text{send}} \leq P_{lim}$ or not, sends particles in the local node n to other nodes in `RealDstNeighbors`$[t][]$ and receives particles from other nodes in `RealSrcNeighbors`$[t][]$, where $t =$ `trans` $\in \{0, 1\}$ argument being 1 iff we have transitional state of helpand-helper configuration. The other arguments are as follows; `psnew` $= p_n \in \{0, 1\}$ being 1 iff n will have secondary subdomain in the next step and thus may have some secondary particles to receive; `nextmode` $= p'_n \in \{0, 1\}$ being 1 iff we will be in secondary mode in the next step and thus may have some particles to send to other nodes as their secondary particles; and `sbuf` is the pointer to `SendBuf`$[0]$ or `SendBuf`$[Q_n]$ to specify the location of $sbuf(0, 0, 0)$.

The function is very similar to its level-4p counterpart shown in §4.10.46, but have one small *difference* that this function has the argument `nextmode` $= p'_n$, because it and its caller works regardless of p'_n, to avoid referring to meaningless elements in `RealDstNeighbors`$[0][1]$ when $p'_n = 0$.

```
static void
xfer_particles(const int trans, const int psnew, const int nextmode,
               struct S_particle *sbuf) {
  const int nn=nOfNodes, ns=nOfSpecies;
  int ps, s, t, i, req, sdisp, *nofr, *nofs;
```

The first part to post `MPI_Irecv()` scanning `NOfRecv`$[p][s][m_i]$ for all $p \in \{0, p_n\}$, $s \in [0, S)$ and $m_i \in$ `RealSrcNeighbors`$[t][p]$`.nbor[]` is perfectly equivalent to the level-4p counterpart. However, the second part to post `MPI_Isend()` scanning `NOfSend`$[p][s][m_i]$ for all $p \in \{0, p'_n\}$, $s \in [0, S)$ and $m_i \in$ `RealDstNeighbors`$[t][p]$`.nbor[]` is a little bit *different* from the counterpart because $p \in \{0, p'_n\}$ instead of $p \in \{0, 1\}$ as discussed above. The last part to confirm the completions of all `MPI_Irecv()` and `MPI_Isend()` by `MPI_Waitall()` is perfectly equivalent to the counterpart again.

```
  for (ps=0,t=0,nofr=NOfRecv,req=0; ps<=psnew; ps++) {
    const int n = RealSrcNeighbors[trans][ps].n;
    const int *nbor = RealSrcNeighbors[trans][ps].nbor;
    for (s=0; s<ns; s++,t++,nofr+=nn) {
```

```
      struct S_particle *rbuf = RecvBufBases[t];
      for (i=0; i<n; i++) {
        const int nid = nbor[i];
        const int nrecv = nofr[nid];
        if (nrecv) {
          MPI_Irecv(rbuf, nrecv, T_Particle, nid, t, MCW, Requests+req++);
          rbuf += nrecv;
        }
      }
    }
  }
  for (ps=0,t=0,sdisp=0,nofs=NOfSend; ps<=nextmode; ps++) {
    const int n = RealDstNeighbors[trans][ps].n;
    const int *nbor = RealDstNeighbors[trans][ps].nbor;
    for (s=0; s<ns; s++,t++,nofs+=nn) {
      for (i=0; i<n; i++) {
        const int nid = nbor[i];
        const int sdnxt = nofs[nid];
        const int nsend = sdnxt - sdisp;
        nofs[nid] = 0;
        if (nsend) {
          MPI_Isend(sbuf+sdisp, nsend, T_Particle, nid, t, MCW,
                    Requests+req++);
        }
        sdisp = sdnxt;
      }
    }
  }
  MPI_Waitall(req, Requests, Statuses);
}
```

4.13.47 xfer_boundary_particles_v()

xfer_boundary_particles_v()

The *level-4s's own* function `xfer_boundary_particles_v()`, called solely from `exchange_particles4s()` but twice, sends particles in vertical interior halo planes of the local node n's subcuboids to other nodes share the planes as their vertical exterior halo planes, and receives particles in n's vertical exterior halo planes from the nodes. The function is given three arguments; `psnew` $= p_n \in \{0,1\}$ being 1 iff n has secondary subcuboid in the next step and thus may have some secondary particles received; `trans` $= t \in \{0,1\}$ being 1 iff we have transitional state of helpand-helper configuration and thus the shape of secondary subcuboid is determined by the subdomain of *new* parent; and `d` $= d \in \{0,1\}$ for the transfer along x ($d = 0$) or y ($d = 1$) axis.

```
static void
xfer_boundary_particles_v(const int psnew, const int trans, const int d) {
  const int ns=nOfSpecies;
  int vphi=d*2*2;
  const int vphead=VPlaneHead[vphi], vptail=VPlaneHead[vphi+2*2];
  int i, s, req=0, ps;
  struct S_vplane *vp;
  struct S_particle *p;
  Decl_For_All_Grid();
```

At first we examine the number of entries for d-th dimensional transfers in `VPlane[]` being $V = V_t - V_h$ is 0 where $\{V_h, V_t\}$ = `VPlaneHead`$[\{d, d{+}1\}][0][0]$ and, if so, return to the caller without doing nothing because n has neither subcuboids at all nor any neighbors along d-th axis in this case.

Otherwise, we have V pairs of $hbuf^s_v(d,p,\beta,m)$ and $hbuf^r_v(d,p,\beta,m)$ from/to which primary ($p = 0$) or secondary ($p = 1$) halo particles are sent/received to/from the node m = `VPlane`$[v]$`.nbor` being in d-th dimensional lower ($\beta = 0$) or upper ($\beta = 1$) neighbor family of the subdomain $n^p = \{n, parent(n)\}[p]$, whose neighbor index is $k = 3^2 + \nu_y \cdot 3^1 + \nu_x \cdot 3^0$ where $\nu_x = \{2\beta, 0\}[d]$ and $\nu_y = \{0, 2\beta\}[d]$. The serieses of both type buffers are commonly formed as a conceptual 8-dimensional array as a whole having elements $[d][p][\beta][z][s][y][x][i]$ for $d \in \{0,1\}$, $p \in \{0, p_n\}$, $\beta \in \{0,1\}$ $z \in [\zeta^l_p(n), \zeta^u_p(n))$, $s \in [0, S)$, $y \in [y^t_l(p,k), y^t_u(p,k))$, $x \in [x^t_l(p,k), x^t_u(p,k))$ and $i \in [0, \mathcal{P}_O(p, s, gidx(x,y,z)))$, where $y^t_b(p,k)$ and $x^t_b(p,k)$ ($t \in \{s,r\}$, $b \in \{l,u\}$) determine a vertical interior ($t = s$) or exterior ($t = r$) halo plane in it shared with the neighbor of the subdomain n^p. The buffer $hbuf^s_v(d,p,\beta,m)$ has the portion $[d][p][\beta][z][][][][]$ for $z \in [\zeta^l_p(n), \zeta^u_p(n)) \cap \zeta^l_{p'}(m)\zeta^u_{p'}(m)$ where $p' = 0$ if the k-th neighbor subdomain of n^p is primary one for m, or $p' = 1$ otherwise, i.e., secondary one. In the buffer $hbuf^s_v(d,p,\beta,m)$, all particles in the corresponding grid-voxels have been copied from `Particles[]` by `sort_particles()`, `move_and_sort()` or `sort_received_particles()`, and thus its correspondent $hbuf^s_v(d,p',1{-}\beta,n)$ in the node m has particles which $hbuf^r_v(d,p,\beta,m)$ should have.

Therefore, we scan all entries `VPlane`$[v]$ for all $v \in [V_h, V_t)$ to post `MPI_Irecv()` to receive particles to $hbuf^r_v(d,p,\beta,m)$. The location and the size of $hbuf^r_v(d,p,\beta,m)$ is specified by the elements of `VPlane`$[v]$; its index in `Particles[]` is `.rbuf` and the size is `.nrecv`. The tag of `MPI_Irecv()` is $\tau = p \cdot 3^D + k'$ = `VPlane`$[v]$`.rtag` where $k' = 3^D - 1 - k$, to make the combination (m, τ) unique in all receptions even when m occurs twice or more in `VPlane[].nbor` due to its membership in two families and/or periodic system boundaries.

```
if (vphead==vptail)  return;

for (i=vphead,vp=VPlane+vphead; i<vptail; i++,vp++) {
  const int nrecv = vp->nrecv;
  if (nrecv)
    MPI_Irecv(Particles+vp->rbuf, nrecv, T_Particle, vp->nbor, vp->rtag,
              MCW, Requests+req++);
}
```

Next, we scan all entries `VPlane`$[v]$ for all $v \in [V_h, V_t)$ again to post `MPI_Isend()` to send particles from $hbuf^s_v(d,p,\beta,m)$ to the node m as its (not n's) primary ($p' = 0$) or secondary ($p' = 1$) ones. The location and the size of $hbuf^s_v(d,p,\beta,m)$ is specified by the elements of `VPlane`$[v]$; its index in `BoundarySendBuf[]` is `.sbuf` and the size is `.nsend`. The tag of `MPI_Isend()` is $\tau = p' \cdot 3^D + k$ = `VPlane`$[v]$`.stag`, to make the combination (n, τ) unique in all receptions in m even when m occurs twice or more in `VPlane[].nbor`, and to match the tag of `MPI_Irecv()` posted by m because n is in $(3^D{-}1{-}k)$-th neighbor family of the subdomain $\{m, parent(m)\}[p']$.

```
for (i=vphead,vp=VPlane+vphead; i<vptail; i++,vp++) {
  const int nsend = vp->nsend;
  if (nsend)
    MPI_Isend(BoundarySendBuf+vp->sbuf, nsend, T_Particle, vp->nbor,
              vp->stag, MCW, Requests+req++);
}
```

Then, we return to the caller if neither `MPI_Irecv()` nor `MPI_Isend()` have been posted at all due to empty vertical halo planes. Othewise, we confirm the completion of all posted `MPI_Irecv()` and `MPI_Isend()` by `MPI_Waitall()` giving it the number of posts, and `Requests`[] and `Statuses`[] for each of posts.

Finally, we scan n's vertical exterior halo planes of primary ($p = 0$) subcuboid always and then secondary ($p = 1$) one if $p_n = 1$. The series of $hbuf_v^r(d, p, \beta, m)$, starting from `Particles`[`VPlane`[V_h]`.rbuf`], is determined by `make_bxfer_sched()` and `make_brecv_sched()` to form a conceptual 8-dimensional array to have elements $[d][p][\beta][z][s][y][x][i]$ as discuessed above where $y_b^r(p, k)$ are determined by

$$\texttt{Grid\_Exterior\_Boundary}(\nu_y,\ \delta_y(n^p),\ y_l^r(p,k),\ y_u^r(p,k){-}\delta_y(n^p))$$

and, if $d = 0$, $x_b^r(p, k)$ are determined by

$$\texttt{Grid\_Exterior\_Boundary}(\nu_x,\ \delta_x(n^p),\ x_l^r(p,k),\ x_u^r(p,k){-}\delta_x(n^p))$$

but $x_b^r(p, k) = \{-e^g,\ \delta_x(n^p) + e^g\}$ if $d = 1$ because south/north vertical exterior halo planes should have pillars at their west/east ends.

Therefore, we scan $\mathcal{P}_O(p, s, g) =$ `NOfPGridOut`$[p][s][g]$ where $g = gidx(x, y, z)$ and the particles in the corresponding grid-voxel in row-major order of the conceptual array to copy them to the region whose head is at `SendBuf`[`NOfPGridIndex`$[p][s][g]$], while $hbuf_v^r(d, p, \beta, m)$ are scanned sequentially from its top at `Particles`[`VPlane`[V_h]`.rbuf`]. We also let the `nid` element of the copied particles be -2 to indicate that they are not in n's subdomain.

One attention we have to pay in the scanning process is that, when $d = 0$, we will encounter a grid-voxel at g in an exterior pillar with `NOfPGrid`$[p][s][g] = (j{+}1){\cdot}2^{32}{+}\sigma \geq 2^{32}$ to mean the particles received for the grid-voxel should be *relayed* to a node m' in the south ($\beta' = 0$) or north ($\beta' = 1$) neighbor family of n^p and thus we have to copy them into $hbuf_v^s(1, p, \beta', m')$ starting from `BoundarySendBuf`$[j]$.

The other remark is that $hbuf_v^r(d, p, \beta, m)$ are definitely empty if the corresponding neighbor is inexistent. Since in this case it should be `VPlaneHead`$[v] =$ `VPlaneHead`$[v + 1]$ where $v = 4d{+}2p{+}\beta$ for $[d][p][\beta]$, we can skip the unncecessary scan of the vertical exterior halo plane when it holds.

```
if (req==0)  return;
MPI_Waitall(req, Requests, Statuses);

p = Particles + VPlane[vphead].rbuf;
for (ps=0; ps<=psnew; ps++) {
  const int psor2 = ps ? trans + 1 : 0;
  const int zl = ZBound[ps][OH_LOWER];
  const int zu = ZBound[ps][OH_UPPER] - GridDesc[psor2].z;
  int du;
  for (du=OH_LOWER; du<=OH_UPPER; du++,vphi++) {
    int ny;
    int xl, yl, xu, yu;
    if (VPlaneHead[vphi]==VPlaneHead[vphi+1])  continue;
    if (d==OH_DIM_X) {
      ny = 1;
      Grid_Exterior_Boundary(du<<1, GridDesc[psor2].x, xl, xu);
    } else {
```

```
        ny = du<<1;
        xl = -OH_PGRID_EXT;  xu = OH_PGRID_EXT;
      }
      Grid_Exterior_Boundary(ny, GridDesc[psor2].y, yl, yu);
      For_All_Grid_Z(psor2, xl, yl, zl, xu, yu, zu) {
        for (s=0; s<ns; s++) {
          dint *npg=NOfPGrid[ps][s];
          int *npgo=NOfPGridOut[ps][s], *npgi=NOfPGridIndex[ps][s];
          For_All_Grid_XY(psor2, xl, yl, xu, yu) {
            const int g = The_Grid(),  tail = npgi[g] + npgo[g];
            const dint dst = npg[g];
            int i;
            if (Is_Pillar_Voxel(dst)) {
              struct S_particle *q = p;
              int j = Pillar_Upper(dst) - 1;
              for (i=npgi[g]; i<tail; i++)  BoundarySendBuf[j++] = *q++;
            }
            for (i=npgi[g]; i<tail; i++) {
              SendBuf[i] = *p++;  SendBuf[i].nid = -2;
            }
          }
        }
      }
    }
  }
```

4.13.48 xfer_boundary_particles_h()

xfer_boundary_particles_h()

The *level-4s's own* function `xfer_boundary_particles_h()`, called solely from `exchange_particles4s()`, sends particles in horizontal interior halo planes of the local node n's subcuboids to other nodes sharing the planes as their horizontal exterior halo planes, and receives particles in n's horizontal exterior halo planes from the nodes. The function is given an arguments `psnew` $= p_n \in \{0, 1\}$ being 1 iff n has secondary subcuboid in the next step and thus may have some secondary particles to be sent and received.

```
static void
xfer_boundary_particles_h(const int psnew) {
  const int ns=nOfSpecies;
  int ps, ud, s, req=0;
```

We have pairs of $hbuf^s_h(p, \beta, s)$ and $hbuf^r_h(p, \beta, s)$ in `SendBuf[]` from/to which primary ($p = 0$) or secondary ($p = 1$) halo particles of species s are sent/received to/from the node $m =$ `HPlane`$[p][\beta]$`.nbor` having a subcuboid just below ($\beta = 0$) or above ($\beta = 1$) n's subcuboid in its subdomain $n^p = \{n, parentn\}[p]$. The locations of the buffers are given as the indices of `SendBuf[]` in `HPlane`$[p][s]$`.{sbuf,rbuf}`$[s]$ respectively, while their sizes are in `HPlane`$[p][s]$`.{nsend,nrecv}`$[s]$ resepctively. Note that each buffer is a part of $pbuf(p, s)$ in the next step and is for grid-voxels whose indices $g = gidx(x, y, z^\beta_t)$ are given as follows where z^β_s and z^β_r are for $hbuf^s_h(p, \beta, s)$ and $hbuf^r_h(p, \beta, s)$ respectively.

$$x \in [-e^g, \delta_x(n^p) + e^g), \quad y \in [-e^g, \delta_y(n^p) + e^g)$$
$$z^0_s = \zeta^l_p(n), \quad z^1_s = \zeta^u_p(n) - 1, \quad z^0_r = \zeta^l_p(n) - 1, \quad z^1_s = \zeta^u_p(n)$$

Therefore, what we basically have to do is simply sending the particles in $hbuf_h^s(p,\beta,s)$ to m and receiving those in m into $hbuf_h^r(p,\beta,s)$ by posting `MPI_Irecv()` at first and then `MPI_Isend()`. Note that halo particles in the intersection of horizontal exterior halo planes and vertical exterior halo planes are obtained by this simple communication, because m has already performed `xfer_boundary_particles_v()` to have those particles in its horizontal interior halo planes.

One caution is that the tag τ_r for `MPI_Irecv()` is $\texttt{HPlane}[p][\beta]\texttt{.rtag} = (p\cdot 3^D + k')S + s$ while τ_s for `MPI_Isend()` is $\texttt{HPlane}[p][\beta]\texttt{.stag} = (p'\cdot 3^D + k)S + s$, where $p' \in \{0,1\}$ is 1 iff the subcuboid of m is secondary one, $k = 2\beta\cdot 3^2 + 3^1 + 3^0$ and $k' = 3^D - 1 - k$, so that τ_r is unique for n's reception even when m occurs multiple times in $\texttt{HPlane}[][]\texttt{.nbor}$, and τ_s is so for m's reception as well and matches to n's sending. The other caution is that m can be `MPI_PROC_NULL` when n does not have any subcuboid, or its bottom/top surface is in a non-periodic system boundary.

```
for (ps=0; ps<=psnew; ps++) {
  for (ud=OH_LOWER; ud<=OH_UPPER; ud++) {
    struct S_hplane *hp = HPlane[ps] + ud;
    int *nrecv = hp->nrecv,  *rbuf = hp->rbuf;
    const int nbor = hp->nbor,  tag = hp->rtag;
    if (nbor!=MPI_PROC_NULL) {
      for (s=0; s<ns; s++) {
        if (nrecv[s])
          MPI_Irecv(SendBuf+rbuf[s], nrecv[s], T_Particle, nbor, tag+s, MCW,
                    Requests+req++);
      }
    }
  }
}
for (ps=0; ps<=psnew; ps++) {
  for (ud=OH_LOWER; ud<=OH_UPPER; ud++) {
    struct S_hplane *hp = HPlane[ps] + ud;
    int *nsend = hp->nsend,  *sbuf = hp->sbuf;
    const int nbor = hp->nbor,  tag = hp->stag;
    if (nbor!=MPI_PROC_NULL) {
      for (s=0; s<ns; s++) {
        if (nsend[s])
          MPI_Isend(SendBuf+sbuf[s], nsend[s], T_Particle, nbor, tag+s, MCW,
                    Requests+req++);
      }
    }
  }
}
```

Then, we return to the caller if neither `MPI_Irecv()` nor `MPI_Isend()` have been posted at all due to empty horizontal halo planes. Othewise, we confirm the completion of all posted `MPI_Irecv()` and `MPI_Isend()` by `MPI_Waitall()` giving it the number of posts, and `Requests[]` and `Statuses[]` for each of posts.

Finally, we scan $hbuf_h^r(p,s,\beta)$ for all $p \in \{0, p_n\}$, $s \in [0,S)$ and $\beta \in \{0,1\}$ to let the `nid` elements of received particles be -2 to indicate that they are not in n's subdomain.

```
if (req==0)  return;
MPI_Waitall(req, Requests, Statuses);

for (ps=0; ps<=psnew; ps++) {
```

```
    for (ud=OH_LOWER; ud<=OH_UPPER; ud++) {
      struct S_hplane *hp = HPlane[ps] + ud;
      int *nrecv = hp->nrecv,  *rbuf = hp->rbuf;
      if (hp->nbor!=MPI_PROC_NULL) {
        for (s=0; s<ns; s++) {
          const int tail = rbuf[s] + nrecv[s];
          int i;
          for (i=rbuf[s]; i<tail; i++)  SendBuf[i].nid = -2;
        }
      }
    }
  }
}
```

4.13.49 oh4s_exchange_border_data()

oh4s_exchange_border_data_() oh4s_exchange_border_data()

The API functions oh4s_exchange_border_data_() for Fortran and oh4s_exchange_border_data() for C perform inter-node communication for a particle-associated one-dimensional array buf, whose element type is given by type of MPI_Datatype, using send/receive buffers sbuf and rbuf so that its halo portions have the values computed by other nodes which responsible of particles with which the array elements are associated.

The function oh4s_exchange_border_data_() simply calls its counterpart oh4s_exchange_border_data() but type argument is converted into C's value by MPI_Type_f2c(). The function oh4s_exchange_border_data() is also simple because it just calls exchange_border_data_v() twice for west/east bound communications and then south/north ones, and then exchange_border_data_h() for horizontal halo plane, after obtaining the byte-size of each element by MPI_Type_get_extent().

```
void
oh4s_exchange_border_data_(void *buf, void *sbuf, void *rbuf, int *type) {
  oh4s_exchange_border_data(buf, sbuf, rbuf, MPI_Type_f2c(*type));
}
void
oh4s_exchange_border_data(void *buf, void *sbuf, void *rbuf,
                          MPI_Datatype type) {
  MPI_Aint esize, lb;

  MPI_Type_get_extent(type, &lb, &esize);
  exchange_border_data_v(buf, sbuf, rbuf, type, esize, 0);
  exchange_border_data_v(buf, sbuf, rbuf, type, esize, 1);
  exchange_border_data_h(buf, type, esize);
}
```

4.13.50 exchange_border_data_v()

exchange_border_data_v()

The function exchange_border_data_v(), called solely from oh4s_exchange_border_data() but twice, sends particle-associated data in d-th (d = d $\in \{0,1\}$) dimensional vertical interior halo planes of the one-dimensional array buf to other nodes after copying them to sbuf, and recieves those from the nodes into rbuf and then moves them into vertical exterior halo planes of buf. The data type and byte-size of each element is given by other arguments t = type and e = esize.

```
static void
exchange_border_data_v(void *buf, void *sbuf, void *rbuf, MPI_Datatype type,
                       const MPI_Aint esize, const int d) {
  char *b = (char*)buf, *sb = (char*)sbuf,  *rb = (char*)rbuf;
  const int ns=nOfSpecies, pscurr=RegionId[1]<0 ? 0 : 1, vphi=d*2*2;
  const int vphead=VPlaneHead[vphi], vptail=VPlaneHead[vphi+2*2];
  struct S_vplane *vp;
  int ps, s, i, req=0;
  Decl_For_All_Grid();
```

This function has an execution flow similar to the particle transfer function `xfer_boundary_particles_v()`, but has an additional phase to copy particle-associated data elements to the series of $hbuf_v^s(d,p,\beta,m)$ prior to the MPI communications for transfer of particle-associated data. That is, after confirming that $V_h \neq V_t$ where $\{V_h, V_t\} =$ `VPlaneHead`$[\{d, d{+}1\}][0][0]$ to mean the local node n has some subcuboids, we scan particle-associated data elements in grid-voxels of n's vertical interior halo planes, whose population and head index in `buf`[] are given by $\mathcal{P}_O(p,s,g) =$ `NOfPGridOut`$[p][s][g]$ and `NOfPGridIndex`$[p][s][g]$, in the nested loops for all $p \in \{0, p_c\}$, $\beta \in \{0,1\}$, $z \in [\zeta_p^l(n), \zeta_p^u(n))$, $s \in [0,S)$, $y \in [y_l, y_u)$ and $x \in [x_l, x_u)$ in this order, where $p_c \in \{0,1\}$ and $p_c = 1$ iff `RegionId`$[1] = parent(n) \geq 0$, $g = gidx(x,y,z)$, and y_l, y_u, x_l, x_u are given as follows with `Grid_Interior_Boundary()` and $n^p = \{n, parent(n)\}[p]$.

$$(y_l, y_u) = \begin{cases} (0,\ \delta_y(n^p)) & d = 0 \\ (0,\ e^g) & d = 1,\ \beta = 0 \\ (\delta_y(n^p) - e^g,\ \delta_y(n^p)) & d = 1,\ \beta = 1 \end{cases}$$

$$(x_l, x_u) = \begin{cases} (-e^g,\ \delta_x(n^p) + e^g) & d = 1 \\ (0,\ e^g) & d = 0,\ \beta = 0 \\ (\delta_x(n^p) - e^g,\ \delta_x(n^p)) & d = 0,\ \beta = 1 \end{cases}$$

Since we copy all particle-associated data for each grid-voxel we visit to `sbuf`[] from its head by `memcpy()`, giving it *byte-addresses* of the appropriate portions in `buf`[] and `sbuf`[] and total *byte-count* of the elements to be copied knowing the element size is e-byte, the buffer is formed as a conceptual 7-dimensional arrays of $[p][\beta][z][s][y][x][i]$ for p, ..., x shown above and $i \in [0, \mathcal{P}_O(p,s,gidx(x,y,z)))$. Therefore, `sbuf`[] should be formed as a series of buffers each of which has elements as many as $hbuf_v^s(d,p,\beta,m)$. However, we have to pay an attention that $hbuf_v^s(d,p,\beta,m_0)$ must not exist if the corresponding neighbor does not exist, or particles copied from the corresponding vertical interior halo plane will *push* down the buffers following $hbuf_v^s(d,p,\beta,m_0)$ to send incorrect particles to the nodes in the family of the folloiwng neighbors. Therefore we examine inexistence by checking `VPlaneHead`$[j] =$ `VPlaneHead`$[j+1]$ where $j = 4d + 2p + \beta$ for $[d][p][\beta]$.

```
  if (vphead == vptail)  return;

  for (ps=0,i=vphi; ps<=pscurr; ps++) {
    int du;
    const int zl = ZBound[ps][OH_LOWER];
    const int zu = ZBound[ps][OH_UPPER] - GridDesc[ps].z;
    for (du=OH_LOWER; du<=OH_UPPER; du++,i++) {
      int ny;
      int xl, yl, xu, yu;
```

```
    if (VPlaneHead[i]==VPlaneHead[i+1])  continue;
    if (d==OH_DIM_X) {
      ny = 1;
      Grid_Interior_Boundary(du<<1, GridDesc[ps].x, xl, xu);
    } else {
      ny = du<<1;
      xl = -OH_PGRID_EXT;  xu = OH_PGRID_EXT;
    }
    Grid_Interior_Boundary(ny, GridDesc[ps].y, yl, yu);
    For_All_Grid_Z(ps, xl, yl, zl, xu, yu, zu) {
      for (s=0; s<ns; s++) {
        int *npgo=NOfPGridOut[ps][s], *npgi=NOfPGridIndex[ps][s];
        For_All_Grid_XY(ps, xl, yl, xu, yu) {
          const int g = The_Grid(),  nbyte = npgo[g]*esize;
          memcpy(sb, b+npgi[g]*esize, nbyte);
          sb += nbyte;
        }
      }
    }
  }
}
```

Then, as done in `xfer_boundary_particles_v()`, we scan all entries `VPlane`$[v]$ for all $v \in [V_h, V_h)$ twice, at first to post `MPI_Irecv()` to receive particle-associated data into `rbuf`[] and then to post `MPI_Isend()` to send data from `sbuf`[]. One attention we have to pay is that `VPlane`[].{`sbuf`, `rbuf`} have indices of $hbuf_v^s(d,p,\beta,m)$ and $hbuf_v^r(d,p,\beta,m)$ for all $d \in \{0,1\}$, while `sbuf`[] and `rbuf`[] are for particular $d \in \{0,1\}$. In addition, we have to remember that the element size is e-byte and we assume `sbuf`[] and `rbuf`[] are `char`-type buffer so that their elements have any basic type or a series of types. Therefore, the head index of a buffer in `sbuf`[] or `rbuf`[] corresponding to $hbuf_v^s(d,p,\beta,m)$ or $hbuf_v^r(d,p,\beta,m)$ specified by `VPlane`$[v]$ is (`VPlane`$[v]$`.`b − `VPlane`$[V_h]$`.`b) $\cdot\, e$ where $b \in$ {`sbuf`, `rbuf`} correspondingly.

```
rb -= VPlane[vphead].rbuf * esize;
for (i=vphead,vp=VPlane+vphead; i<vptail; i++,vp++) {
  const int nrecv = vp->nrecv;
  if (nrecv)
    MPI_Irecv(rb+vp->rbuf*esize, nrecv, type, vp->nbor, vp->rtag, MCW,
              Requests+req++);
}
sb = (char*)sbuf - VPlane[vphead].sbuf * esize;
for (i=vphead,vp=VPlane+vphead; i<vptail; i++,vp++) {
  const int nsend = vp->nsend;
  if (nsend)
    MPI_Isend(sb+vp->sbuf*esize, nsend, type, vp->nbor, vp->stag, MCW,
              Requests+req++);
}
```

Finally, still as done in `xfer_boundary_particles_v()`, we scan n's vertical exterior halo planes to copy the contents in `rbuf`[] to `buf`[] referring to elements of `NOfPGridOut`[][][] and `NOfPGridIndex`[][][] corresponging to grid-voxels in the planes, after checking if we have posted some `MPI_Irecv()` or `MPI_Isend()` for early return and confirming their completion by `MPI_Waitall()` with `Requests`[] and `Statuses`[], and also after checking if the neighbor corresponding to each plane determined by d, p and β exists, i.e.,

VPlaneHead$[j] \neq$ VPlaneHead$[j+1]$ where $j = 4d + 2p + \beta$ for $[d][p][\beta]$. A difference from xfer_boundary_particles_v() is, besides the source and destination buffers, that the copy of an element is done by memcpy() again with byte-addresses and byte-count. The other difference is that we don't take care of received particle-associated data in exterior pillars because they are copied to corresponding region in buf[] by the first call with $d = 0$ and then copied to that in sbuf[] by the second call with $d = 1$ for *relaying* those data from west/east neighbors to south/north ones.

```
    if (req==0)  return;
    MPI_Waitall(req, Requests, Statuses);

    rb = (char*)rbuf;
    for (ps=0,i=vphi; ps<=pscurr; ps++) {
      const int zl = ZBound[ps][OH_LOWER];
      const int zu = ZBound[ps][OH_UPPER] - GridDesc[ps].z;
      int du;
      for (du=OH_LOWER; du<=OH_UPPER; du++,i++) {
        int ny;
        int xl, yl, xu, yu;
        if (VPlaneHead[i]==VPlaneHead[i+1])  continue;
        if (d==OH_DIM_X) {
          ny = 1;
          Grid_Exterior_Boundary(du<<1, GridDesc[ps].x, xl, xu);
        } else {
          ny = du<<1;
          xl = -OH_PGRID_EXT;  xu = OH_PGRID_EXT;
        }
        Grid_Exterior_Boundary(ny, GridDesc[ps].y, yl, yu);
        For_All_Grid_Z(ps, xl, yl, zl, xu, yu, zu) {
          for (s=0; s<ns; s++) {
            int *npgo=NOfPGridOut[ps][s], *npgi=NOfPGridIndex[ps][s];
            For_All_Grid_XY(ps, xl, yl, xu, yu) {
              const int g = The_Grid(),  nbyte = npgo[g]*esize;
              memcpy(b+npgi[g]*esize, rb, nbyte);
              rb += nbyte;
            }
          }
        }
      }
    }
  }
```

4.13.51 exchange_border_data_h()

exchange_border_data_h()

The function exchange_border_data_h(), called solely from oh4s_exchange_border_data(), sends particle-associated data in horizontal interior halo planes of the one-dimensional array buf to other nodes, and recieves those from nodes into horizontal exterior halo planes also in buf. The data type and byte-size of each element is given by other arguments $t =$ type and $e =$ esize.

```
  static void
  exchange_border_data_h(void *buf, MPI_Datatype type, const MPI_Aint esize) {
```

```
  char *b=(char*)buf;
  const int ns=nOfSpecies, pscurr=RegionId[1]<0 ? 0 : 1;
  int ps, ud, s, req=0;
  Decl_For_All_Grid();
```

This function is very similar to the halo particle transfer function `xfer_boundary_particles_h()` except that the followings; scanning range p for `HPlane`$[p][\beta]$ is given by `RegionId`[1]; send/receive buffer is `buf`[], which we assume a `char`-type buffer with elements of `MPI_Datatype` t having e-bytes for each, rather than `SendBuf`[]; and we do nothing after the confirmation of communication completion while `xfer_boundary_particles_h()` have to scan received particles to modify their `nid` elements.

```
  for (ps=0; ps<=pscurr; ps++) {
    for (ud=OH_LOWER; ud<=OH_UPPER; ud++) {
      struct S_hplane *hp = HPlane[ps] + ud;
      int *nrecv = hp->nrecv,  *rbuf = hp->rbuf;
      const int nbor = hp->nbor,  tag = hp->rtag;
      if (nbor!=MPI_PROC_NULL) {
        for (s=0; s<ns; s++) {
          if (nrecv[s])
            MPI_Irecv(b+rbuf[s]*esize, nrecv[s], type, nbor, tag+s, MCW,
                      Requests+req++);
        }
      }
    }
  }
  for (ps=0; ps<=pscurr; ps++) {
    for (ud=OH_LOWER; ud<=OH_UPPER; ud++) {
      struct S_hplane *hp = HPlane[ps] + ud;
      int *nsend = hp->nsend,  *sbuf = hp->sbuf;
      const int nbor = hp->nbor,  tag = hp->stag;
      if (nbor!=MPI_PROC_NULL) {
        for (s=0; s<ns; s++) {
          if (nsend[s])
            MPI_Isend(b+sbuf[s]*esize, nsend[s], type, nbor, tag+s, MCW,
                      Requests+req++);
        }
      }
    }
  }
  if (req)  MPI_Waitall(req, Requests, Statuses);
}
```

4.13.52 Macro `Check_Particle_Location()`

`Check_Particle_Location()` The macro `Check_Particle_Location`(π, p, s, S, i) is perfectly equivalent to its level-4p counterpart shown in §4.10.47.

```
#ifndef OH_NO_CHECK
#define Check_Particle_Location(P, PS, S, NS, INJ) {\
  const int t = (PS) ? (S)+(NS) : (S);\
  const int pidx = (P) - Particles;\
```

```
  if ((PS)<0 || (PS)>1 || (S)<0 || (S)>=(NS) ||\
      (PbufIndex && ((INJ) ?\
                     (((PS)&&RegionId[1]<0) ||\
                         pidx>=totalParts+nOfInjections) :\
                     (pidx<PbufIndex[t] || pidx>=PbufIndex[t+1]))))\
    local_errstop("'part' argument pointing %c%d%c of the particle buffer is "\
                  "inconsistent with 'ps'=%d and 's'=%d",\
                  specBase?'(':'[', pidx+specBase,\
                  specBase?')':']', PS, (S)+specBase);\
}
#else
#define Check_Particle_Location(P, PS, S, NS, INJ)
#endif
```

4.13.53 Macros Map_Particle_To_Neighbor() and Adjust_Neighbor_Grid()

Map_Particle_To_Neighbor()
Adjust_Neighbor_Grid()

The macros Map_Particle_To_Neighbor($\pi, X_d, d, m, k, 3^d, \delta_d(m), x_d, g$) and Adjust_Neighbor_Grid(x_d, m, d) are perfectly equivalent to their level-4p counterparts shown in §4.10.48.

```
#define Map_Particle_To_Neighbor(P, XYZ, DIM, MYSD, K, INC, UB, G, IDX) {\
  const double xyz = XYZ;\
  const double gsize = Grid[DIM].gsize;\
  const double lb = Grid[DIM].fcoord[OH_LOWER];\
  const double gf =\
    (G = (xyz-lb)*Grid[DIM].rgsize + Grid[DIM].coord[OH_LOWER]) * gsize;\
  if (gf>xyz) G--;\
  else if (gf+gsize<=xyz) G++;\
  G -= SubDomains[MYSD][DIM][OH_LOWER];  IDX += G;\
  if (G<0) {\
    K -= INC;\
    if (xyz<lb) {\
      if (Boundaries[MYSD][DIM][OH_LOWER]) { P->nid = -1;  return(-1); }\
      XYZ += Grid[DIM].fcoord[OH_UPPER] - lb;\
    }\
    if (G<-OH_PGRID_EXT)  K = -OH_NEIGHBORS;\
  } else if (G>=UB) {\
    double ub = Grid[DIM].fcoord[OH_UPPER];\
    K += INC;\
    if (xyz>=ub) {\
      if (Boundaries[MYSD][DIM][OH_UPPER]) { P->nid = -1;  return(-1); }\
      XYZ -= ub - lb;\
    }\
    G-=UB;\
    if (G>=OH_PGRID_EXT)  K = -OH_NEIGHBORS;\
  }\
}
#define Adjust_Neighbor_Grid(G, N, DIM)\
  if (G<0) G += SubDomains[N][DIM][OH_UPPER]-SubDomains[N][DIM][OH_LOWER];
```

4.13.54 oh4s_map_particle_to_neighbor()

oh4s_map_particle_to_neighbor_()
oh4s_map_particle_to_neighbor()

The API functions oh4s_map_particle_to_neighbor_() for Fortran and oh4s_map_particle_to_neighbor() for C are perfectly equivalent to their level-4p counterparts oh4p_map_particle_to_neighbor[_]() shown in §4.10.49 except for their and callees' names.

```
int
oh4s_map_particle_to_neighbor_(struct S_particle *part, const int *ps,
                               const int *s) {
  return(oh4s_map_particle_to_neighbor(part, *ps, *s-1));
}
int
oh4s_map_particle_to_neighbor(struct S_particle *part, const int ps,
                              const int s) {
  const int ns = nOfSpecies,  inj = part>=Particles+totalParts;
  int x, y, z, w, d, dw, mysd;
  const int psnn = ps ? (s+nOfSpecies)*nOfNodes : s*nOfNodes;
  int k = OH_NBR_SELF,  idx = 0;
  int gz, gy, gx;
  int sd;
  Decl_Grid_Info();

  Check_Particle_Location(part, ps, s, ns, inj);
  x = GridDesc[ps].x;  y = GridDesc[ps].y;  z = GridDesc[ps].z;
  w = GridDesc[ps].w;  d = GridDesc[ps].d;  dw = GridDesc[ps].dw;
  mysd = RegionId[ps];
  Do_Z(Map_Particle_To_Neighbor(part, part->z, OH_DIM_Z, mysd, k, 9, z, gz,
                                idx));
  Do_Z(idx *= d);
  Do_Y(Map_Particle_To_Neighbor(part, part->y, OH_DIM_Y, mysd, k, 3, y, gy,
                                idx));
  Do_Y(idx *= w);
  Map_Particle_To_Neighbor(part, part->x, OH_DIM_X, mysd, k, 1, x, gx, idx);

  if (k==OH_NBR_SELF) {
    NOfPGrid[ps][s][idx]++;
    NOfPLocal[psnn+mysd]++;
    part->nid = Combine_Subdom_Pos(k, idx);
    if (inj) {
      if (ps) {
        InjectedParticles[ns+s]++;  Secondarize_Id(part);
      } else {
        InjectedParticles[s]++;
      }
    }
    return(mysd);
  } else if (k<0)
    return(oh4s_map_particle_to_subdomain(part, ps, s));
  sd = AbsNeighbors[ps][k];
  if (sd>=nOfNodes) {
    part->nid = -1;  return(-1);
  }
  Adjust_Neighbor_Grid(gx, sd, OH_DIM_X);
```

```
  Do_Y(Adjust_Neighbor_Grid(gy, sd, OH_DIM_Y));
  Do_Z(Adjust_Neighbor_Grid(gz, sd, OH_DIM_Z));
  NOfPLocal[psnn+sd]++;

  if (sd==mysd) {
    idx = Coord_To_Index(gx, gy, gz, w, dw);
    NOfPGrid[ps][s][idx]++;
    part->nid = Combine_Subdom_Pos(OH_NBR_SELF, idx);
    if (inj)  InjectedParticles[ps ? ns+s : s]++;
  } else {
    NOfPGrid[ps][s][idx]++;
    part->nid = Combine_Subdom_Pos(k, Coord_To_Index(gx, gy, gz, w, dw));
  }
  if (inj && ps)  Secondarize_Id(part);
  return(sd);
}
```

4.13.55 Macros Map_To_Grid, Map_Particle_To_Subdomain() and Local_Coordinate()

Map_To_Grid()
Map_Particle_To_Subdomain()
Local_Coordinate()

The macros Map_Particle_To_Subdomain(x_d, d, π_d), Map_To_Grid$(\pi, X^*_d, X_d, d, x_d, x'_d)$ and Local_Coordinate$(m, n', x_d, x'_d, d, k, 3^d, a)$ are perfectly equivalent to their counterparts shown in §4.10.50.

```
#define Map_To_Grid(P, PXYZ, XYZ, DIM, GG, LG) {\
  const double gsize = Grid[DIM].gsize;\
  const double lb = Grid[DIM].fcoord[OH_LOWER];\
  const double ub = Grid[DIM].fcoord[OH_UPPER];\
  double gf;\
  XYZ = PXYZ;\
  LG = 0;\
  if (XYZ<lb) {\
    if (BoundaryCondition[DIM][OH_LOWER]) { P->nid = -1;  return(-1); }\
    XYZ += (ub - lb);  PXYZ = XYZ;\
    LG = Grid[DIM].coord[OH_LOWER] - Grid[DIM].coord[OH_UPPER];\
  }\
  else if (XYZ>=ub) {\
    if (BoundaryCondition[DIM][OH_UPPER]) { P->nid = -1;  return(-1); }\
    XYZ -= (ub - lb);  PXYZ = XYZ;\
    LG = Grid[DIM].coord[OH_UPPER] - Grid[DIM].coord[OH_LOWER];\
  }\
  GG = (XYZ-lb)*Grid[DIM].rgsize + Grid[DIM].coord[OH_LOWER];\
  gf = GG * gsize;\
  if (gf>XYZ) GG--;\
  else if (gf+gsize<=XYZ) GG++;\
  LG += GG;\
}
#define Map_Particle_To_Subdomain(XYZ, DIM, SDOM) {\
  double thresh = Grid[DIM].light.thresh;\
  if (XYZ<thresh)\
    SDOM = (XYZ - Grid[DIM].coord[OH_LOWER]) / Grid[DIM].light.size;\
  else\
    SDOM = (XYZ - thresh)/ (Grid[DIM].light.size + 1) + Grid[DIM].light.n;\
```

```
}
#define Local_Coordinate(N, MYSD, GG, LG, DIM, K, INC, AA) {\
  GG -= SubDomains[N][DIM][OH_LOWER];\
  if (N==MYSD)  LG = GG;\
  else {\
    const int ub = SubDomains[MYSD][DIM][OH_UPPER];\
    if (LG>=ub+OH_PGRID_EXT)  AA = 1;\
    else {\
      const int inc = LG<ub ? 0 : INC;\
      LG -= SubDomains[MYSD][DIM][OH_LOWER];\
      if (LG<-OH_PGRID_EXT)  AA = 1;\
      k += LG<0 ? -INC : inc;\
    }\
  }\
}
```

4.13.56 oh4s_map_particle_to_subdomain()

oh4s_map_particle_to_subdomain_()
oh4s_map_particle_to_subdomain()

The API functions `oh4s_map_particle_to_subdomain_()` for Fortran and `oh4s_map_particle_to_subdomain()` for C are perfectly equivalent to their level-4p counterparts `oh4p_map_particle_to_subdomain[_]()` shown in §4.10.51 except for their names.

```
int
oh4s_map_particle_to_subdomain_(struct S_particle *part, const int *ps,
                                const int *s) {
  return(oh4s_map_particle_to_subdomain(part, *ps, *s-1));
}
int
oh4s_map_particle_to_subdomain(struct S_particle *part, const int ps,
                               const int s) {
  const int ns = nOfSpecies,  inj = part>=Particles+totalParts;
  const int nx  = Grid[OH_DIM_X].n;
  const int nxy = If_Dim(OH_DIM_Y, nx*Grid[OH_DIM_Y].n, 0);
  const int t = ps ? ns + s : s;
  int w, dw, mysd;
  int sd;
  double x, y, z;
  int px, py, pz;
  int gx, gy, gz;
  int lx, ly, lz;
  int k = OH_NBR_SELF,  aacc = 0;
  Decl_Grid_Info();

  Check_Particle_Location(part, ps, s, ns, inj);
  w = GridDesc[ps].w;  dw = GridDesc[ps].dw;  mysd = RegionId[ps];
  Map_To_Grid(part, part->x, x, OH_DIM_X, gx, lx);
  Do_Y(Map_To_Grid(part, part->y, y, OH_DIM_Y, gy, ly));
  Do_Z(Map_To_Grid(part, part->z, z, OH_DIM_Z, gz, lz));
  if (SubDomainDesc) {
    sd = map_irregular_subdomain(x, If_Dim(OH_DIM_Y ,y, 0),
                                    If_Dim(OH_DIM_Z, z, 0));
    if (sd<0) { part->nid = -1;  return(-1); }
  } else {
```

```
    Map_Particle_To_Subdomain(gx, OH_DIM_X, px);
    Do_Y(Map_Particle_To_Subdomain(gy, OH_DIM_Y, py));
    Do_Z(Map_Particle_To_Subdomain(gz, OH_DIM_Z, pz));
    sd = Coord_To_Index(px, py, pz, nx, nxy);
  }
  Local_Coordinate(sd, mysd, gx, lx, OH_DIM_X, k, 1, aacc);
  Do_Y(Local_Coordinate(sd, mysd, gy, ly, OH_DIM_Y, k, 3, aacc));
  Do_Z(Local_Coordinate(sd, mysd, gz, lz, OH_DIM_Z, k, 9, aacc));
  NOfPLocal[t*nOfNodes+sd]++;
  if (aacc) {
    currMode = Mode_Set_Any(currMode);
    part->nid = Combine_Subdom_Pos(sd+OH_NEIGHBORS,
                                   Coord_To_Index(gx, gy, gz, w, dw));
  } else {
    NOfPGrid[ps][s][Coord_To_Index(lx, ly, lz, w, dw)]++;
    part->nid = Combine_Subdom_Pos(k, Coord_To_Index(gx, gy, gz, w, dw));
  }
  if (inj) {
    if (sd==mysd)  InjectedParticles[t]++;
    if (ps)  Secondarize_Id(part);
  }
  return(sd);
}
```

4.13.57 oh4s_inject_particle()

oh4s_inject_particle_()
oh4s_inject_particle()

The API functions `oh4s_inject_particle_()` for Fortran and `oh4s_inject_particle()` for C are perfectly equivalent to their level-4p counterparts `oh4p_inject_particle[_]()` shown in §4.10.52 except for their and callees' names.

```
int
oh4s_inject_particle_(const struct S_particle *part, const int *ps) {
  return(oh4s_inject_particle(part, *ps));
}
int
oh4s_inject_particle(const struct S_particle *part, const int ps) {
  const int ns = nOfSpecies;
  int inj = totalParts + nOfInjections++;
  struct S_particle *p = Particles + inj;
  int s = Particle_Spec(part->spec - specBase);
  int sd;

#ifndef OH_HAS_SPEC
  if (ns!=1)
    local_errstop("particles cannot be injected when S_particle does not "
                  "have 'spec' element and you have two or more species");
#endif
  if (inj>=nOfLocalPLimit)
    local_errstop("injection causes local particle buffer overflow");
  *p = *part;
  sd = oh4s_map_particle_to_neighbor(p, ps, s);
  if (sd<0)  nOfInjections--;
  return(sd);
```

```
}
```

4.13.58 oh4s_remove_mapped_particle()

oh4s_remove_mapped_particle_()
oh4s_remove_mapped_particle()

The API functions oh4s_remove_mapped_particle_() for Fortran and oh4s_remove_mapped_particle() for C are perfectly equivalent to their level-4p counterparts oh4p_remove_mapped_particle[_]() shown in §4.10.53 except for their names.

```
void
oh4s_remove_mapped_particle_(struct S_particle *part, const int *ps,
                             const int *s) {
  oh4s_remove_mapped_particle(part, *ps, *s-1);
}
void
oh4s_remove_mapped_particle(struct S_particle *part, const int ps,
                            const int s) {
  const int nn = nOfNodes, ns = nOfSpecies, inj = part>=Particles+totalParts;
  OH_nid_t nid = part->nid;
  int sd, g, psreal=ps, mysd, t;
  Decl_Grid_Info();

  Check_Particle_Location(part, psreal, s, ns, inj);
  if (nid<0)  return;
  sd = Subdomain_Id(nid, psreal);
  g = Grid_Position(nid);
  if (sd>=nn) {
    psreal = 1;  Primarize_Id(part, sd);  nid = part->nid;
  }
  mysd = RegionId[psreal];
  part->nid = -1;
  t = psreal ? ns+s : s;
  NOfPLocal[t*nn+sd]--;
  if (inj && sd==mysd)  InjectedParticles[t]--;
  if (Mode_Acc(currMode))  return;
  if (sd!=mysd)  g = Local_Grid_Position(g, nid, psreal);
  NOfPGrid[psreal][s][g]--;
}
```

4.13.59 oh4s_remap_particle_to_neighbor()

oh4s_remap_particle_to_neighbor_()
oh4s_remap_particle_to_neighbor()

The API functions oh4s_remap_particle_to_neighbor_() for Fortran and oh4s_remap_particle_to_neighbor() for C are perfectly equivalent to their level-4p counterparts oh4p_remap_particle_to_neighbor[_]() shown in §4.10.54 except for their and callees' names.

```
int
oh4s_remap_particle_to_neighbor_(struct S_particle *part, const int *ps,
                                 const int *s) {
  return(oh4s_remap_particle_to_neighbor(part, *ps, *s-1));
}
int
oh4s_remap_particle_to_neighbor(struct S_particle *part, const int ps,
```

```
                                  const int s) {
  oh4s_remove_mapped_particle(part, ps, s);
  return(oh4s_map_particle_to_neighbor(part, ps, s));
}
```

4.13.60 oh4s_remap_particle_to_subdomain()

oh4s_remap_particle_to_subdomain_()
oh4s_remap_particle_to_subdomain()

The API functions oh4s_remap_particle_to_subdomain_() for Fortran and oh4s_remap_particle_to_subdomain() for C are perfectly equivalent to their level-4p counterparts oh4p_remap_particle_to_subdomain[_]() shown in §4.10.55 except for their and callees' names.

```
int
oh4s_remap_particle_to_subdomain_(struct S_particle *part, const int *ps,
                                  const int *s) {
  return(oh4s_remap_particle_to_subdomain(part, *ps, *s-1));
}
int
oh4s_remap_particle_to_subdomain(struct S_particle *part, const int ps,
                                 const int s) {
  oh4s_remove_mapped_particle(part, ps, s);
  return(oh4s_map_particle_to_subdomain(part, ps, s));
}
```

4.14 Sample `make` Files

As discussed in §3.14, OhHelp distribution just has sample make files for Fortran and C simulators namely samplef.mk and samplec.mk shown in the following subsections.

4.14.1 samplef.mk for Fortran

The sample make file for Fortran coded simulators, samplef.mk, at first declares that the Fortran compiler `FC` is used for linking, and then declares the following sets of files.

- `COMMONHDRS` = {oh_config.h, oh_stats.h}
  C header files commonly `#include`d into library and simulator header/source files.
- `OHHDRS` = {ohhelp1.h, ohhelp2.h, ohhelp3.h, ohhelp4p.h, oh_part.h}
  C header files `#include`d into library header/source files.
- `OHBOJS` = {ohhelp1.o, ohhelp2.o, ohhelp3.o, ohhelp3.o}
  C object files compiled from library sources, ohhelp1.c, ohhelp2.c, ohhelp3.c and ohhelp4p.c.
- `FHDRS` = {ohhelp_f.h}
  Fortran header file `#include`d into Fortran simulator source files.
- `FMODS` = {oh_type.o, oh_mod1.o, oh_mod2.o, oh_mod3.o, oh_mod4p.o}
  Fortran object files compliled from Fortran module files `use`d in Fortran module/ source files, oh_type.F90, oh_mod1.F90, oh_mod2.F90, oh_mod3.F90 and oh_mod4p.F90.
- `FOBJS` = {simulator.o, sample.o}
  Fortran object files compiled from Fortran simulator source files, simulator.F90 and sample.F90.
- `OBJS` = `FOBJS` ∪ `FMODS` ∪ `OHOBJS`
  All object files to be linked.

```
LINKER          = $(FC)

COMMONHDRS      = oh_config.h oh_stats.h
OHHDRS          = ohhelp1.h ohhelp2.h ohhelp3.h ohhelp4p.h oh_part.h
OHOBJS          = ohhelp1.o ohhelp2.o ohhelp3.o ohhelp4p.o

FHDRS           = ohhelp_f.h
FMODS           = oh_type.o oh_mod1.o oh_mod2.o oh_mod3.o oh_mod4p.o

FOBJS           = simulator.o sample.o

OBJS            = $(FOBJS) $(FMODS) $(OHOBJS)
```

Then the make file defines the dependency shown in the Table 2 in §3.14, and the pseudo-dependency for cleaning the working directory by removing object files and `.mod` files.

```
simulator:      $(OBJS)
                $(LINKER) $(FFLAGS) $(LDFLAGS) $(OBJS) -o $@
```

```
$(FOBJS):%.o:   %.F90 $(FMODS) $(FHDRS) $(COMMONHDRS)
                $(FC) $(FFLAGS) -c $< -o $@
simulator.o:    sample.o
$(FMODS):%.o:   %.F90 $(COMMONHDRS)
                $(FC) $(FFLAGS) -c $< -o $@
oh_mod1.o:      oh_type.o
oh_mod2.o:      oh_mod1.o
oh_mod3.o:      oh_mod2.o
oh_mod4p.o:     oh_mod3.o
$(OHOBJS):%.o:  %.c $(COMMONHDRS) $(OHHDRS)
                $(CC) $(CFLAGS) -c $< -o $@

clean:;
                rm *.o *.mod
```

4.14.2 samplec.mk for C

The sample make file for C coded simulators, samplec.mk, at first declares that the C compiler CC is used for linking, and then declares the following sets of files.

- COMMONHDRS = {oh_config.h, oh_part.h, oh_stats.h}
  C header files commonly #included into library and simulator header/source files.
- OHHDRS = {ohhelp1.h, ohhelp2.h, ohhelp3.h, ohhelp4p.h}
  C header files #included into library header/source files.
- OHBOJS = {ohhelp1.o, ohhelp2.o, ohhelp3.o, ohhelp4p.o}
  C object files compiled from library sources, ohhelp1.c, ohhelp2.c, ohhelp3.c and ohhelp4p.c.
- CHDRS = {ohhelp_c.h}
  C header file #included into C simulator source files.
- COBJS = {simulator.o, sample.o}
  C object files compiled from C simulator source files, simulator.c and sample.c.
- OBJS = COBJS ∪ OHOBJS
  All object files to be linked.

```
LINKER          = $(CC)

COMMONHDRS      = oh_config.h oh_part.h oh_stats.h
OHHDRS          = ohhelp1.h ohhelp2.h ohhelp3.h ohhelp4p.h
OHOBJS          = ohhelp1.o ohhelp2.o ohhelp3.o ohhelp4p.o

CHDRS           = ohhelp_c.h
COBJS           = simulator.o sample.o

OBJS            = $(COBJS) $(OHOBJS)
```

Then the make file defines the dependency shown in the Table 3 in §3.14, and the pseudo-dependency for cleaning the working directory by removing object files.

```
simulator:      $(OBJS)
                $(LINKER) $(CFLAGS) $(LDFLAGS) $(OBJS) -o $@

$(COBJS):%.o:   %.c $(COMMONHDRS) $(CHDRS)
                $(CC) $(CFLAGS) -c $< -o $@
$(OHOBJS):%.o:  %.c $(COMMONHDRS) $(OHHDRS)
                $(CC) $(CFLAGS) -c $< -o $@

clean:;
                rm *.o
```

Acknowledgments

The author thanks to Prof. Yoshiharu Omura and Prof. Hideyuki Usui who triggered the research work on OhHelp and have patiently supported the author during his snailish development. He also thanks to Dr. Yohei Miyake who kindly gave the author the first target simulator named 3D-Kempo, and to Dr. Hitoshi Sakagami who motivated the author to form his OhHelp program as a library package.

Index

Italicized number refers to the page where the specification and usage of corresponding entry is described, while underlined is for the implementation of the entry.

Symbols

Γ 229, 326, 327, 328, 359, 411, 416, 431, 435, 474, 524
Δ_d^- . . 271, 272, 289, 290, 294, 310, 314, 433
Δ_d^l 50, *51*, *58*, 59, 66, 80, 271, 284, 289–294, 310, 312, 314, 428, 432, 433
Δ_d^u . . 50, *51*, *58*, 59, 66, 80, 271, 284, 289, 290, 292–294, 310, 312, 314, 428, 432
Π_d 29, *30*, 32, *33*, 50, 51, 58, 110, 121, 147, 160, 165, 271, 289–292, 294, 315, 435, 450
Π_d^- 271, 289–292, 294, 314, 433
$\Phi_d(f)$ 274, 275, 297, 300, 303, 304, 306–309, 334, 403, 454
α *11*, 27, 30, 45, 78, 88, 89, 136, 137, 159, 162, 174, 268, 332, 452
γ_d *64*, 80, 85, 270, 271, 272, 277, 288–290, 294, 310, 312–315, 317, 318, 428, 432
$\delta_d(n)$ *84*, 86, 88, 330, 331, 341, 353, 363, 372, 373, 376, 381, 386, 387, 390, 401, 402, 410, 412, 428–431, 434, 440, 446, 447, 457, 468, 475, 481, 491, 493, 494, 497, 500, 502, 504, 506–513, 521, 523, 539, 540, 543, 547
$\delta_d^l(n)$ *50*, 56, *57*, 58, 62, 64, 84, 270, 271, 272, 275, 284, 289, 291–296, 301, 303–305, 310, 312, 313, 315, 317, 318, 330, 331, 341, 387, 401, 428, 429, 433, 446, 512
$\delta_d^{\max}$. *88*, 89, 271, 274, 290, 293, 294, 300, 310, 317, 318, 326, 331, 353, 359, 361, 373, 381, 402, 403, 430, 435, 443, 447, 450, 452, 469, 473–475, 479, 494, 514
$\delta_d^{\min}$ 271, 289–294, 310, 314, 315, 433
$\delta_d^u(n)$ *50*, 56, *57*, 58, 62, 64, 84, 270, 271, 272, 275, 284, 289, 291–296, 301, 303–305, 310, 312, 313, 315, 317, 318, 330, 331, 341, 387, 401, 429, 446, 512
$\zeta_p^l(n)$ *84*, 90, 441, 447–449, 469, 506–510, 521–524, 538, 540, 543
$\zeta_p^u(n)$ *84*, 90, 441, 447–449, 469, 506–510, 521–524, 538, 540, 543
ν_d . . *28*, *32*, 146, 147, 160, 165, 288, 313, 341, 386, 387, 391, 401, 428, 476, 504, 506, 512, 518, 520, 521, 523, 538, 539
π_d *28*, 30, *32*, 50, 51, 58, 146, 147, 160, 165, 288, 291, 292, 313–315, 433, 435, 549
ρ_n 181, 182, 185, 186
σ_d *53*, 54, 55, *59*, 60, 110, 111, 115, 118, 119, 122, 125, 127, 128
$\phi_d^l(f)$ 19, 20, *55*, 56, *61*, 62, 72, 77, 79, 91, 284, 298, 300
$\phi_d^u(f)$ 19, 20, *55*, 56, *61*, 62, 72, 77, 79, 91, 284, 298, 300

A

`AbsNeighbors` 229, 230, 326, 328, 347, 360, 367, 369, 402, 415, 431, 443, 463, 477, 484, 486, 513, 548
`accMode` 135, 162, 174, 211
`add_boundary_curr()` 108, 116, 117, 120, 126, 127
`add_boundary_current()` 108, 112, 115, 117, 120, 123, 125, 127, 373, 494
`Add_Pillar_Voxel()` . 519, 520, 522, 525, 532
`add_population()` 331, 346, 353, 372, 373, 451, 462, 468, 493, 494, 495
`Adjacent` 273, 288, 294, 321
`adjust_field_descriptor()` . . 334, 347, 359, 401, 403, 454, 463, 474, 512, 514
`Adjust_Neighbor_Grid()` . 429, 431, 547, 548
`Adjust_Subdomain()` 270, 314, 315, 433
`allocate()` 19, 56, 57, 110
`Allocate_NOfPGrid()` 329, 331, 350, 352, 355, 358, 361, 446, 451, 466, 467, 470, 474, 479
alternative secondary receiving block (of `CommList`) *143*, 144, 244, 254, 255, 263, *336*, 338, 347, 375, 378, 383, 385, 389, *455*, 456, 457, 463, 488, 496, 498, 500, 502, 507, 518
alternative secondary sending block (of `CommList`) . . . *455*, 456, 457, 463, 488, 496, 498, 500, 502, 518, 520, 523
`AltSecRLIndex` . . 455, 457, 488, 498, 502, 518
`AltSecRList` 336, 338, 378, 385, 455, 456, 488, 502, 518
anywhere accommodation 26, *37*, 38, 134, 135, 152, *173*, 184, 189, 190, 195, 206, 211, 228, 239, 242, 244, 254, 256, 324, 327–331, 334, 354, 362, 364, 366, 367, 369, 370, 376, 382, 391, 401, 402, 404, 410, 411, 416, 429, 434, 435, 445, 454, 456, 469, 483, 486–488, 491, 492, 496, 497, 512–514
`assign_particles()` 137, 139, 157, 182, 185, 188

B

B (number of boundary condition types) . . . *52*, 55, 56, *59*, 60, 62, 273, 276, 284, 288–290, 293, 297–299, 334, 454

$\boldsymbol{B}$ (magnetic field) 18, 19, 108, 112, 118, 123, 127

B_n *15*, 16, 182, 183, 186–188

`bcond` *52*, 53, *59*, 64, 67, 77, 81, 89, 108, 110, 119, 121, 284, 285, 289, 290, 342, 360, 459, 478

`BorderExc` 275, 276, 277, 280, 282, 297, 300–302, 304, 305, 309, 311, 321, 334, 347, 351, 369, 372, 401, 454, 463, 467, 486, 494, 512

`Boundaries` 273, 282, 284, 287, 288, 303, 312, 428, 476, 547

boundary coordinate *270*, 271, 272, 284, 294, 296

boundary plane *270*, 273, 276, 284, 302–305, 313, 317, 321, 323, 324, 329, 334, 339, 342, 346, 372, 373, 390, 433, 454, 462, 494

`BoundaryCommFields` 275, 276, 282, 284, 297–299, 302, 334, 357, 358, 454, 472, 473

`BoundaryCommTypes` 273, 275, 276, 282, 284, 298, 299, 303, 334, 357, 358, 454, 472, 473

`BoundaryCondition` 342, 360, 432, 459, 478, 549

`BoundarySendBuf` 441, 445, 449, 450, 474, 489, 490, 518, 519, 521, 522, 525, 526, 531, 534, 538, 539

`bounds` 51, *52*, 58, *59*, 64, 67, 77, 81, 89, 108, 110, 119, 121, 273, 284, 285, 287–289, 291, 293

`build_new_comm()` 147, 155, 161, 169, 170, 201, 206, 209, 342, 350, 378, 458, 459, 467

C

C 52, *53*, 55, 59, *60*, 273, 275, 276, 280, 282, 284, 297–300, 302, 309, 311, 334, 351, 357, 372, 454, 467, 472, 494

C header files:

- `oh_config.h` 18, 25, 42, 73, 74, 106, 129, 130, 132, 153, 325, 326, 442, 554, 555
- `oh_part.h` 42, 106, 129, 130, 226, 231, 326, 554, 555
- `oh_stats.h` 98, 99, 129, 130, 132, 147, 325, 442, 554, 555
- `ohhelp1.h` *17*, 129, 130, 132, 157, 234, 282, 325, 346, 442, 462, 554, 555
- `ohhelp2.h` *17*, 129, 130, 226, 234, 282, 326, 327, 346, 462, 554, 555
- `ohhelp3.h` *17*, 128–130, 270, 282, 346, 462, 554, 555
- `ohhelp4p.h` . . . *18*, 129, 325, 346, 554, 555
- `ohhelp4s.h` *18*, 129, 442, 462
- `ohhelp_c.h` 18, 25, 26, 31, 41, 49, 75, 87, *106*, 119, 129, 130, 146, 153, 231, 278, 279, 343, 344, 460, 555

C object files:

- `ohhelp1.o` 554, 555
- `ohhelp2.o` 554, 555
- `ohhelp3.o` 554, 555
- `ohhelp4p.o` 554, 555
- `sample.o` 555
- `simulator.o` 555

C source files:

- `ohhelp1.c` *17*, 128–130, 134, 146–148, 157, 226, 554, 555
- `ohhelp2.c` . *17*, 129, 130, 134, 234, 554, 555
- `ohhelp3.c` *17*, 129, 130, 282, 554, 555
- `ohhelp4p.c` *18*, 129, 346, 554, 555
- `ohhelp4s.c` *18*, 129, 462
- `sample.c` 119, 128–130, 555
- `simulator.c` 128, 129, 555

C structs:

- `current` 119, 121, 124, 126, 127
- `ebfield` 20, 22, 61, 119, 121, 123, 127, 128
- `S_mycommc` 31, *32*, 106
- `S_particle` 22, 42, 43, 44, 46, 47, 57, 63, 73, 76, 80, 82–84, 91, 98, 106, 119, 123, 124

`cbufsize` *89*, 92, 469, 473, 474

`cd` 53, 60, 69–72, 108, 110, 112, 119, 121–123

`cfields` *53*, 56, *60*, 61, 62, 64, 77, 89, 108, 110, 119, 121, 275, 284, 285, 297, 334, 357, 358, 454, 472, 473

`Check_Particle_Location()` 328, 333, 350, 358, 427, 430, 435, 437, 444, 453, 466, 474, 546, 548, 550, 552

`clear_border_exchange()` 276, 280, 301, 309, 311, 351, 401, 467, 512

`clear_stats()` 150, 157, 214, 215, 218

`Combine_Subdom_Pos()` 327, 431, 435, 444, 492, 548, 550

`CommList` 142, 144, 164, 190–198, 200, 201, 206, 243, 254, 255, 263, 264, 330, 335, 336, 338, 339, 369, 370, 376–379, 383, 384, 387, 388, 393–395, 400, 417, 454, 455–457, 487, 488, 497, 498, 500, 506, 507, 518

`Comms` 140, 145, 164, 207, 208

`comp_xyz()` 282, 295, 296

`compare_int()` 157, 187, 188

contacting subcuboid *441*, 449, 518

Coord_To_Index() 352, 353, 355, 375, 387, 402, 431, 435, 467, 470, 504, 513, 548, 550
count_next_particles() 137, 143, 157, 198, 201, 248
count_population() 327, 330–333, 348, 353, 364, 365, 370, 410, 411, 416, 443–445, 451–453, 462, 468, 486, 492
count_real_stay() 157, 191, 193, 205
count_stay() 134, 137–139, 157, 177, 184, 382
ctypes 52, *55*, 56, 59, *60*, 61, 62, 64, 71, 77, 89, 108, 110, 112, 119, 121, 123, 284, 285, 297, 334, 357, 358, 454, 472, 473
current 119, 121, 124, 126, 127
current scattering *19*, 21, 22, 100, 108, 114, 124
current_scatter() 108, 112, 114, 120, 122, 124
currMode 134, 135, 162, 170–174, 239, 321, 364, 435, 437, 482, 550, 552

D

D *11*, 25, 28, 30, 32, 33, 42, 50–53, 55–62, 64–67, 71, 74, 77, 79, 81, 91, 132, 133, 142–144, 146, 147, 153, 160, 161, 164–167, 169, 191, 192, 195, 229, 230, 238, 240, 262, 270, 272, 273, 275, 276, 280, 282, 284, 287–291, 293–298, 300–307, 309–313, 315, 317, 318, 321, 323, 325–328, 331, 334, 336–342, 351–353, 359–361, 367, 372, 373, 377, 378, 381, 384–387, 390–399, 401–403, 408, 411, 412, 415, 416, 418, 424, 428–433, 435, 436, 442, 443, 448, 449, 454, 455, 457–459, 467, 471, 476, 478, 484, 494, 495, 498, 501, 502, 505, 506, 508, 512–514, 518, 520, 521, 523, 538, 541
DBL_MAX 132, 215
$\mathcal{D}$ . . . *84*, 88, 89, 440, 450, 452, 469, 473, 474
Decl_For_All_Grid() 353, 362, 372, 373, 381, 387, 410, 412, 413, 423, 468, 480, 492, 493, 495, 506, 507, 511, 520, 523, 534, 537, 543, 545
Decl_Grid_Info() 229, 230, 258, 260, 262, 327, 367, 410, 412, 413, 416, 418, 420, 422, 423, 430, 434, 436, 438, 443, 484, 492, 527, 530, 531, 533–535, 548, 550, 552
dint . . 133, 329, 355, 359, 412, 444, 447, 470
Do_Y() 351, 402, 430, 435, 436, 467, 548, 550
Do_Z() 351, 402, 430, 435, 436, 467, 548, 550
dprint() 155, 224, 225
DstNeighbors 146, 147, 166, 190–192, 195, 209, 240, 288, 342, 377, 391, 458, 459, 476, 477

E

E (electric field) 18, 19, 108, 112, 118, 123, 128
e^g 325, 326, 331, 334, 353, 355, 357, 361, 363, 372, 373, 381, 387, 390, 402, 403, 428, 430, 434, 435, 442, 443, 446, 450, 453, 454, 470–472, 474, 479, 481, 488, 493, 494, 500, 504, 506, 508, 509, 511, 514, 521, 523, 539, 540, 543
$e_l(f)$. *53*, 56, *59*, 62, 274, 284, 299, 333, 453
$e_l^b(f)$ *53*, 54, 56, *60*, 62, 69, 274, 284, 299, 300, 334, 453
$e_u^{\max}$ 274, 299, 300
$e_l^{\min}$ 274, 299, 300
$e_l^r(f)$ *53*, 56, *60*, 62, 70, 274, 284, 299, 300, 334, 453
$e_u(f)$. *53*, 56, *59*, 62, 274, 284, 299, 333, 453
$e_u^b(f)$ *53*, 54, 56, *60*, 62, 69, 274, 284, 299, 301, 334, 453
$e_u^r(f)$ *53*, 56, *60*, 62, 70, 274, 284, 299, 301, 334, 453
eb 19–22, 52, 53, 56, 57, 59–62, 68, 69, 71, 108, 110–112, 119, 121–123
ebfield . . 20, 22, 61, 119, 121, 123, 127, 128
errstop() 154, 165, 167, 168, 183, 268, 290, 293, 299, 350, 358, 359, 466, 472, 474, 478
exchange_border_data_h() 448, 456, 464, 542, 545
exchange_border_data_v() 447, 450, 451, 456, 464, 504, 519, 542
exchange_particles() 139, 143, 148, 215, 227, 228, 233, 243, 244, 248, 253, 263, 264, 351, 369, 370, 410, 456, 467, 486
exchange_particles4p() 327, 331, 332, 334, 341, 346, 351, 354, 366–368, 369, 371, 374, 375, 383, 390, 400–402, 404, 409, 410, 412, 415, 418, 422, 425, 483, 485–487
exchange_particles4s() 440, 441, 443, 446, 447, 450–452, 454, 456, 457, 462, 467–469, 480, 482–484, 485, 489, 492, 496, 500, 512–516, 518, 526, 532, 533, 535–537, 540
exchange_population() 323, 329, 331, 334, 346, 353, 365, 370, 371, 373, 374, 411, 440, 445, 447, 451, 454, 462, 468, 476, 487, 492, 494, 495
exchange_primary_particles() . . . 232, 240, 350, 364, 365, 414, 415, 467, 486
exchange_xfer_amount() 323, 337, 338, 340, 342, 348, 371, 409, 441, 456–458, 463, 485, 488, 516
excludeLevel2 . . . . . 270, 287, 311
exit() 168

exterior *339*, 377, 383, 384, 386–392, 428, 429, 431, 433, 435, 440, 443, 470, 474, 480, 490, 493, 502, 504–508, 511, 514, 521
exterior halo plane . . *440*, 443, 453, 476, 488
exterior pillar *445*, 450, 474, 489, 490, 518, 519, 521, 525, 539, 545
`EXTERN` 134, 157, 226, 234, 282, 346, 462
`extern` 134, 234, 282, 346, 462

F

F *52*, 53, 55, *59*, 60, 61, 77, 79, 89, 91, 274, 280, 284, 297–301, 311, 333–335, 347, 351, 357, 359, 361, 374, 378, 401, 403, 453–455, 463, 467, 472, 474, 479, 495, 499, 512, 514
$F(n)$ *14*, 15, 16, 35, 143, 170, 175, 177, 181, 182, 184, 186–188, 190, 208, 337, 363, 379, 441, 449, 450
$F_l(n)$ *16*, 182, 183, 187, 188
`FALSE` 132, 175, 179, 240, 243, 364, 366, 482, 483
`FamIndex` 146, 169, 170, 175, 208
`FamMembers` 146, 169, 170, 175, 208
`FCD` 108, 110, 112, 119, 121, 123
$fdisp(f, x, y, z)$ 297, 300, 301, 306–308
`FEB` 108, 110, 111, 119, 121–123
`fflush()` 224
field solving *19*, 22, 100, 108, 117, 118, 127, 128
field-array *23*, 37, 48, 49, 52–56, 59–65, 68–71, 77, 108, 110–115, 117–125, 127, 128, 130, 135, 270, 274–277, 282, 284, 285, 289, 299, 300, 302–308, 319, 321, 333, 334
`field_array_size()` 120, 121
`Field_Disp()` 275, 297, 301, 306–308
`field_solve_b()` 108, 112, 118, 120, 123, 128
`field_solve_e()` 108, 112, 117, 120, 123, 127
`FieldDesc` 274, 277, 280, 282, 297–301, 303, 311, 319, 320, 333, 334, 347, 351, 359, 369, 374, 378, 401, 403, 453, 454, 463, 467, 474, 486, 495, 512, 514
`FieldTypes` 274, 282, 284, 297–299, 301, 311, 333, 334, 357, 358, 401, 453, 472, 473, 512
`FirstNeighbor` 341, 342, 360, 377, 458, 459, 477
float.h (standard header file) 132
floating particle 138, 157, 179, 181, 182, 184, 189
`For_All_Grid()` 331, 351, 352, 353, 354, 363, 372, 381, 388, 403, 410, 412, 414, 423, 451, 467, 468, 481, 488–490, 493–495, 503, 504, 511
`For_All_Grid_Abs()` 331, 351, 352, 353, 354, 373, 451, 467, 468
`For_All_Grid_From()` 351, 353, 354, 379, 381, 383
`For_All_Grid_XY()` 467, 468, 503, 505, 510, 511, 522, 524, 543, 545
`For_All_Grid_XY_At_Z()` . 451, 467, 468, 511
`For_All_Grid_Z()` 451, 467, 468, 503, 507, 508, 522, 524, 543, 545
`For_Y()` 351, 353, 381, 467, 504
`For_Z()` 351, 353, 381, 467, 504
Fortran header files:
ohhelp_f.h 18, 25, *106*, 108, 129, 130, 153, 231, 278, 279, 343, 460, 554
Fortran module files:
oh_mod1.F90 . . 26, 38, 129, 130, 154, 554
oh_mod2.F90 . . . . . 41, 129, 130, 232, 554
oh_mod3.F90 . 49, 108, 129, 130, 278, 554
oh_mod4p.F90 . . . . . 75, 93, 129, 130, 344
oh_mod4s.F90 . . . . . 87, 129, 130, 460
oh_type.F90 28, 42, 129, 130, 146, 326, 554
Fortran modules:
`oh_type` 28
`ohhelp1` 26
`ohhelp2` 41
`ohhelp3` 49, 108
`ohhelp4p` 75
`ohhelp4s` 87
`sample` 108
Fortran object files:
oh_mod1.o 554
oh_mod2.o 554
oh_mod3.o 554
oh_mod4p.o 554
oh_type.o 554
sample.o 554
simulator.o 554
Fortran source files:
sample.F90 . . . . . 108, 128–130, 554
simulator.F90 . . . . . 128, 129, 554
Fortran types:
`oh_mycomm` 28, 43, 49, 63, 75, 87, 108
`oh_particle` 42, 43, 44, 46, 47, 49, 73, 75, 76, 80, 82–84, 91, 98, 108, 113, 114
`fprintf()` 168
`fsizes` *55*, 56, 57, *61*, 62, 64, 69–71, *77*, 79, 89, 91, 108, 110, 119–123, 284, 285, 297, 298, 300, 334, 361, 479
`ftypes` *52*, 54, 56, *59*, 61, 62, 64, 69, 70, 77, 89, 108, 110, 119, 121, 274, 284, 285, 297, 300, 333, 357, 358, 453, 472, 473
function aliases:
`oh_accom_mode()` *107*, 153
`oh_all_reduce()` *107*, 153
`oh_allreduce_field()` *107*, 112, 123, 278
`oh_bcast_field()` *107*, 111, 122, 278

oh_broadcast() *107*, 153
oh_exchange_border_data() . . . *107*, 460
oh_exchange_borders() *107*, 111, 112, 122, 123, 278
oh_families() *107*, 153
oh_grid_size() *107*, 278
oh_init() *107*, 110, 121, 153, 232, 279, 343, 460
oh_init_stats() *107*, 153
oh_inject_particle() *107*, 231, 343, 460
oh_map_particle_to_neighbor() *107*, 114, 124, 278, 279, 343, 460
oh_map_particle_to_subdomain() *107*, 278, 279, 343, 460
oh_max_local_particles() *107*, 110, 121, 231, 343
oh_neighbors() *107*, 153
oh_particle_buffer() *107*, 460
oh_per_grid_histogram() . *107*, 343, 460
oh_print_stats() *107*, 153
oh_reduce() *107*, 153
oh_reduce_field() *107*, 278
oh_remap_injected_particle() *107*, 231
oh_remap_particle_to_neighbor() *107*, 343, 460
oh_remap_particle_to_subdomain() *107*, 343, 460
oh_remove_injected_particle() *107*, 231
oh_remove_mapped_particle() *107*, 343, 460
oh_set_total_particles() *107*, 231
oh_show_stats() *107*, 153
oh_stats_time() *107*, 153
oh_transbound() *107*, 111, 122, 153, 232, 279, 343, 460
oh_verbose() *107*, 153

G

G 326, 329, 330, 334, 358, 359, 361, 443, 444, 446, 454, 474, 479
gather_hspot_recv() 324, 331, 337, 339, 340, 342, 347, 351, 352, 386, 390, 392, 394–396, 398, 485
gather_hspot_send() 324, 337, 342, 347, 368, 386, 392, 393, 398, 485
gather_hspot_send_body() 329, 339, 340, 347, 388, 393, 398, 485
$gidx(x, y, z)$. 326, 330, 331, 341, 352–355, 359, 363, 372, 373, 376, 387, 391, 402, 410, 412, 428, 430–433, 435, 440, 443, 445–447, 467, 468, 470, 474, 481, 493, 494, 513, 521, 522, 524, 538–540, 543
GreaterHeap 142, 164, 203–205, 209, 210
Grid 270, 271, 272, 282, 288–290, 293, 294, 310, 312, 314, 317, 402, 428, 432, 433, 435, 474, 514, 547, 549, 550
grid size 48, *64*, *270*, 271, 277, 288, 310
grid-position (of particle) 229, *323*, 325, 327, 335, 341, 359, 364, 376, 391, 394, 411, 412, 414–416, 424, 429–431, 434, 442, 443, 458, 482, 492, 524, 525
grid-size-aware *270*, 271, 272, 288
grid-voxel *17*, *72*, 73–75, 78, 79, 82, 84–86, 88, 89, 91, 92, *323*, 324, 326, 329, 330, 335–338, 346–348, 353, 361, 363, 371, 375, 376, 379–381, 383, 387–390, 400, 410, 412, 414–416, 422, 424, 428, 431, 432, 435, 442–446, 450, 462–464, 468, 485, 489, 491, 492, 499–502, 506, 507, 518, 519, 521, 523, 525, 528, 530, 538–540, 543, 544
Grid_Boundary() 386, 387
Grid_Exterior_Boundary() 504, 506, 523, 539, 545
Grid_Interior_Boundary() . . . 504, 521, 543
Grid_Position() 327, 411, 412, 414–416, 443, 492, 525, 552
Grid_X() 354, 381, 468
Grid_Y() 354, 381, 468
Grid_Z() 354, 381, 468, 494, 522, 524
GridDesc 331, 347, 353, 355, 358, 361, 367, 369, 371, 372, 375, 378, 381, 387, 391, 401, 402, 412, 423, 430, 435, 450, 463, 470, 474, 475, 479, 484, 486, 489–491, 493, 496, 499, 504, 506, 507, 512–514, 520, 522–524, 539, 543, 545, 548, 550
gridMask 229, 326, 327, 359, 443, 474
GridOffset 341, 347, 360, 367, 369, 401, 402, 416, 458, 463, 477, 484, 486, 491, 512, 513, 524
gridOverflowLimit . 324, 338, 356, 361, 381
GroupWorld 145, 164, 207

H

$H(n)$ *14*, 15, 16, 35, 137, 140, 143, 177, 178, 181–187, 191, 193, 205, 208, 256, 391, 408
halo particle *17*, *84*, 85, 86, 89, 92–94, *440*, 441–443, 445–448, 450–452, 455, 456, 459, 463, 464, 469, 482, 485, 487–490, 493, 496, 498, 501, 502, 507–509, 511, 518, 520, 521, 523–527, 529, 531, 532, 538, 540–542, 546
halo plane *440*, 447, 487, 492–495
$hbuf_h^r(p, \beta, s)$ . . . 441, 448, 509–511, 540, 541
$hbuf_h^s(p, \beta, s)$ . . . 441, 448, 509–511, 540, 541

$hbuf_v^r(d,p,\beta,m)$ 441, 449, 518, 519, 523, 524, 538, 539, 544
$hbuf_v^s(d,p,\beta,m)$ 441, 449, 450, 518, 519, 521, 522, 538, 539, 543, 544
helpand *12*, 13–15, 17, 19, 20, 22, 23, 26–28, 30–41, 45, 48, 65, 66, 71, 79, 80, 92, 99, 104, 106, 111, 122, 134, 140, 141, 143, 145, 147, 152, 155, 159, 161, 162, 169, 171, 174, 176, 177, 180, 181, 185, 188–190, 192–195, 197, 198, 201, 202, 204, 207, 209, 212, 213, 234, 243, 244, 249, 251, 253–255, 257, 323, 324, 331, 334–339, 342, 346, 347, 350, 354, 363, 366, 367, 369–371, 375–378, 382–385, 388, 389, 391, 393, 394, 398, 399, 402–404, 406, 408, 409, 412, 414, 418, 422, 425, 441, 446, 448, 453–455, 457, 458, 462–464, 466, 467, 469, 481–483, 486, 487, 489, 491, 496–498, 500–502, 514, 515, 517, 518, 526–528, 532–537
helper *11*, 12–15, 17, 19, 20, 22, 23, 26–28, 30–41, 45, 48, 53, 60, 65, 68, 70–72, 79, 92, 99, 104, 106, 111, 122, 134, 135, 137, 140, 141, 143, 145, 152, 155, 157, 169, 171, 174, 176–178, 180–183, 185, 188–190, 192–195, 201, 202, 204, 205, 207, 209, 212, 213, 234, 243, 244, 249, 251, 253–256, 323, 324, 329, 331, 334–337, 339, 342, 346, 347, 350, 354, 363, 365–367, 369–372, 375–378, 382, 383, 385, 388, 390–394, 397, 398, 402–409, 412, 418, 422, 425, 441, 446, 448, 449, 453–455, 457, 462–464, 466, 467, 469, 481, 483, 486, 487, 489, 491, 493, 494, 496–498, 500, 502, 514, 515, 517, 518, 526–528, 532–537
horizontal exterior halo plane *441*, 491, 500, 509, 511, 540, 541, 545
horizontal halo plane *441*, 448, 456, 463, 464, 471, 475, 490, 500, 502, 508, 541, 542
horizontal interior halo plane *441*, 500, 502, 506, 540, 541, 545
hot-spot *74*, 77–79, *324*, 330, 335–340, 343, 347, 361, 375, 376, 379, 380, 382, 383, 385–400, 421, 440–442, 455, 456, 458, 475, 485, 496, 497, 499, 501, 525–531
hot-spot sending block (of `CommList`) 335, *336*, 338, 339, 370, 375, 376, 378, 383, 386, 388, 392, 395, 397, 417, 455, 496
`HotSpot` 339, 384–386, 388, 390, 392, 393, 395, 398, 399
`HotSpotList` 338, 339, 359, 384
`HotSpotTop` 338, 339, 384, 388
`HPlane` 448, 475, 488, 491, 502, 507–511, 540, 541, 546
`HSReceiver` 340, 360, 396
`HSRecv` 339, 359, 360, 391, 392, 396
`HSRecvFromParent` 340, 360, 394, 399
`HSSend` 340, 360, 394

I

`If_Dim()` 351, 359, 387, 391, 403, 434, 435, 467, 474, 514, 550
`Index_To_Coord()` 352, 391
`init1()` 134–139, 141–145, 147, 151, 152, 154, 155, 159–161, 162, 167–170, 237, 287, 288, 336, 455, 458, 476, 477
`init2()` 162, 163, 167, 227, 228, 232, 235, 236, 287, 337, 478, 479
`init3()` 147, 162, 167, 236, 270, 273, 280, 284, 285, 287, 289, 293, 297, 334, 351, 356, 358, 454, 467, 473, 479
`init4p()` 163, 238, 298, 326, 327, 329, 331–334, 337, 339–342, 346, 350, 351, 354, 355, 356, 358, 361, 363, 401–404, 471–474, 476–478
`init4s()` 442–444, 447–454, 456–459, 462, 466, 467, 469, 470, 471, 473, 479, 491, 513–515
`init_fields()` 167, 168, 270, 274–276, 282, 289, 297, 301, 302, 309, 334, 358, 454
`init_subdomain_actively()` 168, 270–273, 282, 284, 288, 289
`init_subdomain_passively()` . 167, 168, 270, 272, 273, 282, 284, 288, 293, 296
`initialize_eb()` . . . 108, 109, 111, 120, 122
`initialize_particles()` 108, 109, 111, 120, 122
`InjectedParticles` 137, 138, 163, 231, 239, 244, 247, 249, 252, 253, 262, 265–267, 333, 363, 367, 414, 431, 432, 435, 437, 453, 481, 483, 548, 550, 552
inner extension *302*, 325
inner extent *305*, 306, 307
`INT_MAX` 132, 215, 218, 268, 359, 361, 385, 386, 393, 399, 474
interior *339*, 376, 381, 383, 384, 386–390, 402, 434, 440, 443, 446, 487, 490, 493, 502, 504–506
interior halo plane *440*, 476, 531–533
interior pillar *445*, 450, 519, 521, 525
`InteriorParts` . . 451, 475, 526–529, 531, 532
irregular process coordinate *271*, 272, 280, 282, 284, 288, 310, 315, 351, 435, 467
`Is_Boundary()` 390, 391
`Is_Pillar_Voxel()` . 519, 520, 525, 532, 539

J

J (current density) 19

K

K_p 148, 150, 215
K_t 147, 148–150, 215

L

`LessHeap` 142, 164, 203, 204, 209, 210
limits.h (standard header file) 132
`Local_Coordinate()` 433, 435, 549, 550
`local_errstop()` 154, 166, 168, 172, 194, 265–267, 294, 350, 427, 437, 466, 500, 546, 551
`Local_Grid_Position()` 341, 416, 437, 458, 524, 525, 552
`logGrid` 229, 326, 327, 359, 443, 474
Lorentz force law 10, 18
`lorentz()` . . . 21, 22, 106, 109, 113, 120, 124

M

make files:
  samplec.mk *128*, 130, 554, 555
  samplef.mk *128*, 130, 554
`make_brecv_sched()` 447, 450, 451, 456, 464, 468, 504, 518, 523, 539
`make_bsend_sched()` 445–447, 450, 451, 456, 464, 468, 504, 518, 520, 523
`make_bxfer_sched()` 441, 450, 456, 457, 459, 463, 485, 489, 490, 518, 520, 522–524, 539
`make_comm_count()` 137–139, 143, 144, 157, 184, 197, 200, 201, 209, 216, 244, 256, 370, 378, 486
`make_recv_count()` 137, 138, 143, 157, 198, 200, 248
`make_recv_list()` 191, 205, 209, 323, 329, 331, 333, 334, 338, 341, 342, 347, 350, 354, 367, 370, 375, 381, 396, 401, 402, 404, 440, 441, 445, 451, 454, 456–459, 463, 467, 469, 485, 488, 496, 499, 512, 514, 515
`make_send_count()` . 138, 143, 157, 198, 200
`make_send_sched()` 323, 324, 333, 338, 339, 342, 347, 368–370, 377, 383, 387, 390, 392, 395, 398, 412, 420, 441, 442, 447, 448, 453, 456, 457, 459, 463, 477, 485, 488, 500, 503, 505, 507
`Make_Send_Sched_Body()` 445, 456, 504, 505, 506
`make_send_sched_body()` . 329–331, 333, 337, 339, 347, 352, 353, 384–386, 387, 420, 445, 451, 456, 463, 501, 504, 505
`make_send_sched_hplane()` 445, 447, 448, 451, 453, 463, 508, 509, 511
`make_send_sched_self()` 447, 448, 451, 456, 463, 468, 502, 504, 507, 511
`malloc()` 20, 24, 61, 119–121, 132, 154, 167, 288, 350, 466
`malloc_field_array()` 120, 121
`map_irregular()` 272, 282, 317, 318
`map_irregular_range()` . . 272, 282, 317, 318
`map_irregular_subdomain()` 280, 282, 315, 317, 351, 435, 467, 550
`Map_Particle_To_Neighbor()` . 270, 272, 273, 312, 313, 428, 430, 431, 547, 548
`Map_Particle_To_Subdomain()` 272, 314, 315, 342, 433, 435, 549, 550
`Map_To_Grid()` 432, 435, 459, 549, 550
`maxdensity` 84, *88*, 469, 473
`MAXFRAC` 108, 110, 119, 121
`maxfrac` *27*, *30*, 44, 45, 50, 57, 64, 76, 78, 88, 108, 119, 136, 159, 162, 235, 268, 283, 285, 473
`maxFraction` 136, 159, 162, 174
maximum density *84*, 88, *440*
`maxlocalp` *44*, 45, 50, 57, 72, 76–78, *88*, 89, *90*, 110, 121, 226, 235, 237, 283, 285, 332, 356, 358, 361, 452, 469, 471, 473, 474, 478, 479
Maxwell's equation 10, 19, 118, 127, 128
`MCW` 133, 145, 155, 162, 164, 166, 168, 170, 173, 185, 192, 198, 207, 218, 220, 222, 241, 243, 257, 263, 264, 268, 321, 356, 377, 392, 394, 395, 397, 409, 410, 426, 472, 498, 517, 536, 538, 546
`mem_alloc()` . 154, 162–165, 167, 169, 170, 237, 238, 287, 293, 298, 350, 355–357, 360, 466, 470, 472, 474, 475, 477, 481
`mem_alloc_error()` 154, 167, 168, 268, 350, 361, 466, 474
`memcpy()` 288, 298, 357, 360, 374, 478, 495, 543, 545
`Message` 272, 290, 293, 294
`minmargin` *88*, 469, 473
`Mode_Acc()` 135, 175, 365–367, 369–371, 384, 386, 391, 437, 483, 484, 486, 487, 496, 552
`MODE_ANY_PRI` 135, 216
`MODE_ANY_SEC` 135, 197
`Mode_Is_Any()` 135, 174, 197, 198, 206
`Mode_Is_Norm()` 135, 173, 209, 244
`MODE_NORM_PRI` 134, 135, 162, 174, 197, 206, 239, 241, 362, 480
`MODE_NORM_SEC` 134, 135, 172, 174, 184, 197, 205, 239, 243, 244, 362, 364, 480, 482
`Mode_PS()` 135, 170, 172, 174, 175, 203, 206, 216, 220, 239,

240, 254, 321, 362, 364, 365, 369, 372, 378, 391, 480, 482, 486, 487, 493, 498
MODE_REB_SEC 134, 135, 174, 239, 362, 364, 480, 482
Mode_Set_Any() 135, 173, 435, 487, 550
Mode_Set_Norm() 135
Mode_Set_Pri() 135
Mode_Set_Sec() 135
Monte Carlo collision 17
move_and_sort() 441, 443, 444, 446, 450–453, 456, 458, 464, 468, 485, 488, 490, 520, 521, 525, 533, 535, 538
move_and_sort_primary() 323, 327–333, 348, 351, 353, 365, 413, 453, 485, 486
move_and_sort_secondary() 323, 327, 328, 330–333, 337, 348, 353, 371, 389, 400, 411, 416, 417, 422, 424, 453, 486, 490, 533
move_injected_from_sendbuf() 139, 227, 228, 234, 247, 249, 252, 262
move_injected_to_sendbuf() . 138, 139, 227–230, 234, 246, 247, 250, 261, 262
Move_Or_Do() . . . 327, 328, 341, 416, 418, 419, 421, 422, 424, 443, 444, 450, 456, 458, 520, 524, 525, 526, 527, 529–534
move_to_sendbuf_4s() 441, 443, 444, 446, 451–453, 456, 464, 485, 489, 490, 525, 526, 529, 530, 535
move_to_sendbuf_dw() 227–229, 234, 247, 251, 252, 259, 260, 348, 464
move_to_sendbuf_dw4p() 327, 328, 330, 332, 333, 337, 348, 400, 416, 418, 419, 421, 422, 530
move_to_sendbuf_dw4s() . 443, 444, 446, 451–453, 456, 464, 525, 527, 528, 530
move_to_sendbuf_primary() 136–138, 148, 215, 227, 228, 233, 240, 242, 245, 248, 250, 252, 257, 260–262, 351, 364, 365, 410, 412, 464, 467, 486
move_to_sendbuf_sec4p() . . . 327, 328, 330, 332, 333, 348, 371, 389, 400, 411, 412, 416, 418, 420, 422, 424, 489, 526
move_to_sendbuf_secondary() 137–139, 148, 199, 215, 227, 228, 234, 248, 252, 254, 257, 260–262, 348, 367, 371, 410, 418, 464
move_to_sendbuf_uw() . . . 227–229, 234, 246, 247, 250–252, 257, 260, 348, 464
move_to_sendbuf_uw4p() 327, 328, 330, 332, 333, 337, 348, 400, 416–419, 420, 529
move_to_sendbuf_uw4s() . 443, 444, 446, 451–453, 456, 464, 525, 527, 528, 529
mpi.h (standard header file) . . . 106, 132, 146
MPI_Abort() 168
MPI_Allgather() 137, 185
MPI_Allreduce() 23, 136, 173, 212, 346, 374
mpi_allreduce_wrapper() 346, 370, 374, 485
MPI_Alltoall() 136, 139, 163, 173, 198
MPI_Alltoallv() 228, 242, 256
MPI_Barrier() 106, 155, 218
MPI_Bcast() 23, 202
MPI_BYTE 164, 214, 237
MPI_Comm 146
MPI_Comm_c2f() 164, 208
MPI_Comm_create() 207
MPI_Comm_f2c() 133
MPI_Comm_group() 164
MPI_COMM_NULL 28, 31, 32, 145, 164, 202, 374, 493, 495
MPI_Comm_rank() 134, 162
MPI_Comm_size() 134, 162, 170, 268, 356, 472
MPI_COMM_WORLD 133, 134, 145, 162, 164
MPI_Datatype 139, 144, 151, 228, 276, 340, 457, 542, 546
MPI_DATATYPE_NULL 304–306, 308, 309
MPI_DOUBLE 68, 69, 300, 302, 303, 319–321, 334
MPI_DOUBLE_PRECISION 68, 69
MPI_Finalize() 168
MPI_Get_count() 192, 377
MPI_Group 145
MPI_Group_incl() 207
MPI_Group_translate_ranks() 207
MPI_IN_PLACE 212, 213, 374, 495
MPI_INT . . . 360, 378, 392, 394, 395, 476, 498
MPI_Irecv() 228, 256, 263, 337, 386, 390–395, 398, 409, 410, 426, 427, 448, 517, 536, 538, 539, 541, 544, 546
MPI_Isend() 228, 256, 263, 264, 410, 426, 427, 448, 517, 536, 538, 539, 541, 544, 546
MPI_LONG_LONG_INT 300, 302, 303, 321, 334, 374, 378, 495, 498
MPI_Op 151
MPI_Op_create() 214
MPI_Op_f2c() 212, 213
MPI_ORDER_C 309
MPI_PROC_NULL . . 448, 475, 502, 508, 541, 546
MPI_Recv() 166, 191, 241, 321, 377
MPI_Reduce() 23, 151, 213, 214, 220–223, 346, 372, 374, 495
MPI_Request 228, 263, 264
MPI_Send() 166, 191, 241, 321, 377, 394, 395, 397
MPI_Sendrecv() . 166, 191, 240, 241, 321, 377
MPI_SOURCE 396
MPI_Status 228
MPI_SUM 69, 320, 374, 495

MPI_TAG 396
MPI_Type_commit() 163, 164, 214, 237, 306–308, 360, 476
MPI_Type_contiguous() 144, 164, 214, 228, 237, 309
MPI_Type_create_subarray() 309
MPI_Type_f2c() 202, 212, 213, 542
MPI_Type_free() 309
MPI_Type_get_extent() 542
MPI_Type_struct() 139, 163, 307–309
MPI_Type_vector() 139, 163, 306–309, 340, 360, 457, 476
MPI_UB 163, 304
MPI_Waitall() 228, 256, 395, 398, 410, 427, 517, 536, 539, 541, 544, 546
MPI_Wtime() 214, 215
MyComm 145, 146, 160, 164, 202, 208, 374, 493, 495
mycomm *28*, *31*, 36, 44, 45, 50, 57, 64, 77, 89, 108, 110, 121, 146, 159, 161, 162, 235, 236, 283, 285, 287, 356, 471
MyCommC 145, 146, 160, 161, 164, 208, 236, 285
MyCommF . . . 146, 160, 161, 164, 208, 236, 285
myRank 134, 162, 175, 185, 189, 193, 197, 200, 201, 203, 218, 224, 240, 243–245, 248, 253, 254, 263–267, 359, 364, 367, 369, 375, 387, 391, 395, 399, 405, 408, 413, 418, 423, 474, 484, 486, 496, 506, 507, 515, 516, 527, 534

N

N *11*, 12, 13, 21, 24, 27, 28, 30–35, 45, 50–52, 56–59, 62, 68, 74, 78, 88, 98, 110, 121, 134, 136–143, 145–147, 159, 162–167, 169, 170, 172–177, 180, 181, 184, 192, 195, 200, 201, 203–205, 208, 217, 220, 227, 228, 230, 237, 238, 242, 245, 246, 248, 250, 252, 263, 268, 270–273, 284, 287–291, 293–296, 310, 312, 313, 317, 326–328, 330, 332, 335–342, 356, 359, 360, 365, 376, 377, 381, 384, 389, 392, 393, 397, 405, 408–410, 413, 415, 417, 418, 424, 431, 436, 437, 445, 448–450, 452, 454–459, 472, 475–477, 497, 499, 501, 502, 505–508, 510, 525
nbor *28*, 29, 30, *32*, 33, 44, 50, 57, 64, 66, 67, 77, 80, 81, 89, 108, 110, 119, 121, 147, 160, 161, 165, 166, 236, 241, 283, 285
nbound *52*, *59*, 64, 77, 89, 273, 284, 285, 289, 297
neighbor 12, 26, *28*, 30, *32*, 35–37, 48–50, 55, 58, 60, 65–67, 79–81, 86, 104, 108, 115, 116, 125, 126, *133*, 136, 141, 143, 144, 146, 147, 160, 166, 168, 173, 184, 189–192, 195, 196, 206, 209, 229, 230, 257, 323–329, 335–337, 339, 341, 342, 347, 375, 377, 378, 384–386, 391–394, 398, 399, 404, 406, 408, 411, 412, 415, 416, 418, 428–433, 435, 441, 442, 445, 446, 448–450, 455–460, 463, 464, 476, 477, 482, 484, 487, 488, 490–494, 496, 498, 501, 502, 504–506, 516, 518–521, 523–526, 538, 539, 543–545
Neighbor_Grid_Offset() 401, 402, 451, 512, 513
Neighbor_Id() 312, 313
Neighbor_Subdomain_Id() 327, 328, 414, 416, 418, 443, 444, 525
Neighbors 146, 147, 166, 172, 201, 209, 230, 312, 313, 324–327, 341, 342, 360, 367, 378, 384, 385, 393, 398, 402, 404–408, 457, 458, 459, 476, 477, 484, 501, 502, 513, 516
NeighborsShadow 147, 161, 165, 166, 169, 201, 209
NeighborsTemp 147, 161, 166, 169
NodeQueue 141, 145, 164, 177, 180, 181, 203–205, 207
Nodes . . 141, 164, 177–186, 189, 190, 194, 197, 203, 205–209, 243, 249, 251, 255, 367, 375, 391, 405–408, 484, 497, 515
NodesNext 141, 164, 189, 190, 194, 203–206, 244, 367, 375, 391, 407, 408, 484, 497, 515
nOfBoundaries 273, 284, 298, 302
nOfExc 273, 275, 298, 309, 334, 372, 454, 493
nOfFields 274, 298, 301, 333, 374, 375, 403, 453, 495, 496, 514
nOfInjections 228, 238, 239, 244, 246, 247, 250, 252, 261, 265–267, 333, 363, 367, 411, 415, 418, 424, 427, 430, 435, 437, 453, 481, 484, 527, 546, 551
nOfLocalPLimit 226, 237, 265, 332, 333, 365, 371, 437, 452, 453, 479, 489, 527, 551
nOfLocalPLimitShadow 332, 356, 358, 361, 452, 471, 473, 474, 478
nOfLocalPMax 136, 137, 174–176, 188, 238, 362, 480
nOfNodes 130, 134, 162, 170, 172, 175, 176, 185, 189, 193, 197, 200, 201, 203, 208, 216, 218, 220, 237, 239, 240, 245, 248, 253, 254, 258, 260, 262–267, 287, 289, 293, 312, 317, 362, 364, 375, 384, 387, 391, 402,

405, 409, 413, 418, 423, 425, 426, 430, 436, 438, 480, 487, 496, 501, 506, 507, 513, 515, 517, 534–536, 548, 550, 552
nOfParticles . . . 137, 174, 203, 238, 362, 480
NOfPGrid . . 323, 329, 330, 340, 355, 358, 359, 363–365, 370, 372–374, 389, 391, 394, 400, 410–414, 417, 418, 421, 422, 424, 429, 431, 433–437, 444, 445, 446, 470, 474, 480, 481, 486–490, 492–495, 505–507, 518, 519, 521, 525, 528, 530–532, 534, 535, 539, 548, 550, 552
NOfPGridIndex 446, 447, 470, 479, 485, 488, 539, 543, 544
NOfPGridIndexShadow 446, 447, 470, 479, 485, 488
NOfPGridOut 330, 331, 334, 355, 361, 364, 365, 371, 388, 389, 395, 399, 412, 414, 423, 446, 447, 470, 479, 485, 488, 491, 502, 507, 509–511, 520, 521, 523, 539, 543, 544
NOfPGridOutShadow . 446, 470, 479, 485, 488
NOfPGridTotal . . 323, 329, 330, 331, 355, 358, 359, 363, 365, 370–374, 378, 379, 383, 389, 395, 411–416, 423, 424, 440, 443, 444, 445–447, 470, 474, 486–488, 491–495, 507, 511, 523, 531–535
NOfPGridZ 445, 447, 474, 487, 493, 494
NOfPLocal . 135, 136, 137, 154, 159, 162, 172–176, 191, 193, 199, 205, 216, 217, 231, 239, 241, 242, 245, 246, 248–253, 258, 265–268, 332, 333, 358, 362, 363, 365, 413, 414, 429, 431, 434, 436, 437, 452, 453, 473, 481, 522, 548, 550, 552
NOfPrimaries 130, 136, 163, 173, 175, 176, 191, 193, 194, 216, 217, 238, 241, 243, 245, 333, 362, 382, 413, 480
NOfPToStay 137, 138, 163, 177–181, 184, 185
NOfRecv 138, 157, 159, 163, 171, 174–176, 184, 197, 198, 200, 203, 209, 216, 217, 256, 336, 337, 340, 348, 360, 371, 395, 409, 410, 423, 426, 456, 457, 463, 476, 488, 491, 517, 526, 527, 536
NOfSend 138, 157, 159, 163, 171, 174–176, 184, 197, 198, 200, 201, 203, 209, 216, 217, 256, 336, 337, 340, 348, 360, 370, 389, 400, 409, 410, 417, 418, 423–426, 456, 457, 463, 464, 476, 486, 488, 491, 505, 507, 516, 517, 519, 525, 527, 534–536
nOfSpecies 136, 159, 162, 170, 172, 175, 185, 189, 193, 197, 203, 216, 237, 239, 240, 244, 245, 248, 253, 254, 258, 260, 262–267, 355, 362, 364, 367, 372, 375, 381, 384, 387, 391, 394, 395, 399, 403, 409, 410, 412, 413, 416, 418, 420, 422, 423, 425, 426, 430, 436, 438, 470, 480, 484, 487, 492, 493, 496, 499, 501, 506, 507, 511, 514, 517, 520, 523, 527, 530, 531, 533–537, 543, 545, 548, 550–552
normal accommodation 26, *37*, 38, 135, 152, 162, *173*, 184, 189, 205, 206, 211, 216, 239, 240, 243, 244, 254, 324, 325, 328–331, 334, 362, 365–367, 369–371, 376, 377, 382, 385, 391, 401, 402, 404, 411, 413, 415, 416, 437, 445, 454, 483, 484, 486, 492, 497, 513, 514, 518
nphgram 24, *27*, *31*, 36, 43–50, 57, 64, 73, 76, 88, 96, 97, 108, 110, 111, 119, 121, 122, 136, 159, 161, 162, 235, 236, 283, 285, 332, 355, 358, 473
npmax *88*, 469, 473
nspec *27*, *30*, 44, 50, 57, 64, 76, 88, 110–112, 121–123, 136, 159, 162, 235, 283, 285
NULL 30–35, 44, 57–59, 61, 76, 77, 79, 90, 91, 119, 121, 134, 136, 138, 140, 141, 146, 159–166, 169, 170, 172, 175, 203, 208, 209, 226, 235–237, 247, 251, 258, 272, 284, 285, 287, 289, 298, 315, 331, 356, 358, 360, 361, 363, 371, 372, 385–387, 394, 397, 400, 412, 427, 435, 447, 471, 473–475, 478, 479, 481, 494, 509, 511

O

oh13_init() 48, *62*, 64, 65, 106, 270, 277, 285, 287, 311
oh13_init_() 277, 285, 287
oh1_accom_mode() 26, *37*, 107, 135, 152, 211
oh1_accom_mode_() 152, 211
oh1_all_reduce() 26, 38, *39*, 107, 145, 152, 212, 320, 374
oh1_all_reduce_() 152, 212
oh1_broadcast() 26, *38*, 39, 107, 145, 152, 160, 164, 192, 197, 201, 212, 213, 319, 321, 334, 376, 378
oh1_broadcast_() 152, 201
oh1_families() 26, *34*, 36, 107, 146, 152, 169, 175, 208
oh1_families_() 152, 169
oh1_init() 26, *27*, *30*, 33–37, 41, 44, 45, 48, 50, 57, 62–64, 76, 77, 88–90, 98, 101–105, 107, 133, 134, 136, 138, 146, 147, 151, 152, 154, 158, 162, 235, 236, 241, 283, 285
oh1_init_() 133, 146, 152, 158, 162, 235, 236, 283, 285
oh1_init_stats() 26, 99, *101*, 102, 107, 150–152, 214, 215

oh1_init_stats_() 151, 152, 214
oh1_neighbors() 26, *33*, 34–36, 107, 147, 152, 161, 165, 166, 168, 209
oh1_neighbors_() 152, 168
oh1_print_stats() 26, 99, 101, 102, *103*, 107, 151, 152, 222, 223
oh1_print_stats_() 151, 152, 222, 223
oh1_reduce() 26, 38, 39, *40*, 107, 145, 152, 213, 320, 374, 495
oh1_reduce_() 152, 213
oh1_show_stats() . 26, 99, 101, *102*, 104, 107, 151, 152, 215, 218, 219, 220, 222
oh1_show_stats_() 151, 152, 218
oh1_stats_time() 26, 99, *102*, 107, 149–152, 172, 177, 203, 207, 215, 218, 240, 245, 248, 254, 526, 528, 533–535
oh1_stats_time_() 151, 152, 215
oh1_transbound() 17, 24, 26–28, 30, 31, 33, 34, *36*, 37, 41, 45, 46, 48, 63, 65, 79, 92, 96, 99, 101, 103, 104, 107, 131, 134–136, 138, 152, 154, 159, 164, 171, 172, 184, 209, 231, 238, 277, 311, 343, 350, 362, 459, 466, 480
oh1_transbound_() . 131, 152, 171, 172, 231, 238, 277, 311, 343, 362, 459, 480
oh1_verbose() 26, *105*, 107, 152, 153, 155, 223
oh1_verbose_() 152, 153, 155, 223
oh2_init() 41, *43*, 45, 48, 50, 57, 62, 63, 76, 88, 98, 107, 226–228, 231, 232, 235, 236, 241, 261, 268, 283, 285
oh2_init_() 228, 231, 235, 236, 262, 283, 285
oh2_inject_particle() 41, 42, *46*, 47, 48, 96–98, 106, 107, 138, 168, 226–229, 231, 232, 245, 248, 261, 262, 265, 266, 267, 436
oh2_inject_particle_() 231, 265, 436
oh2_max_local_particles() 41, 44, *45*, 77, 88, 107, 110, 121, 167, 168, 226, 231, 232, 268, 361, 469, 473
oh2_max_local_particles_() 231, 268
oh2_remap_injected_particle() . . . 41, 97, 107, 226–229, 231, 232, 266, 267
oh2_remap_injected_particle_() *46*, 231, 266
oh2_remove_injected_particle() . . 41, 97, 107, 226–229, 231, 232, 266, 267
oh2_remove_injected_particle_() *47*, 231, 267
oh2_set_total_particles() 41, *47*, 48, 97, 107, 138, 154, 170, 231, 237, 268
oh2_set_total_particles_() 170, 268
oh2_transbound() 17, 24, 41–44, *45*, 47, 48, 65, 79, 96, 97, 99, 100, 104, 107, 134, 231, 232, 237, 238, 268, 277, 311, 362, 480
oh2_transbound_() 231, 238, 277, 311, 362, 480
oh3_allreduce_field() 48, 53, 60, *69*, 70, 107, 112, 123, 275, 277, 320
oh3_allreduce_field_() 277, 320
oh3_bcast_field() 48, 53, 54, 60, *68*, 107, 111, 122, 275, 277, 319, 334
oh3_bcast_field_() 277, 319
oh3_exchange_borders() 49, *70*, 107, 111, 112, 122, 123, 134, 135, 273, 276, 277, 302, 309, 320, 334, 335, 372, 454, 494
oh3_exchange_borders_() 277, 320
oh3_grid_size() 48, *64*, 107, 270, 272, 277, 310
oh3_grid_size_() 277, 310
oh3_init() . . 22, 48, *49*, *57*, 62–67, 69–71, 74, 77, 89, 98, 107, 108, 110, 119–121, 228, 270, 271, 273–277, 280, 282, 283, 284, 285, 287, 297, 351, 355, 467
oh3_init_() 228, 277, 283, 284, 285, 287, 355
oh3_map_particle_to_neighbor() 48, 64, *65*, 67, 79, 107, 114, 124, 134, 147, 270, 272, 273, 277, 278, 312, 313, 429
oh3_map_particle_to_neighbor_() 277, 313, 429
oh3_map_particle_to_subdomain() 48, 64, 65, *67*, 81, 107, 272, 277, 278, 295, 314, 315, 317, 434
oh3_map_particle_to_subdomain_() 277, 315, 434
oh3_map_region_to_adjacent_node_() 277, 313
oh3_map_region_to_node_() 277, 315
oh3_reduce_field() 48, 53, 60, *70*, 107, 275, 277, 319, 320
oh3_reduce_field_() 277, 319, 320
oh3_transbound() 24, 48, 49, 63, *65*, 79, 96, 97, 99, 100, 104, 107, 111, 122, 134, 277, 282, 311, 362, 480
oh3_transbound_() 277, 311, 362, 480
oh4p_init() 72, 74, *75*, 76–81, 86, 89, 107, 328, 332, 342, 343, 346, 354, 356, 469
oh4p_init_() 343, 354, 356, 361, 469
oh4p_inject_particle() 42, 74, 75, *81*, 82, 94, 96–98, 107, 332, 333, 343, 350, 429, 436, 437, 440, 551
oh4p_inject_particle_() . . . 343, 436, 551
oh4p_map_particle_to_neighbor() . 73–75, *79*, 81–83, 93, 97, 98, 107, 237, 323, 327–329, 331–333, 343, 350–352, 427, 428, 429, 434, 436–438, 440, 548
oh4p_map_particle_to_neighbor_() 343, 429, 548

oh4p_map_particle_to_subdomain() 73–75, 79, *81*, 82, 83, 94, 97, 98, 107, 323, 327–329, 331–333, 342, 343, 350–352, 427–429, 431–433, 434, 437, 439, 550
oh4p_map_particle_to_subdomain_() 343, 434, 550
oh4p_max_local_particles() 74, *77*, 88, 107, 332, 338, 343, 350, 356, 358, 361, 469, 473, 474
oh4p_max_local_particles_() 343, 361
oh4p_per_grid_histogram() 74, *78*, 79, 91, 107, 331, 334, 343, 355, 361, 479
oh4p_per_grid_histogram_() . 343, 361, 479
oh4p_remap_particle_to_neighbor() 74, 75, *83*, 95, 97, 107, 343, 438, 552
oh4p_remap_particle_to_neighbor_() 343, 438, 552
oh4p_remap_particle_to_subdomain() 74, 75, *83*, 95, 98, 107, 343, 429, 434, 439, 553
oh4p_remap_particle_to_subdomain_() 343, 439, 553
oh4p_remove_mapped_particle() 74, 75, *82*, 83, 94, 97, 98, 107, 327–329, 332, 333, 341, 343, 350, 416, 427, 437, 438, 439, 552
oh4p_remove_mapped_particle_() 343, 437, 552
oh4p_transbound() 24, 72–76, *79*, 82, 97–99, 107, 323, 329, 332, 343, 346, 360, 362
oh4p_transbound_() 343, 362
oh4s_exchange_border_data() 86, 89, *92*, 107, 446, 448, 449, 459, 474, 475, 479, 542, 545
oh4s_exchange_border_data_() . . . 459, 542
oh4s_init() 84, 86, *87*, 88–92, 107, 447, 452, 459, 462, 469, 470, 471, 478
oh4s_init_() 459, 469, 470, 471, 479
oh4s_inject_particle() 42, 86, *94*, 98, 107, 440, 452, 453, 460, 466, 551
oh4s_inject_particle_() 460, 551
oh4s_map_particle_to_neighbor() 86, *93*, 98, 107, 440, 443, 444, 451–453, 459, 460, 466, 467, 548, 551, 552
oh4s_map_particle_to_neighbor_() 459, 548
oh4s_map_particle_to_subdomain() . 86, 87, 93, *94*, 98, 107, 443, 444, 451–453, 459, 460, 466, 467, 548, 550, 553
oh4s_map_particle_to_subdomain_() 460, 550
oh4s_particle_buffer() 86, 88, *90*, 91, 92, 107, 452, 453, 459, 466, 469, 471, 472, 474, 478
oh4s_particle_buffer_() 459, 478
oh4s_per_grid_histogram() . 85, 86, *91*, 92, 107, 446, 447, 451, 459, 466, 470, 479
oh4s_per_grid_histogram_() 459, 479
oh4s_remap_particle_to_neighbor() 86, *95*, 98, 107, 460, 552
oh4s_remap_particle_to_neighbor_() 460, 552
oh4s_remap_particle_to_subdomain() 87, *95*, 98, 107, 460, 553
oh4s_remap_particle_to_subdomain_() 460, 553
oh4s_remove_mapped_particle() 86, 87, *94*, 98, 107, 443, 444, 452, 453, 458, 460, 466, 524, 552, 553
oh4s_remove_mapped_particle_() . 460, 552
oh4s_transbound() 24, 84–87, 90, 91, *92*, 93, 98, 107, 448, 449, 459, 462, 475, 476, 479, 480, 492
oh4s_transbound_() 459, 480
oh_accom_mode() (function alias) . . . *107*, 153
oh_all_reduce() (function alias) . . . *107*, 153
oh_allreduce_field() (function alias) *107*, 112, 123, 278
oh_bcast_field() (function alias) *107*, 111, 122, 278
OH_BIG_SPACE . . 25, 42, 73, 326, 359, 443, 474
oh_broadcast() (function alias) . . . *107*, 153
oh_config.h (C header file) 18, 25, 42, 73, 74, 106, 129, 130, 132, 153, 325, 326, 442, 554, 555
OH_CTYPE_FROM 276, 299, 357, 472
OH_CTYPE_N 276, 287, 297, 298, 302, 356, 357, 471, 472
OH_CTYPE_SIZE 276, 299, 357, 472
OH_CTYPE_TO 276, 299, 357, 472
OH_DEFINE_STATS 100, 148, 157
OH_DIM_X 133, 287, 288, 292, 297, 301, 302, 304–306, 308, 313, 315, 317, 321, 359, 387, 402, 403, 430, 435, 436, 474, 513, 514, 543, 545, 548, 550
OH_DIM_Y 133, 292, 297, 301, 302, 304–306, 308, 313, 315, 351, 352, 359, 373, 387, 391, 402, 403, 430, 435, 436, 474, 513, 514, 548, 550
OH_DIM_Z 133, 292, 297, 301, 302, 305, 308, 313, 315, 351, 352, 359, 373, 387, 391, 402, 403, 430, 435, 436, 474, 475, 477, 506, 513, 514, 548, 550
OH_DIMENSION 25, 27, 43, 49, 65, 67, 75, 87, 108, 110, 113, 115, 117–121, 123, 124, 126–130, 132, 133, 153, 278, 279, 287–290, 292–294, 296–298, 301, 302, 304–306,

309, 310, 313, 315, 318, 321, 351, 352, 359, 373, 403, 442, 472, 478, 514
oh_exchange_border_data() (function alias) *107*, 460
oh_exchange_borders() (function alias) *107*, 111, 112, 122, 123, 278
oh_families() (function alias) *107*, 153
OH_FTYPE_BL 274, 299, 300, 357, 472
OH_FTYPE_BU 274, 301, 357, 472
OH_FTYPE_ES 274, 298, 299, 301, 357, 472
OH_FTYPE_LO 274, 299, 357, 472
OH_FTYPE_N 274, 287, 297, 298, 301, 356, 357, 471, 472
OH_FTYPE_RL 274, 300, 357, 472
OH_FTYPE_RU 274, 299, 301, 357, 472
OH_FTYPE_UP 274, 299, 357, 472
oh_grid_size() (function alias) . . . *107*, 278
OH_HAS_SPEC 42, 229, 265–267, 437, 551
oh_init() (function alias) *107*, 110, 121, 153, 232, 279, 343, 460
oh_init_stats() (function alias) . . . *107*, 153
oh_inject_particle() (function alias) *107*, 231, 343, 460
OH_LIB_LEVEL 25, 106, 108, 119, 153, 231, 232, 278, 343
OH_LIB_LEVEL_4P 25, 325, 343, 442, 460
OH_LIB_LEVEL_4PS 25, 132, 442
OH_LIB_LEVEL_4S 25, 442
OH_LOWER 270, 287, 288, 290, 292–294, 296, 299–302, 304–306, 308–310, 312, 315, 318, 321, 357, 387, 401, 403, 428, 429, 432–434, 472, 475, 477, 480, 502, 506, 508–510, 512, 514, 520, 523, 539, 543, 545–547, 549
oh_map_particle_to_neighbor() (function alias) *107*, 114, 124, 278, 279, 343, 460
oh_map_particle_to_subdomain() (function alias) *107*, 278, 279, 343, 460
oh_max_local_particles() (function alias) *107*, 110, 121, 231, 343
oh_mod1.F90 (Fortran module file) 26, 38, 129, 130, 154, 554
oh_mod1.o (Fortran object file) 554
oh_mod2.F90 (Fortran module file) 41, 129, 130, 232, 554
oh_mod2.o (Fortran object file) 554
oh_mod3.F90 (Fortran module file) 49, 108, 129, 130, 278, 554
oh_mod3.o (Fortran object file) 554
oh_mod4p.F90 (Fortran module file) 75, 93, 129, 130, 344
oh_mod4p.o (Fortran object file) 554
oh_mod4s.F90 (Fortran module file) 87, 129, 130, 460
oh_mycomm 28, 43, 49, 63, 75, 87, 108, 130, 146
OH_NBR_BCC 442, 502
OH_NBR_SELF . 325, 384–386, 392–394, 398, 430, 434, 442, 501, 502, 507, 548, 550
OH_NBR_TCC 442, 502
OH_NEIGHBORS 133, 144, 147, 164–167, 173, 191, 192, 195, 209, 241, 288, 313, 325, 360, 367, 377, 378, 384, 385, 392, 394, 395, 397, 408, 442, 474, 477, 484, 498, 501, 502, 507, 513, 516, 520, 547, 550
oh_neighbors() (function alias) . . . *107*, 153
OH_nid_t 42, 229, 230, 326, 359, 414, 415, 417, 424, 438, 443, 474, 525, 528, 534, 552
OH_NO_CHECK 25, 74, 427, 546
oh_part.h (C header file) 42, 106, 129, 130, 226, 231, 326, 554, 555
oh_particle 42, 43, 44, 46, 47, 49, 73, 75, 76, 80, 82–84, 91, 98, 108, 113, 114, 130
oh_particle_buffer() (function alias) *107*, 460
oh_per_grid_histogram() (function alias) *107*, 343, 460
OH_PGRID_EXT 325, 355, 357, 359, 362, 372, 387, 402, 410, 428, 434, 442, 470–472, 480, 492, 493, 504, 506, 507, 511, 514, 539, 543, 545, 547, 549
OH_POS_AWARE 132, 163, 229, 230, 238, 262, 298, 300, 325, 327, 358, 442
oh_print_stats() (function alias) . . *107*, 153
oh_reduce() (function alias) *107*, 153
oh_reduce_field() (function alias) . *107*, 278
oh_remap_injected_particle() (function alias) *107*, 231
oh_remap_particle_to_neighbor() (function alias) *107*, 343, 460
oh_remap_particle_to_subdomain() (function alias) *107*, 343, 460
oh_remove_injected_particle() (function alias) *107*, 231
oh_remove_mapped_particle() (function alias) *107*, 343, 460
oh_set_total_particles() (function alias) *107*, 231
oh_show_stats() (function alias) . . . *107*, 153
oh_stats.h (C header file) 98, 99, 129, 130, 132, 147, 325, 442, 554, 555
oh_stats_time() (function alias) . . . *107*, 153
oh_transbound() (function alias) *107*, 111, 122, 153, 232, 279, 343, 460
oh_type (Fortran module) 28
oh_type.F90 (Fortran module file) 28, 42, 129, 130, 146, 326, 554
oh_type.o (Fortran object file) 554

OH_UPPER 270, 287, 288, 290, 292–294, 296, 299–302, 304, 305, 309, 310, 312, 315, 318, 321, 357, 387, 401, 403, 428, 429, 432, 434, 472, 475, 477, 480, 502, 506, 508–510, 512, 514, 520, 523, 539, 543, 545–547, 549
oh_verbose() (function alias) *107*, 153
ohhelp1 (Fortran module) 26
ohhelp1.c (C source file) *17*, 128–130, 134, 146–148, 157, 226, 554, 555
ohhelp1.h (C header file) *17*, 129, 130, 132, 157, 234, 282, 325, 346, 442, 462, 554, 555
ohhelp1.o (C object file) 554, 555
ohhelp2 (Fortran module) 41
ohhelp2.c (C source file) *17*, 129, 130, 134, 234, 554, 555
ohhelp2.h (C header file) *17*, 129, 130, 226, 234, 282, 326, 327, 346, 462, 554, 555
ohhelp2.o (C object file) 554, 555
ohhelp3 (Fortran module) 49, 108
ohhelp3.c (C source file) *17*, 129, 130, 282, 554, 555
ohhelp3.h (C header file) *17*, 128–130, 270, 282, 346, 462, 554, 555
ohhelp3.o (C object file) 554, 555
ohhelp4p (Fortran module) 75
ohhelp4p.c (C source file) *18*, 129, 346, 554, 555
ohhelp4p.h (C header file) *18*, 129, 325, 346, 554, 555
ohhelp4p.o (C object file) 554, 555
ohhelp4s (Fortran module) 87
ohhelp4s.c (C source file) *18*, 129, 462
ohhelp4s.h (C header file) . *18*, 129, 442, 462
ohhelp_c.h (C header file) 18, 25, 26, 31, 41, 49, 75, 87, *106*, 119, 129, 130, 146, 153, 231, 278, 279, 343, 344, 460, 555
ohhelp_f.h (Fortran header file) 18, 25, *106*, 108, 129, 130, 153, 231, 278, 279, 343, 460, 554
Op_StatsPart 151, 214, 220, 221
Op_StatsTime 131, 151, 214, 222, 223
outer extension *302*, 325
outer extent *305*, 306

P

P *11*, 12, 13, 20, 137, 174, 203–205
P_{comm} *89*, 92, 93, 469, 474
P_{halo} *88*, 89, 452, 469, 473
P_{hot} *74*, 77, 78, 324, 332, 338, 343, 356, 361, 379, 380
P_{lim} 44, *45*, 46, 72, 76, *78*, 82, 90–93, 226, 227, 231, 235, 237, 238, 265, 268, 332, 343, 356, 358, 361, 365, 371, 411–413, 415, 418, 419, 422, 425, 437, 452, 459, 469, 471, 478, 479, 489, 526, 532, 533, 535, 536
P'_{lim} . 88, *89*, 90, 452, 469, 471, 473, 474, 478
$P_{\max}$ *11*, 14, 15, 20, 44, 74, 76, 137, 140, 174, 175, 177–179, 181, 182, 186–188, 338
P_{mgn} *89*, 452, 469, 473
P_n *11*, 12–15, 136, 137, 140, 173–175, 177, 178, 181, 186, 203–205, 209, 210, 365, 411–415
P_n^{get} 140, 178–183, 186–188
$P_n^{\min}$ 13, *14*, 15, 140, 177–179, 181, 182, 186, 187
P_n^{put} 140, 177–179
P_n^{send} 365, 370, 371, 384, 385, 389, 390, 398–400, 411–413, 415, 418, 419, 422, 425, 488, 489, 501, 503, 526, 527, 532, 533, 535, 536
$\mathcal{P}_L(p,s,g)$ . . 323, 329, 340, 394, 400, 412, 444
$\mathcal{P}_O(p,s,g)$ 330, 388, 389, 395, 399, 412, 446, 463, 509, 511, 520–524, 538, 539, 543
$\mathcal{P}_T(p,s,g)$. 323, 329, 379, 383, 389, 395, 412, 440, 444, 445, 447, 494, 509, 511
$\mathcal{P}_Z(z)$ 440, 445, 447, 494, 499
parent(n) *15*, 20, 21, 44, 84, 134, 137, 139, 140, 177–181, 184, 185, 189, 191, 193, 194, 197–199, 207, 209, 243, 248–253, 255–257, 260, 262, 265–267, 275, 301, 312, 313, 326, 330, 341, 353, 363, 365–370, 383, 385–387, 389, 390, 398, 399, 401, 402, 404, 412, 427, 430, 435, 437, 446, 449, 458, 468, 481–483, 486, 491, 504, 506, 511–515, 538, 540, 543
Parent_New() 368, 369, 370, 484, 485, 487, 501
Parent_New_Diff() 368, 385, 393, 398, 484, 485, 502
Parent_New_Same() 368, 369, 385, 484
Parent_Old() 368, 384, 393, 398, 484, 485, 501
particle pushing *18*, 21, 24, 100, 106, 113, 123
particle transferring *18*, 20
particle-associated . . . *86*, 89, 92, 93, 447–450, 456, 459, 464, 469, 475, 542–545
particle_push() 21, 22, 106, 111, 113, 120, 122, 123
Particle_Spec() 229, 261, 265–267, 367, 415, 418, 420, 424, 437, 484, 528, 529, 534, 551
Particles 139, 226, 227, 228, 231, 233–235, 237, 240, 242, 245–252, 254, 257–262, 265, 266, 268, 323, 324, 328, 332, 333, 337, 343, 348, 351, 356, 364, 365, 370, 371, 410–416, 418–424, 427, 429, 430, 435, 437, 441, 449, 452, 453,

456, 459, 460, 464, 467, 469, 472, 478, 482, 487, 489, 492, 518, 523, 525, 526, 528–535, 538, 539, 546, 548, 550–552
`pbase` *44*, 45, 50, 57, 76, 88, 108, 110, 111, 119, 121, 122, 227, 235, 236, 237, 283, 285
`pbuf` 21, 22, 24, 43, *44*, 45–50, 57, 72, 75, *76*, 79, 81, 88, 90, *91*, 92, 96, 99, 108, 110–112, 119, 121–123, 226, 235, 236, 237, 283, 285, 332, 356, 469, 472, 473, 478
$pbuf(p, s)$ . . . 226, 227, 234, 240, 245–248, 250–252, 258–261, 323, 328, 363, 367, 410, 412, 414, 415, 418–422, 424, 427, 451, 481, 509–511, 528–532, 534, 540
$pbuf_i(p, s)$ 451, 489, 526–529, 531, 532
`PbufIndex` 328, 358, 363, 427, 444, 474, 481, 546
`pcoord` 29, *30*, 32, *33*, 44, 50, 51, 57, 58, 64, 77, 89, 110, 121, 147, 160, 165, 236, 283, 285, 289
per-grid histogram . 24, *72*, 73, 74, 77–82, 85, 86, 89, 91, 92, 97, 300, 302, 321, *323*, 324–326, 328–331, 333–335, 338, 340, 343, 346–348, 352, 353, 355, 357, 358, 361, 364, 365, 370–375, 378, 381, 384, 387, 401–403, 410–413, 440, 443, 444, 447, 453, 454, 457, 459, 460, 462, 463, 468, 470, 472, 479, 485–488, 490, 492–496, 502, 506, 512, 514, 532, 533
per-grid index *85*, 86, 89, 91, 92, 459, 479, 485, 486, 488, 490, 532, 533
per-plane histogram *440*, 445, 447, 463, 487, 493, 494, 496, 499, 500
`pic()` 110, 120, 121
`Pillar_Lower()` 519, 525, 532
`Pillar_Upper()` 519, 520, 525, 532, 539
`pop_heap()` 142, 157, 204, 210
position-aware particle management *17*, 18, 24, *72*, 74, 79, 84, 86, 92, 130, 132, 135, 147, 164, 176, 179, 206, 209, 229, 230, 238, 262, 265, 300, 321, *323*, 324, 325, 328, 330, 336, 343, 346, 364, 365, 367, 369–371, 376, 378, 382, 391, 409–411, 418, 424, 431, 435, 440, 442, 451, 459, 462, 485, 486, 489, 490, 492, 517, 532
`Primarize_Id()` 229, 230, 262, 327, 367, 415, 437, 443, 484, 552
`Primarize_Id_Only()` 327, 418, 424, 444, 528, 534
primary execution . . . *100*, 101, 102, 214, 215
primary family . . . *38*, 39–41, 53, 60, 68–70, 212, 213, 258, 277, 319, 321, 323, 329, 338, 371, 372, 374, 378, 379, 381, 408, 440, 445, 447, 487, 492–494, 499
primary mode *11*, 12, 27, 30, 34–37, 71, 99, 103, 104, 113, 123, 134, 135, 137–139, 148, 155, 158, 162, 170, 171, 174–176, 189, 192, 195, 197, 203, 206, 216, 232–234, 239, 240, 242, 244, 251, 253, 254, 311, 323, 329–331, 346, 348, 350, 351, 354, 363–366, 372, 410, 411, 413, 441, 445, 456, 457, 462, 466, 467, 469, 477, 481, 482, 485, 487, 490, 493, 500, 517, 518, 526, 528, 532, 533, 535
primary particle . . . *11*, 12, 15, 17, 19–21, 24, 27, 28, 31, 35, 37, 44, 66, 72, 74, 76, 79–85, 90–92, 100, 104, 111–114, 122–124, 136–140, 142, 148–150, 157, 159, 173, 176–179, 181–184, 188–190, 193, 194, 199–201, 205, 216, 220, 226–229, 235, 240, 242, 244–255, 257, 258, 260, 262, 265, 323, 326, 327, 329, 330, 332, 335, 337, 338, 340, 341, 361, 365, 367, 369–371, 375–377, 381, 383, 389, 404, 411, 412, 414–416, 419, 420, 424, 434, 436, 437, 440, 441, 444–446, 448, 450, 452, 454, 456, 457, 482, 486, 488–492, 497, 499–502, 505, 507, 525, 531, 534, 538, 540
primary receiving block (of `CommList`) *143*, 191, 192, 197, 198, 255, 263, *335*, 336, 339, 341, 347, 375–379, 382, 383, 385, 395, 441, *455*, 463, 464, 477, 488, 491, 496–498, 500–502, 505, 507, 518
primary sending block (of `CommList`) *143*, 144, 191, 192, 197, 198, 255, 264, *335*, 336, 347, 375, 377, 378, 383–385, 387, 389, *455*, 463, 477, 488, 496, 498, 500–502, 505, 518, 520, 523
primary subcuboid 89, 90, 92, 93, *440*, 441, 447–450, 452, 463, 475, 480, 490, 502, 518, 520, 523, 539
primary subdomain 10, *11*, 12, 19, 20, 22, 27, 28, 30–33, 35, 37, 48, 61, 65–72, 79–82, 84, 89, 91, 100, 111–115, 117, 118, 122–125, 127, 128, 134, 136–138, 143, 146–148, 150, 157, 159, 173, 177, 179, 180, 189–191, 193, 206, 227, 229, 230, 234, 239, 240, 242, 244–250, 253, 255, 257, 258, 262, 265, 273–277, 300–302, 304, 312, 313, 319–321, 323, 326, 331, 335–337, 339–343, 347, 348, 353, 358, 360, 364–366, 370, 371, 376, 381, 382, 384, 386–391, 393, 395, 398, 399, 403, 408, 409, 411, 414,

416, 418–420, 422, 424, 429, 431, 436, 437, 440, 441, 447–450, 455, 457–460, 462, 463, 468, 474, 475, 477, 482, 487, 490–493, 497, 500–502, 504–506, 514, 517, 518, 524, 525, 529, 530, 538
`PrimaryCommList` 456, 457, 463, 477, 482, 488, 490, 491, 500, 502, 506, 507, 513, 518
`primaryParts` . . . 138, 154, 170, 172, 227, 231, 235, 239, 240, 246, 247, 251, 252, 258, 260, 268, 333, 362, 366, 411, 419, 420, 424, 453, 480, 492, 528, 529, 534
`PrimaryRLIndex` 457, 477, 488, 500, 518
`print_stats()` 148, 150, 151, 158, 218, 219, 222, 223
`push_heap()` 137, 142, 157, 203, 204, 209

Q

$q(n)$ 136, 137, 170, 172, 173, 175, 176, 191, 193, 205, 241, 242, 245, 365
$q^{\text{inj}}(n)$ 137, 138, 239, 244, 247, 249, 253, 265, 363, 414, 481
Q_n *11*, 12–16, 44, 46, 47, 82, 138, 172, 173, 187, 227, 235, 265, 266, 371, 415, 417–419, 422, 424, 425, 427, 437, 489, 500, 507, 511, 525–528, 532, 533, 535, 536
Q_n^{get} . . . 140, 177–183, 186, 187, 189–191, 194, 199, 204, 205, 249, 375, 382, 497
Q_n^{inj} 228, 238, 239, 244, 246, 247, 250, 252, 261, 265, 266, 363, 367, 415, 418, 419, 424, 427, 437, 481
Q_n^m . . *11*, 12, 14, 15, 20–22, 44, 138, 139, 140, 172, 173, 177–182, 184, 186, 187, 227, 235, 248, 249, 251, 252, 258, 260, 376, 379, 382, 383, 419, 420, 424, 497, 499, 500, 507, 509–511, 527–529, 534
Q_n^{stay} 140, 178, 179, 181
$\mathcal{Q}_n^m$ 379, 380
`qsort()` . . . 132, 157, 187, 188, 282, 295, 296

R

R_n *15*, 16, 181, 182, 186–188
R_n^{flt} *15*, 16, 181, 182, 185–188
R_n^{get} 181, 182, 183, 189, 190, 194, 199, 205, 249, 375, 497
rank() *28*, 30, *32*, 50, 51, 58, 160, 165, 288, 291, 315
$rbuf(p, s)$ . . . 227, 240–243, 245–248, 251, 252, 254–256, 258, 262, 332, 348, 365, 371, 414, 416, 419–421, 423, 426, 453, 464, 489, 490, 526–529, 531, 534, 535
`rcounts` 24, *28*, *31*, 36, 45, 63, 64, 138, 159, 161, 163, 236, 237, 285, 355, 358, 473
`RealDstNeighbors` 324, 341, 342, 347, 354, 360, 364, 369, 378, 404–410, 418, 423, 425, 426, 458, 463, 477, 482, 486, 491, 515–517, 527, 535, 536
`RealSrcNeighbors` 324, 341, 342, 347, 354, 360, 364, 369, 378, 404–409, 418, 423, 425, 426, 458, 463, 477, 482, 486, 491, 515–517, 527, 534, 536
`rebalance1()` 134, 137, 139–142, 145, 146, 148, 152, 155, 159, 174–176, 179, 189–193, 197, 203, 205, 206, 209, 210, 215, 234, 244, 256, 346, 350, 367, 369, 370, 382, 462, 467, 483, 486
`rebalance2()` 141, 144, 174, 203, 234, 239, 244, 253–256, 369, 486
`rebalance4p()` 327, 328, 331, 333, 341, 342, 346, 350, 362, 367, 369, 401, 402, 483, 484
`rebalance4s()` 440, 443, 444, 450, 453, 458, 462, 467, 480, 482, 483, 485, 486, 513
`receive_particles()` 143, 227, 228, 234, 254, 255, 263, 264
receiving plane *276*, 277, 284, 299, 302–305, 321, 325, 326, 334, 346, 363, 372, 373, 443–445, 450, 454, 462, 470, 493–495
`RecvBufBases` 227, 228, 238, 240–242, 245–248, 251, 252, 254–256, 258, 259, 262, 263, 332, 333, 371, 413, 414, 416, 419–421, 423, 426, 453, 528–531, 534–536
`RecvBufDisps` 228, 238, 243, 256
`RecvCounts` 138, 152, 159, 163, 171, 174, 198
`reduce_population()` 329, 330, 334, 346, 370, 372, 374, 445, 454, 462, 487, 493, 494
`RegionId` 134, 159, 162, 175, 209, 266, 267, 302, 311, 313, 364, 366, 367, 393, 427, 430, 435, 437, 482–484, 543, 546, 550, 552
regular process coordinate *271*, 272, 282, 284, 287–289, 314, 315, 360, 433, 435, 478
`remove_heap()` 137, 142, 157, 204, 210
`repiter` 30, 33, 44, 57, 62, 64, 77, 90, *101*, 102–104, 151, 160, 167, 236, 284, 285
`reportIteration` . . . 151, 152, 160, 167, 218
`Requests` 228, 238, 256, 263, 264, 337, 386, 392, 395, 398, 409, 410, 427, 456, 517, 536, 538, 539, 541, 544, 546
`reshape()` 50, 51, 56, 110
`RLIndex` 144, 190, 191, 195, 335, 336, 338, 341, 377, 378, 384, 454, 455, 457, 488, 500, 518
`rotate_b()` 108, 109, 118, 120, 127

rotate_e() 108, 109, 118, 120, 128
Round() 219, 220, 222

S

S *20*, 24, 27, 28, 30, 31, 42, 46, 47, 72, 79, 80, 82–84, 91, 92, 121, 136, 137–139, 142, 143, 159, 162–164, 170, 172, 177, 184, 185, 190–194, 200, 201, 205, 217, 226–229, 237–240, 245–248, 250–252, 256, 258, 260–263, 265–267, 328–330, 332, 334–337, 339, 340, 355, 358–361, 363, 365, 367, 372, 373, 376, 379, 381, 383, 384, 388, 389, 391, 394–397, 399, 403, 409–420, 422–427, 437, 444–446, 448, 451–457, 470, 474, 475, 479, 481, 483, 493–495, 497, 499, 501, 502, 505–511, 521–527, 530, 532, 534, 536, 538, 541, 543, 546, 590
S_bcomm 276
S_borderexc 276
S_brdesc 274
S_commlist 142, 144, 164, 190, 193, 194, 198, 200, 201, 255, 263, 264, 335, 338, 376, 379, 380, 382, 383, 388, 389, 394, 395, 397, 400, 417, 454, 456, 477, 482, 497, 500, 502, 506, 507
S_commsched_context 144, 190
S_flddesc 274
S_grid 271
S_griddesc 331, 450
S_heap 131, 141, 142
S_hotspot 338, 339, 384, 386, 388
S_hotspotbase 339
S_hplane 448
S_interiorp 451
S_message 272
S_mycommc 31, *32*, 106, 145, 146, 159, 161, 236, 285
S_mycommf 146, 159
S_node 139, 140, 141
S_particle 22, 42, 43, 44, 46, 47, 57, 63, 73, 76, 80, 82–84, 91, 98, 106, 119, 123, 124, 226–229, 231, 235–237, 261, 285, 325, 429, 434, 437, 442, 450
S_realneighbor 341, 458
S_recvsched_context 338, 376, 379–381, 457, 497, 499
S_stats 150
S_statscurr 149, 150
S_statspart 150, 151
S_statstime 150, 151, 214
S_statstotal 150, 157, 158, 215
S_subdomdesc 272
S_vplane 448, 475
sample (Fortran module) 108
sample.c (C source file) . . . 119, 128–130, 555
sample.F90 (Fortran source file) 108, 128–130, 554
sample.o (C object file) 555
sample.o (Fortran object file) 554
samplec.mk (make file) *128*, 130, 554, 555
samplef.mk (make file) *128*, 130, 554
sbuf(p, s, n) 337, 348, 371, 418, 423, 425, 426, 456, 464, 527, 534, 536
sbuf(s, n) . . . 227, 240–242, 247, 252–254, 256, 261, 262, 337, 365, 414, 415, 425
scatter() 108, 109, 115, 120, 125
scatter_hspot_recv() 324, 337, 338, 347, 369, 386, 397, 398, 399, 485
scatter_hspot_recv_body() 329–331, 333, 337, 339, 340, 347, 397, 398, 399, 420, 485
scatter_hspot_send() 324, 329, 331, 333, 337–340, 347, 375, 386, 388, 395, 420, 485
sched_comm() 136, 139, 141–144, 147, 157, 168, 190, 191, 193, 206
sched_recv() 329, 331, 333, 338, 347, 352–354, 375, 379, 380, 381, 445, 447, 451, 457, 463, 466, 496, 497, 499
Sched_Recv_Check() . 338, 379, 380, 382, 383
Sched_Recv_Return() 354, 379, 380, 383
schedule_particle_exchange() . . . 136, 139, 141, 143, 144, 147, 157, 184, 189, 193, 195, 197, 201, 204–206, 369, 382
scoord 50, *51*, *58*, 59, 64, 77, 89, 110, 121, 284, 285, 289
scounts 24, *28*, *31*, 35, 36, 45, 63, 64, 138, 159, 161, 163, 236, 237, 285, 355, 358, 473
sdid *27*, *30*, 44, 50, 57, 64, 76, 88, 106, 108, 110–112, 119, 121–123, 134, 159, 161, 162, 235, 236, 283, 285
sdoms 21, 22, *50*, 51, 52, 56, *57*, 58, 59, 61, 64, 66–71, 77, 80, 89, 108, 110–112, 119, 121–123, 270, 271, 282, 284, 285, 287–289, 291, 293
Secondarize_Id() 328, 367, 431, 432, 436, 444, 484, 548, 550
secondary execution . *100*, 101, 102, 214, 215
secondary family *38*, 39–41, 68, 69, 212, 213, 258, 277, 319, 321, 374, 378, 408, 494, 495
secondary mode *11*, 12, 13, 24, 27, 30, 33–35, 37, 71, 103, 104, 112, 113, 123, 134, 135, 139, 142, 144, 148, 158, 171, 174, 175, 189, 192, 195, 197, 198, 204–206, 216, 220,

233, 234, 240, 242, 244–248, 250–255, 321, 329, 330, 335, 346, 348, 351, 363, 364, 369, 371, 372, 378, 391, 407, 409–411, 413, 440, 441, 445, 454, 462, 467, 481–483, 485, 487, 490–493, 496, 517, 526, 528, 532, 533, 535, 536
secondary particle *11*, 12, 14, 17, 19–21, 24, 27, 28, 31, 33, 35, 37, 44, 66, 72, 74, 76, 79–85, 90–92, 100, 104, 106, 111–114, 122–124, 136–140, 142, 144, 148, 150, 157, 159, 173, 176–180, 183, 184, 189–191, 193, 194, 198–201, 204, 205, 216, 226–230, 235, 242, 245–249, 251–255, 257, 258, 260, 262, 265, 323, 326–330, 332, 335, 337, 338, 340, 341, 361, 365, 367, 369–371, 375–377, 381, 383, 386, 389, 404, 412, 414–416, 418–420, 422, 424, 425, 434, 436, 437, 441, 444–446, 448, 450, 452, 454, 456, 457, 482, 483, 486, 488–490, 497, 499, 501, 502, 505, 507, 525, 526, 528, 531, 533, 534, 536–538, 540
secondary receiving block (of `CommList`) *143*, 144, 197, 198, 243, 244, 254, 255, 263, *336*, 338, 347, 375–378, 383, 385, 389, *455*, 457, 463, 477, 488, 496, 500–502, 506, 507, 518
secondary sending block (of `CommList`) *143*, 144, 197, 198, 255, 264, *336*, 338, 347, 375, 377, 378, 383–385, 387, 389, *455*, 457, 463, 477, 488, 496, 500–502, 506, 518, 520, 523
secondary subcuboid . 89, 90, 92, 93, *440*, 441, 446–450, 452, 463, 475, 480, 489, 490, 502, 518, 520, 523, 537, 539–541
secondary subdomain 10, *11*, 12, 13, 19, 22, 27, 28, 30, 31, 33–35, 37, 48, 61, 65–72, 79–82, 84, 89, 100, 111–115, 117, 118, 122–125, 127, 128, 134, 137, 138, 143, 147, 148, 150, 159, 173, 175, 177, 179, 180, 191, 193, 229, 230, 234, 244, 245, 248, 249, 251, 253, 255–258, 262, 265, 274–277, 280, 301, 302, 304, 309, 311–313, 319–321, 323, 326–328, 331, 336, 337, 339–343, 347, 348, 351, 353, 363, 365, 367, 370–372, 378, 384–390, 392, 393, 398, 399, 401–403, 408–413, 415, 416, 418–420, 422, 424, 425, 429, 431, 436, 437, 440, 441, 446, 449, 450, 455, 456, 458–460, 462, 463, 467, 468, 482–484, 487, 489, 491, 492, 494, 501, 502, 504–506, 512, 514, 517, 518, 524–527, 529, 530, 532, 536, 538
`Secondary_Injected()` 328, 367, 418, 424, 444, 484, 528, 534
`secondaryBase` 227, 235, 237, 240, 252, 333, 366, 420, 424, 453, 529, 534
`SecRLIndex` 336, 338, 378, 384, 455, 457, 488, 500, 518
`SecRList` 144, 197, 244, 335, 336, 369, 376, 378, 384, 454, 455, 487, 488, 500, 518
`SecRLSize` 144, 197, 244, 369, 487
`SecSLHeadTail` . . 144, 192, 197, 243, 255, 487
`send_particles()` 143, 227, 228, 234, 254, 255, 263, 264
`SendBuf` 139, 226, 227, 228, 233–235, 238, 240–242, 245–252, 254, 256–258, 260–264, 323, 324, 330, 332, 333, 337, 348, 351, 356, 364, 365, 371, 411–422, 424, 425, 441, 446–448, 452, 453, 456, 459, 463, 464, 467, 469, 472, 478, 482, 489, 490, 507, 509, 510, 521, 523, 525–536, 539, 540, 546
`SendBufDisps` 227, 233, 238, 240–242, 245–248, 250, 252–256, 258, 260, 262–264, 333, 337, 351, 414, 415
`SendCounts` 138, 152, 159, 163, 171, 174, 198
sending plane *276*, 277, 284, 299, 302–305, 321, 325, 326, 334, 336, 339, 363, 365, 372, 377, 391, 443, 445, 450, 453, 454, 470, 494
`set_border_comm()` 275, 276, 282, 297, 302, 303, 304
`set_border_exchange()` 134, 270, 273, 276, 282, 300, 302, 304, 309, 321
`set_field_descriptors()` 270, 274, 275, 280, 297, 300, 301, 311, 351, 403, 467, 512
`set_grid_descriptor()` 331, 347, 351, 358, 367, 369, 378, 402, 450, 463, 467, 474, 484, 486, 513
`set_sendbuf_disps()` 138, 228, 233, 246, 247, 250, 252, 262, 348, 351, 414, 415, 424, 464
`set_sendbuf_disps4p()` 337, 342, 348, 418, 423, 424, 535
`set_sendbuf_disps4s()` 456, 458, 464, 526, 527, 533, 534, 535
`set_total_particles()` 135, 137, 138, 154, 170, 172, 268
shadow *134*, 138, 146, 159, 160, 170, 174, 175, 198, 208, 209, 227, 235, 252, 270, 273, 284, 287, 288, 297, 298, 333, 420, 424, 446–448, 453, 479–481, 485, 488, 529, 534
simulator.c (C source file) 128, 129, 555
simulator.F90 (Fortran source file) 128, 129, 554
simulator.o (C object file) 555

simulator.o (Fortran object file) 554
SLHeadTail 144, 191, 192, 197, 243, 255, 369, 487
Sort_Particle() 443, 450, 520, 531, 532, 535
sort_particles() 324, 327, 329–333, 348, 353, 364, 365, 371, 411, 441, 443, 446, 450–453, 464, 468, 485–490, 520, 521, 526, 531, 532, 538
sort_received_particles() . . 227, 323, 327, 330, 332, 333, 348, 365, 371, 415, 423, 441, 443, 446, 450, 452, 453, 464, 485, 488, 490, 520, 521, 531, 535, 538
specBase 228, 236, 261, 265–267, 285, 355, 367, 413, 418, 423, 427, 437, 470, 484, 488, 527, 534, 546, 551
Special_Pexc_Sched() . . . 176, 184, 205, 209
species 17, *20*, 21, 24, 27, 28, 30, 31, 35, 42, 46, 72, 79–85, 91, 92, 111, 113, 114, 122–124, 136–138, 142, 144, 159, 190, 193, 194, 199–201, 226–229, 236, 240–242, 245, 246, 248–250, 252, 253, 255, 256, 262, 285, 323, 329, 330, 335–337, 339, 340, 388, 389, 391, 393–397, 403, 414–416, 418, 420, 424, 429, 434, 437, 444–446, 448, 456, 495, 502, 507, 508, 521, 525, 531, 534, 540
SPH method 17, 130, 440
sprintf() 224
SrcNeighbors 146, 147, 166, 192, 240, 341, 342, 360, 377, 384, 458, 459, 476, 477, 501
standard header files:
  float.h 132
  limits.h 132
  mpi.h 106, 132, 146
  stdarg.h 132
  stdio.h 132
  stdlib.h 119, 132
  string.h 132
statistics 17, 26, 30, 33, 37, 98, 99, 101–104, 129, 130, 132, 147–152, 157, 158, 160, 167, 171, 175, 198, 214–223
Stats 150, 214–223
stats 30, 33, 37, 44, 57, 62, 64, 77, 90, *101*, 102–104, 151, 160, 167, 236, 284, 285
stats_comm() 150, 158, 216, 217
STATS_CURRENT_SCATTERING 100
STATS_FIELD_SOLVING 100
STATS_PART_GET_PRI_MAX 148, 217, 221
STATS_PART_GET_PRI_MIN 148, 217, 221
STATS_PART_GET_SEC_MAX 148
STATS_PART_GET_SEC_MIN 148
STATS_PART_MOVE_PRI_AVE 148, 217, 220, 221
STATS_PART_MOVE_PRI_MAX . . . 148, 217, 221
STATS_PART_MOVE_PRI_MIN . . . 148, 217, 221
STATS_PART_MOVE_SEC_AVE 148, 220
STATS_PART_MOVE_SEC_MAX 148
STATS_PART_MOVE_SEC_MIN 148, 217, 218, 220, 221
STATS_PART_PG_PRI_AVE . . 148, 217, 220, 221
STATS_PART_PG_SEC_AVE 148, 220
STATS_PART_PRIMARY 148, 215, 216, 220
STATS_PART_PUT_PRI_MAX 148, 217, 221
STATS_PART_PUT_PRI_MIN 148, 217, 221
STATS_PART_PUT_SEC_MAX 148
STATS_PART_PUT_SEC_MIN 148
STATS_PART_SECONDARY . . . 148, 215, 216, 220
STATS_PARTICLE_PUSHING 100
STATS_PARTS 148, 215
stats_primary_comm() 136, 150, 158, 175, 216, 217
STATS_REB_COMM 99, 148, 207
STATS_REBALANCE 99, 148, 203
stats_reduce_part() 151, 158, 214, 220, 221
Stats_Reduce_Part_Max() 221
Stats_Reduce_Part_Min() 131, 221
Stats_Reduce_Part_Sum() 221
stats_reduce_time() 151, 158, 214, 222, 223
stats_secondary_comm() 138, 150, 158, 198, 216, 217, 218
STATS_TB_COMM 100, 148, 240, 254
STATS_TB_MOVE 99, 148, 245, 248, 413, 418, 423, 527
STATS_TB_SORT 99, 325, 410–412, 416, 442, 492, 532, 535
STATS_TIMINGS . . 99, 101, 102, 147, 215, 218
STATS_TRANSBOUND 99, 100, 147, 172
STATS_TRY_STABLE 99, 148, 177
statsMode 151, 152, 160, 167, 171, 172, 175, 214, 215, 218, 220, 223, 239, 362, 480
StatsPartStrings 148, 157, 222
StatsTimeStrings 100, 148, 157, 222
Statuses 228, 238, 256, 337, 395, 396, 398, 410, 427, 456, 517, 536, 539, 541, 544, 546
stdarg.h (standard header file) 132
stdio.h (standard header file) 132
stdlib.h (standard header file) 119, 132
strcat() 132, 224
string.h (standard header file) 132
subcuboid *84*, 85, 89, *440*, 441, 442, 445–449, 454, 455, 457, 463, 472, 485, 489, 490, 497, 499–502, 505–511, 518–521, 523, 524, 531, 537–541, 543
subdomain code 229, 230, 326, 327, 328, 359, 367, 415, 416, 418, 424, 431, 432, 436, 474, 492

subdomain size *271*
Subdomain_Id() 229, 230, 257–261, 265, 327, 328, 415, 437, 443, 552
SubDomainDesc 272, 282, 284, 288, 289, 293, 295, 296, 310, 315, 317, 318, 360, 435, 478, 550
SubdomainId 134, 159, 162, 175, 209
SubDomains 270, 282, 284, 287, 288, 300, 301, 303, 311, 387, 401, 402, 428, 429, 433, 434, 506, 512, 513, 547, 549
SubDomainsFloat 270, 288, 310, 312
substance . . *162*, 163, 170, 174, 227, 239, 268, 270, 273, 274, 276, 297, 298, 333, 334, 342, 360, 363, 446–448, 453, 454, 459, 472, 475, 478, 479, 481, 485, 488

T

T_Commlist 144, 191, 192, 197, 378, 394, 397, 498
T_Hgramhalf 340, 360, 409, 410, 457, 476, 517
T_Histogram 131, 139, 163, 173, 198
T_n 182, 188
T_Particle 228, 241, 243, 257, 426, 536, 538
T_StatsTime 151, 214, 222
TempArray 139, 163, 166, 167, 185, 187, 190, 195, 207, 242, 243, 256, 342, 356, 360, 405–408, 459, 472, 477, 515
The_Grid() 354, 363, 372, 373, 381, 383, 388, 410, 412, 414, 468, 481, 493–495, 505, 510, 522, 524, 539, 543, 545
totalLocalParticles 227, 235, 237, 239, 333, 363, 453, 481
TotalP 137, 138, 154, 159, 162, 170, 172, 174, 226, 231, 239, 246, 247, 250, 251, 258, 268, 328, 333, 362, 363, 410, 414, 420, 422, 424, 453, 480, 481, 492, 529, 531, 534
totalp *27*, *31*, 36, 44, 45, 47, 48, 50, 57, 64, 72, 76, 88, 108, 110–112, 119, 121–123, 138, 159, 161, 162, 235, 236, 283, 285
totalParts 138, 154, 170, 172, 227, 231, 235, 237, 239, 251, 260, 261, 265–268, 328, 333, 362, 363, 411, 418, 419, 427, 437, 453, 479–481, 492, 528, 546, 548, 550–552
TotalPGlobal 136, 163, 172, 173, 175, 178, 203, 204, 209, 210, 238, 240, 247, 333, 362, 365, 366, 480
TotalPNext 137, 157, 159, 162, 170, 174–176, 184, 197–201, 239, 245–249, 251, 252, 258, 333, 363, 384, 389, 395, 399, 410–413, 419, 420, 422, 453, 481, 491, 492, 501, 502, 507, 509–511, 528, 529, 531
transbound1() 135–139, 147, 151, 152, 154, 155, 162, 167, 168, 170, 171, 172, 175, 176, 179, 184, 203, 215, 238, 239, 268, 270, 287, 311, 350, 362, 364, 466, 480
transbound2() 135, 137, 138, 151, 170, 172, 227, 228, 232, 238, 239, 240, 243, 244, 270, 287, 311, 362, 363
transbound3() 134, 172, 238, 270, 274, 282, 287, 301, 309, 311
transbound4p() 328, 329, 331–333, 337, 346, 350, 353, 358, 362, 364, 366, 367, 427, 431, 436, 446, 480–482
transbound4s() 440, 444, 446–448, 451–453, 456, 462, 466, 468, 474, 480, 482, 483
transitional family tree *324*, 331, 339, 342, 354, 367, 369, 371, 378, 388, 404, 409, 412, 418, 422, 425, 469, 486, 489, 502, 517, 527, 533, 534, 536, 537
TRUE 132, 175, 176, 184, 240, 243, 366, 482, 483
try_primary1() 134, 136–138, 152, 155, 169, 170, 174, 175, 176, 179, 189, 203, 216, 234, 239, 346, 350, 364, 462, 466, 482
try_primary2() . 136–139, 147, 148, 174, 175, 215, 227, 228, 234, 239, 240, 245
try_primary4p() 332, 333, 346, 350, 351, 354, 362, 364, 371, 404, 410–413, 415, 440, 482, 485, 486, 492
try_primary4s() 440, 462, 466, 469, 480, 482, 485, 486, 492, 515
try_stable1() 137, 140, 141, 148, 152, 155, 168, 174, 175, 176, 179, 184, 185, 188, 189, 191, 197, 207, 215, 234, 243, 256, 346, 350, 366, 369, 370, 462, 466, 483, 486
try_stable2() . . 144, 174, 176, 179, 184, 234, 239, 243, 253, 254, 256, 369, 486
try_stable4p() . 346, 350, 362, 366, 369, 483
try_stable4s() 440, 462, 466, 480, 482, 483, 485, 486

U

upd_real_nbr() 342, 347, 405–407, 408, 458, 459, 463, 515, 516
update_descriptors() 334, 347, 351, 369, 378, 401, 403, 454, 463, 467, 486, 512, 514
update_neighbors() . 327, 341, 342, 347, 360, 367, 369, 401, 443, 451, 457–459, 463, 467, 477, 484, 486, 512, 513

update_real_neighbors() 324, 342, 347, 354, 360, 364, 369, 370, 378, 404, 407, 408, 458, 459, 463, 469, 477, 482, 486, 515, 516
update_stats() . 150, 151, 158, 218, 219, 221
URN_PRI 354, 360, 364, 404–406, 469, 477, 482, 515
URN_SEC 354, 369, 404, 469, 486
URN_TRN . . . 354, 370, 378, 404–406, 469, 515

V

va_end() 168, 224
va_start() 168, 224
verbose 30, 33, 44, 57, 62, 64, 77, 90, *105*, 106, 152, 160, 162, 236, 285
verbose messaging 17, 26, 30, 33, 105, 130, 132, 147, 151, 152, 155, 160, 162, 172, 175, 177, 203, 223–225
Verbose() 152, 155, 162, 172, 175, 177, 180, 203, 223, 224
verboseMode 151, 152, 153, 155, 160, 162, 225
vertical exterior halo plan *441*, 445, 446, 450, 464, 490, 519, 523, 525, 537, 539, 541, 542, 544
vertical halo plane . . . *441*, 447, 448, 456, 463, 464, 474, 475, 489, 490, 518, 539
vertical interior halo plane *441*, 445, 446, 450, 464, 489, 490, 518–521, 523, 525, 534, 535, 537, 542, 543
VPlane 448, 449, 450, 475, 491, 518, 520–524, 538, 539, 544
VPlaneHead 448, 449, 450, 475, 491, 518, 519, 537–539, 543, 545
Vprint() 131, 224, 225
vprint() 152, 155, 162, 172, 175, 177, 180, 203, 223, 224
Vprint_Norank() 224, 225
vprintf() 132, 168, 224

X

xfer_boundary_particles_h() 441, 448, 452, 456, 464, 485, 490, 540, 546
xfer_boundary_particles_v() 441, 447, 450–452, 456, 464, 485, 490, 504, 519, 520, 537, 541, 543–545
xfer_particles() . . 323, 332, 333, 337, 338, 342, 348, 371, 425, 441, 451–453, 456, 458, 464, 485, 489, 490, 536

Z

ZBound 447, 448, 475, 480, 481, 488, 491, 502, 507, 508, 519, 520, 523, 539, 543, 545
zbound 84, 447, 469, 473, 475
ZBoundShadow 447, 469, 475, 480, 481

Revision History

v0.9

Gen: The OhHelp library package and this document are born on a hot summer day. (2009/08/21) 1

v0.9.5

Gen: Add "Implementation" section to this document, and fix bugs. (2010/07/21) 1

v0.9.5-01

Gen: The followings are to allow `S_particle` without `spec`. 1

oh_part.h: Introduce constant macro `OH_HAS_SPEC`. 42

`Particle_Spec()`: Define this macro to give $s = 0$ if `S_particle` misses `spec`. 229

`move_injected_to_sendbuf()`: Use `Particle_Spec()` to allow `S_particle` without `spec`. 262

`oh2_inject_particle()`: Use `Particle_Spec()` to allow `S_particle` without `spec`. 265

`oh2_inject_particle()`: Confirm $S = 1$ when `OH_HAS_SPEC` is undefined. 265

v0.9.5-02

oh_mod2.F90: Change mode of `totalp` of `oh2_init()` from `inout` to `out`. 43

oh_mod3.F90: Change mode of `totalp` of `oh3_init()` from `inout` to `out`. 49

v0.9.5-03

Gen: Add "Implementation" section. 130

v0.9.5-04

ohhelp1.h: The followings are for coding style changes. 1

`OH_NEIGHBORS`: `#else#if` is replaced with `#elif`. 133

`OH_NEIGHBORS`: Expressions for 3^D are parenthesized. 133

`RegionId`: Adjust declaration and explanation orders. 134

`NOfPToStay`: Adjust declaration and explanation orders. 137

`RecvCounts`: Adjust declaration and explanation orders. 138

`NOfSend`: Adjust declaration and explanation orders. 138

`MyComm`: Adjust declaration and explanation orders. 145

ohhelp1.h: Declare dimension independent C API prototypes at first. 153

ohhelp1.h: Declare dimension dependent C API prototpyes at second. 154

ohhelp1.h: Declare Fortran API prototpyes at last. 154

ohhelp1.h: Adjust prototype declaration and function definition orders. 155

`Verbose()`: Remove redundant condition `L==0 && verboseMode`. 155

v0.9.5-05

ohhelp1.h: The followings are for bug fixes. 1

`S_statscurr`: `time.val` should have $2K_t + 2$ elements rather than $2K_t + 1$. 149

`S_statscurr`: `time.ev` should have $2K_t + 2$ elements rather than $2K_t + 1$. 150

v0.9.5-06

ohhelp1.c: Functions are defined in top-down order. 157

v0.9.5-07

ohhelp1.c: The followings are for bug fixes. 1

`init1()`: Third arg of `local_errstop` is $(3^D - 1) - i$ rather than $3^D - i$. 166

`try_primary1()`: Setting `RegionId[1]` and `SubdomainId[1]` to -1 should be done regardless of `level`. 175

v0.9.5-08

ohhelp2.h: The followings are for coding style changes. 1

ohhelp2.h: Declare level independent C API prototypes at first. 231

ohhelp2.h: Declare level dependent C API prototypes at second. 232

ohhelp2.h: Declare Fortran API prototypes at last. 232

ohhelp2.h: Adjust prototype declaration and function definition orders. 233

v0.9.5-09

ohhelp_c.h: Define aliase and then declare prototypes. 232

ohhelp_c.h: `#else#if` for `OH_DIMENSION` is replaced with `#elif`. 279

ohhelp_c.h: Remove new-lines in the alias definition of `oh_init()` with `oh3_init()`. . . . 279

v0.9.5-10
ohhelp2.c: The followings are for coding style changes. 1
ohhelp2.c: Functions are defined in top-down order. 234
exchange_particles(): Remove unnecessary rlsize[]. 257
v0.9.5-11
ohhelp2.c: The followings are for enhancement of the handling of PrimaryInjection[][]. . . 1
move_injected_from_sendbuf(): Add argument pinj. 234
move_to_sendbuf_primary(): Remove the copy from PrimaryInjection[0][] to PrimaryInjection[1][]. 245
move_to_sendbuf_primary(): Add argument PrimaryInjection[0][] to the call of move_injected_from_sendbuf(). 247
move_to_sendbuf_secondary(): Add argument PrimaryInjection[1][] to the call of move_injected_from_sendbuf(). 252
move_injected_from_sendbuf(): Add argument pinj instead of referring to PrimaryInjection[1][] even in primary mode. 262
v0.9.5-12
ohhelp2.c: The followings are for bug fixes. 1
set_sendbuf_disps(): The condition for disp+=PrimaryInjection[s] should be $m = n$ instead of $mS = n$ for node m. 253
v0.9.5-13
ohhelp3.h: The followings are for coding style changes. 1
ohhelp3.h: Adjust the order of variables/structures declarations and explanations. 270
ohhelp3.h: Declare C API prototypes and then Fortran API. 278
ohhelp3.h: #else#if for OH_DIMENSION is replaced with #elif. 279
v0.9.5-14
ohhelp_f.h: #else#if for OH_DIMENSION is replaced with #elif. 279
ohhelp_f.h: Remove new-lines in the alias definition of oh_init() with oh3_init(). . . . 279
v0.9.5-15
ohhelp3.c: The followings are for coding style changes. 1
ohhelp3.c: Adjust prototype declaration and function definition orders. 283
init_subdomain_actively(): Clarify the initialization of Grid[d] for $d \geq D$. 290
init_subdomain_passively(): Use lo and up instead of equivalent sd[][][β]. 293
init_subdomain_passively(): Clarify the initialization of Grid[d] for $d \geq D$. 294
Field_Disp(): #else#if for OH_DIMENSION is replaced with #elif. 297
init_fields(): Allocation of BorderExc is delayed until it is really necessary. 301
oh3_map_particle_to_neighbor(): #else#if for OH_DIMENSION is replaced with #elif. 313
oh3_map_particle_to_subdomain(): #else#if for OH_DIMENSION is replaced with #elif. 315
v0.9.5-16
ohhelp3.c: The followings are for bug fixes. 1
init_fields(): Remove redundant argument bd[][]. 283
oh3_init_(): Add specBase=1. 286
oh3_init(): Add specBase=0. 286
init3(): Remove redundant argument bd[][] of init_fields(). 289
init_subdomain_actively(): bc[][] should be in $[b, B+b)$ rather than $[0, B)$. 290
init_subdomain_actively(): System lower boundary should be Δ_d^l rather than 0. . . . 292
init_subdomain_actively(): bc[d][β] can be accessed if $d < D$. 292
init_subdomain_passively(): Elements $[D-1]$ of lo, up and h are unnecessary. 293
init_subdomain_passively(): Grid[i].coord[OH_UPPER] must be Grid[d].coord[OH_UPPER]. 294
init_subdomain_passively(): Grid[d].{n, light.n, light.thresh} are set to be 0 rather than left undefined. 294
init_subdomain_passively(): Texts "rank-n's" in the message argument in two calls of local_errstop() are replaced with "rank-n". 294

`init_subdomain_passively()`: Initial values of `h[d]`, `lo[d]`, `up[d]` are those corresponding to `sdd[0]` rather than special values meaning undefined. This makes search process start from $m = 1$ rather than $m = 0$. 296
`init_subdomain_passively()`: Set `sdd`$[m]$`.coord`$[D-1]$`.{n,h}` $= \{1, m\}$ rather than left undefined. 296
`init_subdomain_passively()`: Set `sdd`$[h_d]$`.coord`$[d]$`.n` $= N - h_d$ rather than left undefined. 296
`init_fields()`: Remove redundant argument `bd[][]`. 297
`init_fields()`: c and f in the message given to `errstop` are offset by `cfid`. 299
`init_fields()`: Initializing `bx`[][0][][] is entrusted to `set_border_exchange()` while that for `bx`[][1][][] is done by `clear_border_exchange()` except for `deriv` elements. 301
`set_field_descriptors()`: $\sigma(f, t, m)$ should be the number of elements from the base to the upper corner element inclusively, rather than to the imaginary element just outside of the upper boundaries. 301
`set_border_comm()`: `wdh[OH_DIM_Y]` should be accessed if $D > 1$. 304
`clear_border_exchange()`: Elements `buf`, `deriv` and `type` are cleared. 309
`Neighbor_Id()`: Renamed from `NeighborId()` to obey the naming convention. 312
`Neighbor_Id()`: In the RHS of the second condition, we have to refer to `n` rather than `(N)` to avoid multiple reference of `Neighbors[ps][n]` since `n` is overwritten by this macro. 312
`oh3_map_particle_to_neighbor()`: Arguments are renamed to have `x`, `y` and `z` according to the API definition, while local variables are renamed to `xx`, `yy` and `zz`. 313
`Map_Particle_To_Subdomain()`: Since `Grid`$[d]$`.light.thresh` has the absolute system coordinate value, it should be compared with law `XYZ`. 314
`oh3_map_particle_to_subdomain()`: Arguments are renamed to have `x`, `y` and `z` according to the API definition, while local variables are renamed to `xx`, `yy` and `zz`. 315
`map_irregular()`: The first argument for the first and second calls of `map_irregular_range()` should be $2x - \delta_d^{\max}$ and $2x + \delta_d^{\max}$ rather than their reversal. 318
`map_irregular()`: We have to skip subdomains in a wall/pillar rather than scanning all of them. 318
`map_irregular_range()`: We have to search $\min\{m \mid \delta_d^l(i(m)) + \delta_d^u(i(m)) > x'\}$ rather than max of the set. 318
v0.9.6
Gen: Introduce function `oh2_set_total_particles()` (so far). (2010/10/02) 1
v0.9.6-01
Gen: The followings are for `oh2_set_total_particles()`. 1
Gen: Add description of `oh2_set_total_particles()`. 41
Gen: Add description of `oh2_set_total_particles()`. 47
oh_mod2.F90: Introduce function `oh2_set_total_particles()`. 47
Gen: Add description of `oh2_set_total_particles()`. 97
Gen: Add aliase `oh_set_total_particles()`. 107
`TotalP`: Initialized by `set_total_particles()`. 137
`primaryParts`: Initialized by `set_total_particles()`. 138
`totalParts`: Initialized by `set_total_particles()`. 138
ohhelp1.h: Add prototype of `set_total_particles()`. 154
`set_total_particles()`: Introduce function `set_total_particles()`. 170
`transbound1()`: Use `set_total_particles()` to initialize `TotalP`, etc. 172
ohhelp2.h: Add prototype of `oh2_set_total_particles()`. 231
ohhelp_c.h: Add alias `oh_set_total_particles()` and prototype of `oh2_set_total_particles()`. 231
ohhelp_f.h: Add alias `oh_set_total_particles()`. 231
ohhelp2.h: Add prototype of `oh2_set_total_particles_()`. 232
`oh2_set_total_particles()`: Introduce the function `oh2_set_total_particles()`. . . . 268

v0.9.6-02
Gen: The followings are for moving currMode from level-3 to level-1 due to the introduction of oh2_set_total_particles(). 1
currMode: currMode is moved from ohhelp3.h to ohhelp1.h. 134
init1(): Move initialization of currMode from init3(). 162
local_errstop(): Add transbound1() as a caller. 168
transbound1(): Add check of currMode = currmode. 172
transbound1(): currMode is set to the return value. 174
transbound2(): currMode is set to the return value. 239
ohhelp3.h: currMode is moved to ohhelp1.h. 270
init3(): Initialization of currMode is moved to init1(). 287
transbound3(): Setting currMode is moved to transbound1() and transbound2(). 311
v0.9.7
Gen: Intorduce function oh3_grid_size(), modify the method of mapping particle to subsomain, etc. (2011/03/26) 1
v0.9.7-01
Gen: The followings are for oh3_grid_size(). 1
Gen: Add description of oh3_grid_size(). 48
Gen: Add description of oh3_grid_size(). 64
oh_mod3.F90: Introduce function oh3_grid_size(). 64
Gen: Add aliase oh_grid_size(). 107
ohhelp3.h: Add prototype of oh3_grid_size(). 277
ohhelp_c.h: Add alias oh_grid_size() and prototype of oh3_grid_size(). 278
ohhelp_f.h: Add alias oh_grid_size(). 278
oh3_grid_size(): Introduce function oh3_grid_size(). 310
v0.9.7-02
Gen: The followings are for particle/subdomain mapping. 1
SubDomainsFloat: SubDomainsFloat[][][] is introduced as the floating-point and grid-size-aware counterpart of SubDomains[][][]. 270
S_grid: Add elements fcoord[], fsize, gsize, light.rfsize, light.rfsizeplus and light.fthresh for particle/subdomain mapping using floating-point coordinate. . . 271
S_subdomdesc: Add element coord[].fc[] for particle/subdomain mapping using floating-point coordinate. 272
map_irregular_subdomain(): Change coordinate argument type to be double from int. 280
map_irregular(): Change coordinate argument type to be double from int. 282
map_irregular_range(): Change the type of argument p to be double from int. 282
init3(): Add local variable sdf for SubDomainsFloat. 287
init3(): Add allocation and initialization of SubDomainsFloat. 288
init_subdomain_actively(): Add initializations of Grid[d]'s elements, fcoord[], fsize, gsize, light.rfsize, light.rfsizeplus and light.fthresh. 290
init_subdomain_passively(): Add initialization of SubdomainDesc[m].coord[d].fc[β]. 293
init_subdomain_passively(): Add initializations of Grid[d]'s elements, fcoord[], fsize, gsize, light.rfsize, light.rfsizeplus and light.fthresh. 294
Map_Particle_To_Neighbor(): Remove argument of integer particle coordinate and modify the code so that it refers to grid-size-aware subdomain coordinates in SubDomainsFloat[][][] and Grid[]'s elements. 312
oh3_map_particle_to_neighbor(): Remove the conversion from floating-point particle coordinate to integer one and the corresponding argument of Map_Particle_To_Neighbor(). 313
Map_Particle_To_Subdomain(): Modify calculations so that they refer to grid-size-aware elements of Grid[]. 314
Adjust_Subdomain(): Introduce this macro to correct floating-point calculation errors in Map_Particle_To_Subdomain(). 314

oh3_map_particle_to_subdomain(): Modify arguments of Map_Particle_To_Subdomain() and add invocations of Adjust_Subdomain(). 315
map_irregular_subdomain(): Change argument type to be double from int. 317
map_irregular(): Change coordinate argument type to be double from int and modify codes to refer to grid-size-aware elements in SubDomainDesc[] and Grid[]. 318
map_irregular_range(): Change the type of p to be double from int and modify codes to refer to grid-size-aware elements in SubDomainDesc[].coord[].fc[]. 318
v0.9.7-03
Gen: The followings are for miscellaneous revision. 1
init1(): The return value of mem_alloc() is explicitly cast as int* for *sdid, *nphgram and *totalp. 162
init2(): The return value of mem_alloc() is explicitly cast as int* for *pbase. 237
v0.9.8
Gen: The followings are for fixing the bug in the secondary mode stability check with injected particles. (2011/04/29) 1
NOfPToStay: Modify explanation stating that injected particles are exculded from NOfPToStay. 137
InjectedParticles: Moved from ohhelp2.h so that count_stay() refers to it. 138
count_real_stay(): Introduced to sum up $q(m)[p][s][n]$ for all s for the adjustmet of get.prime and get.sec. 157
init1(): Allocation and initialization of PrimaryInjection is moved from init2(). . . . 163
try_stable1(): Add explanation that injected particles are excluded from stay.prime, stay.sec and NOfPToStay[]. 177
try_stable1(): Add explanation that injected particles are excluded from stay.prime, stay.sec and NOfPToStay[]. 180
count_stay(): PrimaryInjection$[0][s]$ is subtracted from NOfPLocal$[0][s][n]$ of n to exclude injected particles from those staying. 185
schedule_particle_exchange(): Add the adjustment of get.prime and get.sec when reb ≤ 0 using the newly introduced count_real_stay(), which is also used for the adjustment of get.sec in the case of reb > 0. 191
schedule_particle_exchange(): Set reb to 3 if it is -1 after the calls of sched_comm() instead of before them. 191
count_real_stay(): Introduced to sum up $q(m)[p][s][n]$ for all s for the adjustmet of get.prime and get.sec. 193
sched_comm(): Use NodesNext[] if reb > 0 instead of reb $\neq 0$. 193
sched_comm(): Scan neighbors if reb is 0, 1 or 2 excluding -1. 195
rebalance1(): Modify the calculation of get.prime so that it is always based on NOfPLocal$[0][s][n]$ instead of relying stay.prime from which we excluded injected particles. 206
ohhelp2.h: PrimaryInjection is moved to ohhelp1.h. 228
init2(): Allocation and initialization of PrimaryInjection is moved to init1(). 238
v0.9.9
Gen: Modifications for position-aware particle management. (2011/05/11) 1
v0.9.9-01
Gen: The followings are to add cases 2 and 3 to the domain of currMode $\{-1, 0, 1\}$ to mean forced anywhere accommodation mode in primary and secondary mode, and to define/use macros for checking and setting of currMode and its function-local version. 1
currMode: Add 2 and 3 to the domain of currMode. 134
MODE_NORM_PRI: Introduce this macro to spcify primary mode and normal accommodation. 134
MODE_NORM_SEC: Introduce this macro to spcify secondary mode and normal accommodation. 134
MODE_REB_SEC: Introduce this macro to spcify rebalanced secondary mode. 134

MODE_ANY_PRI: Introduce this macro to spcify primary mode and anywhere accommodation. . . . 135
MODE_ANY_SEC: Introduce this macro to spcify secondary mode and anywhere accommodation. . . . 135
Mode_PS(): Introduce this macro to examine primary/secondary mode indicator. . . . 135
Mode_Acc(): Introduce this macro to examine normal/anywhere accommodation indicator. . . . 135
Mode_Set_Pri(): Introduce this macro to set mode indicator to primary. . . . 135
Mode_Set_Sec(): Introduce this macro to set mode indicator to secondary. . . . 135
Mode_Set_Norm(): Introduce this macro to set accommodation indicator to normal. . . . 135
Mode_Set_Any(): Introduce this macro to set accommodation indicator to anywhere. . . 135
Mode_Is_Norm(): Introduce this macro to test normal accommodation. . . . 135
Mode_Is_Any(): Introduce this macro to test anywhere accommodation. . . . 135
init1(): Use MODE_NORM_PRI to initialize currMode. . . . 162
set_total_particles(): Add extraction of mode indicator of currMode with Mode_PS(). 170
transbound1(): Use MODE_NORM_SEC as the default return value. . . . 172
transbound1(): Use Mode_PS() to examine the mode indicator of currmode and currMode. . . . 172
transbound1(): TotalPGlobal[N] is set to 1 if Mode_Is_Norm(currMode) is false to indicate forced anywhere accommodation. . . . 172
transbound1(): Use Mode_Set_Any() to turn on accommodation indicator. . . . 173
transbound1(): Use MODE_NORM_PRI and MODE_REB_SEC for return value, and Mode_PS() to examine mode indicator. . . . 174
try_stable1(): Use MODE_NORM_SEC to check if currmode indicates normal accommodation. . . . 184
make_comm_count(): Use MODE_NORM_PRI, MODE_REB_SEC and Mode_ANY_SEC for the examination of currmode. . . . 197
make_comm_count(): Use Mode_Is_Any() to check if anywhere accommodation. . . . 198
rebalance1(): Use Mode_PS() to examine the mode indicator of currmode. . . . 203
rebalance1(): Use MODE_NORM_PRI and MODE_NORM_SEC to examine currmode. . . . 206
build_new_comm(): Use Mode_PS() to examine currmode. . . . 206
build_new_comm(): Use Mode_Is_Norm() to check if currmode indicates normal accommodation. . . . 209
stats_primary_comm(): Use MODE_ANY_PRI and Mode_PS() to examine what currmode indicates. . . . 216
stats_secondary_comm(): Use Mode_PS() to examine the mode indicator of currmode. . 216
update_stats(): Use Mode_PS() to examine the mode indicator of currmode. . . . 220
transbound2(): Use MODE_NORM_SEC for the default return value. . . . 239
transbound2(): Use MODE_NORM_PRI and MODE_REB_SEC for return value, and Mode_PS() to examine mode indicator. . . . 239
try_primary2(): Use Mode_PS() to examine mode indicator. . . . 240
exchange_primary_particles: Use MODE_NORM_PRI() to examine if currmode indicates primary mode and normal accommodation. . . . 241
try_stable2(): Use MODE_NORM_SEC to examine if currmode indicates secondary mode and normal accommodation. . . . 243
rebalance2(): Use Mode_PS() to examine the mode indicator of currmode, and Mode_Is_Norm() to examine if it indicates normal accommodation. . . . 244
exchange_particles(): Use Mode_PS() to extract the mode indicator of currmode. . . . 254
oh3_exchange_borders(): Use Mode_PS() to extract the mode indicator of currMode. . . 321
v0.9.9-02
Gen: The followings are to split exchange_primary_particles() from try_primary2(). . . 1
ohhelp2.h: Introduce function exchange_primary_particles(). . . . 233
try_primary2(): The core part to transfer particles is split out to the new function exchange_primary_particles(). . . . 240

exchange_primary_particles(): Introduce this function for the core part to transfer particles originally done by try_primary2(). 240
v0.9.9-03
Gen: The followings are to make nid of S_particle structure can have particle position in addition to subdomain identifier. 1
Decl_Grid_Info(): Introduced to declare subdomid, gridmask and loggrid to extract subdomain identifier and grid position from nid of S_particle structure if OH_POS_AWARE is defined. 229
Subdomain_Id(): Introduced to extract subdomain identifier from nid of S_particle structure if OH_POS_AWARE is defined. 229
move_to_sendbuf_uw(): Add the first argument ps to obtain the identifier of a subdomain neighboring to the local node's primary or secondary subdomain by Sudomain_Id(). 234
move_to_sendbuf_dw(): Add the first argument ps to obtain the identifier of a subdomain neighboring to the local node's primary or secondary subdomain by Sudomain_Id(). 234
move_to_sendbuf_primary(): Add the first argument ps = 0 of move_to_sendbuf_uw() to let it obtain the identifier of a subdomain neighboring to the local node's primary subdomain by Sudomain_Id(). 246
move_to_sendbuf_primary(): Add the first argument ps = 1 of move_to_sendbuf_uw() to let it obtain the identifier of a subdomain neighboring to the local node's secondary subdomain by Sudomain_Id(). 247
move_to_sendbuf_primary(): Add the first argument ps = 0 of move_to_sendbuf_dw() to let it obtain the identifier of a subdomain neighboring to the local node's primary subdomain by Sudomain_Id(). 247
move_to_sendbuf_secondary(): Add the first argument ps = 0 of move_to_sendbuf_uw() to let it obtain the identifier of a subdomain neighboring to the local node's primary subdomain by Sudomain_Id(). 251
move_to_sendbuf_secondary(): Add the first argument ps = 1 of move_to_sendbuf_uw() and move_to_sendbuf_dw() to let them obtain the identifier of a subdomain neighboring to the local node's secondary subdomain by Sudomain_Id(). 251
move_to_sendbuf_secondary(): Add the first argument ps = 0 of move_to_sendbuf_dw() to let it obtain the identifier of a subdomain neighboring to the local node's primary subdomain by Sudomain_Id(). 252
move_to_sendbuf_uw(): Add the first argument ps to obtain the identifier of a subdomain neighboring to the local node's primary or secondary subdomain by Sudomain_Id(). 258
move_to_sendbuf_uw(): Macro Decl_For_Subdomain_Id() is added to use Subdomain_Id(). 258
move_to_sendbuf_dw(): Reference to Particle[].nid is replaced with Subdomain_Id(Particle[].nid,ps) so that its part of subdomain identifier is extracted if OH_POS_AWARE is defined. 259
move_to_sendbuf_dw(): Add the first argument ps to obtain the identifier of a subdomain neighboring to the local node's primary or secondary subdomain by Sudomain_Id(). 260
move_to_sendbuf_dw(): Macro Decl_For_Subdomain_Id() is added to use Subdomain_Id(). 260
move_to_sendbuf_dw(): Reference to Particle[].nid is replaced with Subdomain_Id(Particle[].nid,ps) so that its part of subdomain identifier is extracted if OH_POS_AWARE is defined. 261
move_injected_to_sendbuf(): Reference to Particle[].nid is replaced with Subdomain_Id(Particle[].nid),0, for which Decl_For_Subdomain_Id() is added, so that its part of subdomain identifier is extracted if OH_POS_AWARE is defined. 262
v0.9.9-04
Gen: The followings are to avoid anywhere accommodation in case of particle injection into secondary subdomain. 1
InjectedParticles: Renamed from PrimaryInjection and doubled to make it possible to inject secondary particles. 138

`init1()`: `PrimaryInjection` is renamed as `InjectedParticles` and its size is doubled for secondary particle injection. 163
`set_sendbuf_disps()`: Add second argument `parent` to indicate whether it should take care the particles injected into secondary subdomain. 233
`move_injected_from_sendbuf()`: First argument is renamed as `injected` and two more arguments are added for secondary particle injection. 234
`transbound2()`: `PrimaryInjection` is renamed as `InjectedParticles` and its size is doubled for secondary particle injection. 239
`rebalance2()`: `InjectedParticles`[0][1][S] are cleared if old and new parents are different. 244
`move_to_sendbuf_primary()`: Add second argument of `set_sendbuf_disps()` to tell it the local node will not have parent in the next simulation step. 246
`move_to_sendbuf_primary()`: Add second argument of `set_sendbuf_disps()` to tell it the local node will not have parent in the next simulation step. 247
`move_to_sendbuf_primary()`: Add second and third arguments of `move_injected_from_sendbuf()` to indicate that injected particles are primary. . . . 247
`move_to_sendbuf_secondary()`: Handling injected particles are moved into this loop because we now have seconday particle injection. 249
`move_to_sendbuf_secondary()`: Add second argument of `set_sendbuf_disps()` to tell it to take care injected secondary particles if the local node will have parent in the next simulation step. 250
`move_to_sendbuf_secondary()`: Add second argument of `set_sendbuf_disps()` to tell it to take care injected secondary particles if the local node will have parent in the next simulation step. 252
`move_to_sendbuf_secondary()`: `move_injected_from_sendbuf()` is called twice, one for primary injection and the other for secondary. 252
`set_sendbuf_disps()`: The argument `parent` is added to handle secondary particle injection so that the space for injected particles in `SendBuf`[] is kept. 253
`move_injected_to_sendbuf()`: Modify the comment, which had only mentioned particles injected into primary subdomain, so as to clarify that we now take care of those injected into secondary subdomains. 262
`move_injected_from_sendbuf()`: Add arguments `mysd` to have n or $parent(n)$ and `rbb` pointing `RecvBufBases`[p][], for secondary particle injection. 262
`oh2_inject_particle()`: Add the logic to regard a particle injected into secondary subdomain as a secondary particle, with the added second dimension of `InjectedParticles`[2][2][S]. 265

v0.9.9-05

Gen: The followings are for renaming oh_dim.h as oh_config.h so that it has cofiguration definitions not only for the dimension. 1
Gen: Change title of the section to reflect the file name change from oh_dim.h to oh_config.h. 25
oh_config.h: Move the definition of `OH_LIB_LEVEL` from ohhelp_f.h and ohhelp_c.h to oh_config.h. 25
oh_mod1.F90: Include oh_config.h rather than oh_dim.h. 26
oh_mod2.F90: Include oh_config.h rather than oh_dim.h. 41
oh_mod3.F90: Include oh_config.h rather than oh_dim.h. 48
Gen: Modify the explanation of the definition `OH_LIB_LEVEL` which is now defined in oh_config.h. 106
Gen: Change reference to oh_dim.h to `oh_config.h`. 129
Gen: Change reference to oh_dim.h to `oh_config.h`. 129
Gen: Rename oh_dim.h as oh_config.h. 130
ohhelp1.h: Rename oh_dim.h as oh_config.h. 132
ohhelp_c.h: Move the definition of `OH_LIB_LEVEL` to oh_config.h. 153
ohhelp_f.h: Move the definition of `OH_LIB_LEVEL` to oh_config.h. 153

v0.9.9-06

Gen: The followings are for changes in lower level libraries for position-aware particle exchange scheduling. 1
OH_LIB_LEVEL_4P: Add commented-out definition of the macro to oh_config.h for the activation of level-4p extention. 25
oh_config.h: Define OH_LIB_LEVEL as 4 if OH_LIB_LEVEL_4P is defined. 25
OH_BIG_SPACE: Add commented-out definition of the macro to oh_config.h for using 64-bit nid element in S_particle. 25
OH_NO_CHECK: Add commented-out definition of the macro to oh_config.h for omitting the argument consistency check in API functions for particle mapping, injection and removal. 25
oh_part.h: Add typedef of OH_nid_t to switch 32-bit/64-bit representation of nid depending on OH_BIG_SPACE. 42
oh_type.F90: Add switch of 32-bit/64-bit representation of nid depending on OH_BIG_SPACE. 42
STATS_TB_SORT: Add the definition of this macro to oh_stats.h for the measurement of the time of particle sorting. 99
StatsTimeStrings: Add the strings corresponding to STATS_TB_SORT. 100
Gen: Make aliases of oh2_max_local_particles() and oh2_inject_particle() level-2/3 specific, and make those of oh3_init(), oh3_transbound(), oh3_map_particle_to_neighbor() and oh3_map_particle_to_subdomain() level-3 specific. 107
OH_POS_AWARE: Add the definition of this macro if OH_LIB_LEVEL_4P is defined to mean position-aware particle management is in effect. 132
TempArray: Add comment that it can be allocated in level-4p (or higher) initializer. . . . 139
Neighbors: Add the element [2] to have neighbors of the new parent of the local node after rebalancing, to keep those fo the old parent in [1]. 147
build_new_comm(): Split this function from rebalance1() to insert position-aware particle scheduling before this function works. 155
init1(): Surround allocation of TempArray with #ifndef OH_POS_AWARE and #endif to let init4p() do that. 163
init1(): Take care the possiblity that the size of CommList required by level-1 library can be less than that required by level-4p library. 164
try_primary(): Add messages to show execution and accommodation modes to the first Verbose() while the old execution mode is removed from the second one. 175
Special_Pexc_Sched(): Introduce this macro to examine if particle management is position-aware in try_stable1() to skip particle exchange scheduling. 176
try_stable1(): Add the examination of position-aware particle management by Special_Pexc_Sched() to skip particle exchange scheduling if it is true. 184
schedule_particle_exchange(): Add setting the return value of count_real_stay() into get.sec for the reference in make_recv_list(). 191
rebalance1(): Add setting the return value of count_real_stay() into get.sec for the reference in make_recv_list(). 206
rebalance1(): Add the examination of position-aware particle management by Special_Pexc_Sched() to skip the call of schedule_particle_exchange() if it is true. 206
rebalance1(): Split the second half after the call of schedule_particle_exchange() as build_new_comm() so that position-aware particle exchange scheduling can be inserted before the new communicators are created. 206
rebalance1(): Add the argument nbridx to specify the element of Neighbors[] to receive that from the parenet. 206
build_new_comm(): This functionn is split from rebalance1() so that position-aware particle exchange scheduling can be inserted before the old second half after the call of schedule_particle_exchange(). 206

build_new_comm(): Modify the element of Neighbors[] so that its parent element [0] may be received in [2] if specified by the argument nbridx if we have to keep [1] for the old parent. 209
build_new_comm(): Add the examination of position-aware particle management by Special_Pexc_Sched() to skip the call of make_comm_count() if it is true. 209
SendBuf: Add comment that it can be allocated in level-4p (or higher) initializer. 227
RecvBufBases: Add a footnote comment that it has one extra element to point the tail of $rbuf(1, S-1)$ for sort_received_particles(). 227
Particle_Spec(): Move this macro from ohhelp2.c so that it is accessible from higher level library sources. 229
Primarize_Id(): Introduced to let the subdomain-part of nid have the identifier m if it has $N + m$ to indicate the particle is injected into the local node's secondary subdomain or its neighbor. 230
ohhelp_c.h: Make aliases of oh2_max_local_particles() and oh_inject_particle() level-2/3 specific. 231
ohhelp_f.h: Make aliases of oh2_max_local_particles() and oh_inject_particle() level-2/3 specific. 231
ohhelp2.h: Make prototype of level-2/3 specific oh2_max_local_particles() follow level independt oh2_set_total_particles(). 232
ohhelp2.h: Make prototype of level-2/3 specific oh2_max_local_particles_() follow level independt oh2_set_total_particles_(). 232
ohhelp2.h: Move prototypes of move_to_sendbuf_primary(), set_sendbuf_disps() and exchange_particles() from ohhelp2.c so that they are called from level-4p library. 233
ohhelp2.c: Move prototypes of move_to_sendbuf_primary(), set_sendbuf_disps() and exchange_particles() to ohhelp2.h so that they are called from level-4p library. . . 235
init2(): Initialize totalParts $= P_{lim}$ so that level-4p functions such as oh4p_map_particle_to_neighbor() judge their argument particle is not injected one. 237
init2(): Surround allocation of SendBuf with #ifndef OH_POS_AWARE and #endif to let init4p() do that. 238
init2(): Let RecvBufBases have one extra element to point the tail of $rbuf(1, S-1)$ for sort_received_particles(). 238
init2(): Let Requests and Statuses have $4NS + 2 \cdot 3^D$ elements to cope with the case in which we could have $4N$ requests for hot-spot gathering and $2 \cdot 3^D$ requests for hot-spot scattering. 238
move_to_sendbuf_primary(): Make it accessible from higher level liberary sources. 245
set_sendbuf_disps(): Make it accessible from higher level liberary sources. 253
exchange_particles(): Make it accessible from higher level liberary sources. 254
move_injected_to_sendbuf(): Add the check of nid of each injected particle to cope with the possibility that the particle is out-of-bounds. 262
move_injected_to_sendbuf(): Add the invocatoin of Primarize_Id() if OH_POS_AWARE is defined and the subdomain part of nid of the injected particle has $N + m$ to indicate it is a secondary particle injected into the secondry subdomain of the local node. 262
oh2_inject_particle(): Add the condition $m \geq 0$ for incrementing NOfPLocal[0][s][m] and InjectedParticles[0][s] to make it possible to inject a particle silently and then determine its location afterward by oh4p_map_injected_prticle(). 265
S_grid: Add element rgsize to have $1/\gamma_d$. 271
ohhelp_c.h: Move aliases definitions and prototype declarations of level-independent level-3 functions to the top. 278
ohhelp_f.h: Move aliases definitions and prototype declarations of level-independent level-3 functions to the top. 278
ohhelp3.h: Make oh3_map_particle_to_neighbor(), oh3_map_particle_to_subdomain(), oh3_init() and oh3_transbound() level-3 specific. 278

ohhelp3.h: Move prototypes of `init3()`, `set_field_descriptors()`, `clear_border_exchange()`, and `map_irregular_subdomain()` from ohhelp3.c so that they are called from level-4p library. . . . 280
ohhelp3.c: Move prototypes of `init3()`, `set_field_descriptors()`, `clear_border_exchange()`, and `map_irregular_subdomain()` to ohhelp3.h so that they are called from level-4p library. . . . 283
`set_border_exchange()`: Add an argument `type` for the base type of the boundary communication which can be `MPI_LONG_LONG_INT`. . . . 283
`set_border_comm()`: Add an argument `basetype` for the base type of the boundary communication which can be `MPI_LONG_LONG_INT`. . . . 283
`init3()`: Make the function not `static` so that it is called from level-4p library. . . . 287
`init_subdomain_actively()`: Add initializations of `Grid`[d]`.rgsize`. . . . 290
`init_subdomain_passively()`: Add initializations of `Grid`[d]`.rgsize`. . . . 294
`init_fields()`: Add `#indef`/`#else`/`#endif` construct to make the terminator of `cf`[] independent of `cfid` if `OH_POS_AWARE` is defined. . . . 298
`init_fields()`: Surround allocation and initialization of `FieldTypes`[][], `BoundaryCommTypes`[][][] and `BoundaryCommFields`[] with `#ifndef OH_POS_AWARE` and `#endif` to let `init4p()` do that. . . . 298
`init_fields()`: Add the special care of the last element `BorderExc`[$C-1$][][][] for which the `type` argument of `set_border_exchange()` is `MPI_LONG_LONG_INT` rather than `MPI_DOUBLE` for the boundary communication of per-grid histogram, if `OH_POS_AWARE` is defined. . 301
`set_field_descriptors()`: Make the function not `static` so that it is called from level-4p library. . . . 301
`set_border_exchange()`: Add an argument `type` for the base type of the boundary communication which can be `MPI_LONG_LONG_INT`. . . . 302
`set_border_exchange()`: Add an argument `basetype` of `set_border_comm()` for the base type of the boundary communication which can be `MPI_LONG_LONG_INT`. . . . 304
`set_border_comm()`: Add an argument `basetype` for the base type of the boundary communication which can be `MPI_LONG_LONG_INT`. . . . 304
`set_border_comm()`: The `bcx->type` is set to the `basetype` argument rather than `MPI_DOUBLE`. . . . 306
`set_border_comm()`: The `bcx/bcy->type` is set to the `basetype` argument or that derived from it rather than `MPI_DOUBLE` or its derivatives. . . . 306
`set_border_comm()`: The `bcx/bcy/bcz->type` is set to the `basetype` argument or that derived from it rather than `MPI_DOUBLE` or its derivatives. . . . 308
`clear_border_exchange()`: Make the function not `static` so that it is called from level-4p library. . . . 309
`oh3_grid_size()`: Add update of `Grid`[d]`.rgsize`. . . . 310
`map_irregular_subdomain()`: Make the function not `static` so that it is called from level-4p library. . . . 317
`oh3_exchange_borders()`: Add the argument `type` of `set_border_exchange()` being `MPI_DOUBLE` unconditionally. . . . 321
samplef.mk: Rename oh_dim.h as oh_config.h. . . . 554
samplec.mk: Rename oh_dim.h as oh_config.h. . . . 555
v0.9.9-07
Gen: The followings are for bug fixes. . . . 1
`init_subdomain_actively()`: Add a space after the first half of `errstop()`'s message. . 290
`init_subdomain_passively()`: Add a space after the first portion of `local_errstop()`'s message. . . . 294
v0.9.9-08
Gen: The followings are for newly designed ohhelp4p.h and ohhelp4p.c. . . . 1
Gen: Add a sentence for the level-4p extension. . . . 1
Gen: Add introductory description of level-4p extension. . . . 17
Gen: Modify the description to include level-4p. . . . 20

Gen: Modify the description about nphgram exculded from level-4p API. 23
Gen: Modify the description of position-awareness. 24
oh_part.h: Add description of oh4p_inject_particle(). 42
oh_part.h: Add description of the necessity of x, y, z elements in S_particle. 42
Gen: Add the section to describe level-4p extension. 72
Gen: Add introduction of oh4p_inject_particle(). 96
Gen: Split S3.9 into three sub-sub-sections. 96
Gen: Add description of level-4p injection and removal. 97
Gen: Add aliases oh_per_grid_histogram() and oh_remove_mapped_particle(), and those of oh4p_init(), oh4p_max_local_particles(), oh4p_transbound(), oh4p_map_particle_to_neighbor(), oh4p_map_particle_to_subdomain() and oh4p_inject_particle(). 107
Gen: Add level-4p library files 129
Gen: Add level-4p library files. 129
Gen: Add description of level-4s. 130
Gen: Add the section for the overview of level-4p extension. 323
ohhelp4p.h: Add this header file with the section for it. 325
ohhelp4p.c: Add this source file with the section for it. 346
samplef.mk: Add level-4p library files. 554
samplec.mk: Add level-4p library files. 555
v0.9.9-09
Gen: The followings are for the further mending of the bug fix in v0.9.8 of the secondary mode stability check not only with particle injection but also with particle removal and position-aware particle management. 1
try_stable1(): Remove the explanation added in v0.9.8 because count_stay() does not exclude injected particles now. 177
try_stable1(): Fix the bug caused by ignoring the possibility that $P_{\max} < Q_n^n + Q_n^{parent(n)}$ and modify/add explanation on this issue. 177
try_stable1(): Remove the explanation added in v0.9.8 because injected particles are included in stay.prime, stay.sec and NOfPToStay[]. 180
try_stable1(): Rename the variable npmove as floating to show its meaning more appropriately. 183
count_stay(): Remove the subtraction of the number of injected particles and the explanation on it. 185
count_stay(): Modify the coding of the addition of NOfPToStay[]. 185
schedule_particle_exchange(): Remove the adjustment of get.prime and get.sec when reb ≤ 0 together with the explanation on it, because they now include injected particles and thus are not necessary to be adjusted. 191
v0.9.9-10
Gen: The followings are for the introduction of oh2_remap_injected_particle() and oh2_remap_injected_particle(), and a few coding style modifications in oh2_inject_particle(). 1
Gen: Add description of oh2_remap_injected_particle(). 41
Gen: Add description of oh2_remove_injected_particle(). 41
Gen: Add description of oh2_remap_injected_particle(). 46
Gen: Add description of oh2_remove_injected_particle(). 47
Gen: Add description of oh2_remove_injected_particle(). 97
Gen: Add description of oh2_remap_injected_particle(). 97
Gen: Add aliases of oh2_remap_injected_particle() and oh2_remap_injected_particle(). 107
nOfLocalPLimit: Add references from oh2_remap_injected_particle() and oh2_remove_injected_particle(). 226
Particles: Add references from oh2_remap_injected_particle() and oh2_remove_injected_particle(). 226

nOfInjections: Add references from oh2_remap_injected_particle() and oh2_remove_injected_particle(). 228
specBase: Add references from oh2_remap_injected_particle() and oh2_remove_injected_particle(). 228
Particle_Spec(): Add uses in oh2_remap_injected_particle() and oh2_remove_injected_particle(). 229
ohhelp2.h: Add prototype of oh2_remap_injected_particle(). 231
ohhelp2.h: Add prototype of oh2_remove_injected_particle(). 231
ohhelp2.h: Add aliases of oh2_remap_injected_particle() and oh2_remove_injected_particle(). 231
ohhelp2.h: Add prototypes of oh2_remap_injected_particle() and oh2_remove_injected_particle(). 232
ohhelp2.h: Add prototypes of oh2_remap_injected_particle_() and oh2_remove_injected_particle_(). 232
oh2_inject_particle(): Modify coding style to use cached local variables for nOfNodes and nOfSpecies. 265
oh2_remap_injected_particle(): Introduced to maintain InjectedParticles[0][][] when a particle is moved after injection. 266
oh2_remove_injected_particle(): Introduced to maintain InjectedParticles[0][][] when a particle is removed after injection. 267
v1.0.0
Gen: Introduction of ohhelp4s.h and ohhelp4s.c. (2012/05/11) 1
v1.0.0-01
Gen: The followings are for newly designed ohhelp4s.h and ohhelp4s.c. 1
Gen: Modify the sentence for the level-4p extension to add the level-4s extension. 1
Gen: Add introductory description of level-4s extension. 17
Gen: Add description about level-4s API. 23
OH_LIB_LEVEL_4S: Add commented-out definition of the macro to oh_config.h for the activation of level-4s extention. 25
OH_LIB_LEVEL_4PS: Introduce this macro being defined iff either OH_LIB_LEVEL_4P or OH_LIB_LEVEL_4P is defined. 25
oh_part.h: Add description of oh4s_inject_particle(). 42
Gen: Add the section to describe level-4s extension. 84
Gen: Add description of level-4s injection and removal. 98
Gen: Add aliases of level-4s's API functions. 107
Gen: Add level-4s library files 129
Gen: Add level-4s library files. 129
Gen: Add description of level-4s. 130
Gen: Add the section for the overview of level-4s extension. 440
ohhelp4s.h: Add this header file with the section for it. 442
ohhelp4s.c: Add this source file with the section for it. 462
v1.0.0-02
Gen: The followings are for changes in libraries other than level-4s to add level-4s extension. 1
Gen: Change the configuration of oh_config.h for the introduction of the level-4s extension. 25
oh_config.h: Change the switch for level-4 definitions from OH_LIB_LEVEL_4P to OH_LIB_LEVEL_4PS 25
STATS_TB_SORT: Change the switch to control the definition from OH_LIB_LEVEL_4P to OH_LIB_LEVEL_4PS. 99
StatsTimeStrings: Change the switch to control the element string definition from OH_LIB_LEVEL_4P to OH_LIB_LEVEL_4PS. 100
OH_POS_AWARE: Change the switch to control the definition from OH_LIB_LEVEL_4P to OH_LIB_LEVEL_4PS. 132
ohhelp_c.h: Surround aliase definitions and prototype delarations for level-4p by #ifdef OH_LIB_LEVEL_4P construct. 343

ohhelp_f.h: Surround aliase definitions for level-4p by `#ifdef OH_LIB_LEVEL_4P` construct. 343
v1.0.0-03
Gen: The followings are bug fix in libraries other than level-4s. 1
`init1()`: Fix the bug by which `TempArray`[n] is mistakingly updated when $n < 0$. 167
`transbound1()`: Insert a workaround code to cope with Intel MPI's bug in `MPI_Alltoall()`. 173
`make_comm_count()`: Insert a workaround code to cope with Intel MPI's bug in `MPI_Alltoall()`. 198
`Vprint_Norank()`: Introduce this macro for `vprint()` without printing rank identifier so as to keep compilers checking the consistency of format string and argments from complaining the inconsistency of `RANKFORMAT` without rank argument. 224
`vprint()`: Use `Vprint_Norank()` instead of `Vprint()` when `verboseMode` < 3 and thus the rank identifier is not printed. 224
`init_subdomain_actively()`: `Message.xyz[i]` is replaced with `Message.xyz[d]`. 290
`upd_real_nbr()`: Add `int` after `const` for the declaration of `nid` which Fujitsu's CC somehow accepted without `int`. 408
v1.1.0
Gen: Introduction of `oh1_neighbors()`, `oh1_families()` and `oh1_accom_mode()`. (2015/07/31) 1
v1.1.0-01
Gen: The followings are for introduction of `oh1_neighbors()`. 1
Gen: Add a description of `oh1_neighbors()`. 26
Gen: Add a description of `oh1_neighbors()`. 33
Gen: Add a description of update of neighboring information. 36
Gen: Add alias of `oh1_neighbors()`. 107
ohhelp1.h: Add a brief description of `NeighborsShadow` and `NeighborsTemp`. 147
ohhelp1.h: Add a brief explanation of `oh1_neighbors()`. 152
ohhelp_c.h: Add alias of `oh1_neighbors()`. 153
ohhelp_f.h: Add alias of `oh1_neighbors()`. 153
ohhelp1.h: Add prototype of `oh1_neighbors()`. 153
ohhelp1.h: Add prototype of `oh1_neighbors_()`. 154
`NeighborsShadow`: Introduce this global pointer. 161
`NeighborsTemp`: Introduce this global pointer. 161
`init1()`: Allocate an array of [3][3^D] for `*nbor` if `oh1_neighbors()` has not been called yet. 165
`init1()`: Let `NeighborsTemp` have `*nbor`, and initialize `NeighborsShadow`[0][] with `*nbor` if `oh1_neighbors()` has been called beforehand. 166
`oh1_neighbors_()`: Introduce the function. 168
`oh1_neighbors()`: Introduce the function. 168
`build_new_comm()`: Add update of `NeighborsShadow`. 209
v1.1.0-02
Gen: The followings are for introduction of `oh1_families()`. 1
Gen: Add a description of `oh1_families()`. 26
Gen: Add a description of `oh1_families()`. 34
Gen: Add a description of update of family information. 36
Gen: Add alias of `oh1_families()`. 107
ohhelp1.h: Add a brief description of `FamIndex` and `FamMembers`. 146
ohhelp1.h: Add a brief explanation of `oh1_families()`. 152
ohhelp_c.h: Add alias of `oh1_families()`. 153
ohhelp_f.h: Add alias of `oh1_families()`. 153
ohhelp1.h: Add prototype of `oh1_families()`. 153
ohhelp1.h: Add prototype of `oh1_families_()`. 154
`FamIndex`: Introduce this global pointer. 169
`FamMembers`: Introduce this global pointer. 169

`oh1_families_()`: Introduce the function. 169
`oh1_families()`: Introduce the function. 169
`try_primary1()`: Add reinitialization of `FamIndex` and `FamMembers`. 175
`build_new_comm()`: Add update of `FamIndex` and `FamMembers`. 208
v1.1.0-03
Gen: The followings are for introduction of `oh1_accom_mode()`. 1
Gen: Add a description of `oh1_accom_mode()`. 26
Gen: Add a description of `oh1_accom_mode()`. 37
Gen: Add alias of `oh1_acc_mode()`. 107
`accMode`: Introduce this global accommodation mode indicator. 135
`ohhelp1.h`: Add a brief explanation of `oh1_accom_mode()`. 152
`ohhelp_c.h`: Add alias of `oh1_accom_mode()`. 153
`ohhelp_f.h`: Add alias of `oh1_accom_mode()`. 153
`ohhelp1.h`: Add prototype of `oh1_accom_mode()`. 153
`ohhelp1.h`: Add prototype of `oh1_accom_mode_()`. 154
`init1()`: Add initialization of `accMode`. 162
`transbound1()`: Add setting of `accMode`. 174
`oh1_accom_mode_()`: Introduce the function. 211
`oh1_accom_mode()`: Introduce the function. 211
v1.1.1
Gen: Bugs of level-1 specific functionalities in `ohhelp1.c` are fixed. (2015/10/23) 1
`try_primary1()`: Corrections of `NOfSend` and `NOfRecv` were done only on `[me]`, i.e., [0][0][n], outside the loop for s. They are now included in the inner-loop for m (`j`) so that the correctoins are performed if $m = n$, i.e., on [0][s][n] for all s. 176
`make_comm_count()`: `putme` were left unchanged when it becomes less than `stay`, i.e., the variable does not hold $\max(0,\ q_{\mathrm{put}}(p) - \sum_{t=0}^{s-1} q_{\mathrm{stay}}(p,t))$ when the second argument of $\max(\cdot,\cdot)$ becomes negative. It is now let have 0 when it is less than `stay`. 199