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

Contents

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.

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

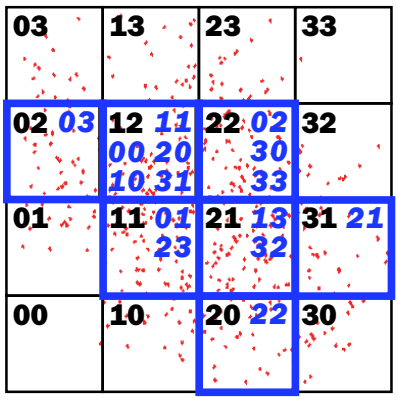


Figure 1: Space domain partitioning.

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 $Q_g = \max_{n\in\mathcal{G}}\{Q_n\}$.

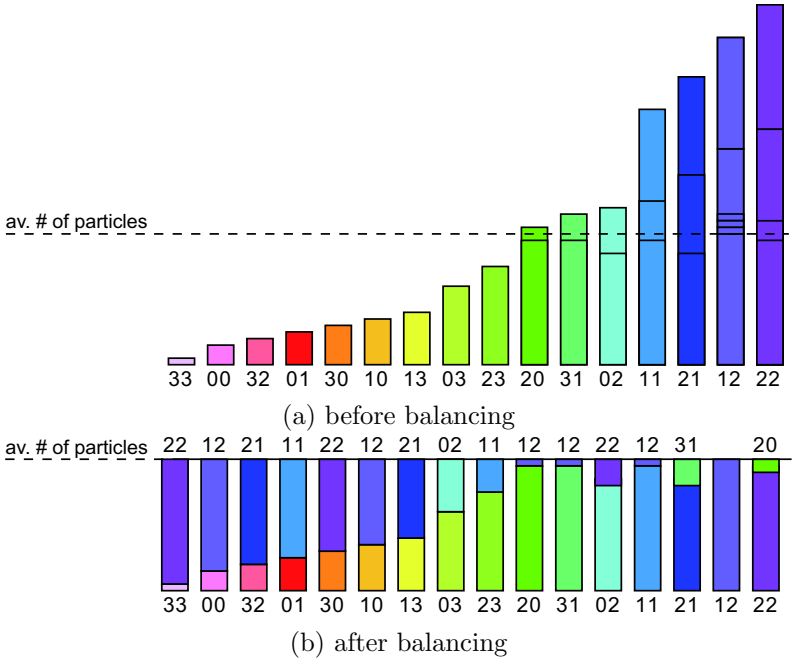
(b4) Assign the subdomain g to the node l as its secondary subdomain and also assign $Q_l^g = (P/N) - Q_l$ particles in the subdomain g to the node l so that $Q_l \leftarrow Q_l + Q_l^g = P/N$. Now Q_g becomes $Q_g - Q_l^g$.

(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

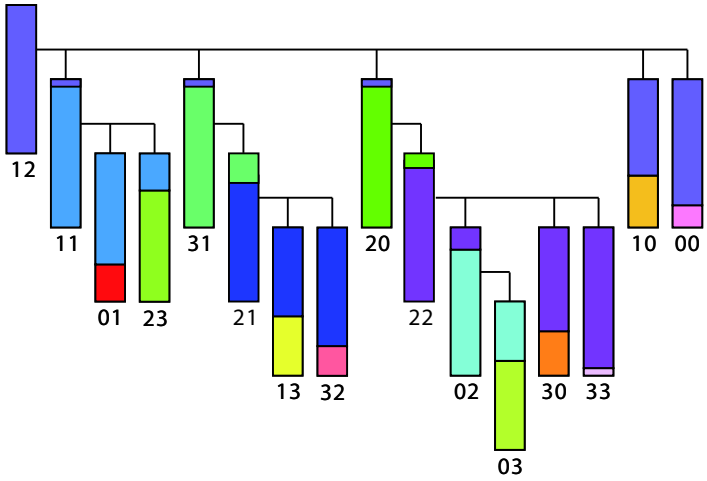


Figure 3: Helpand-helper tree for balancing result in Figure 2(b).

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.

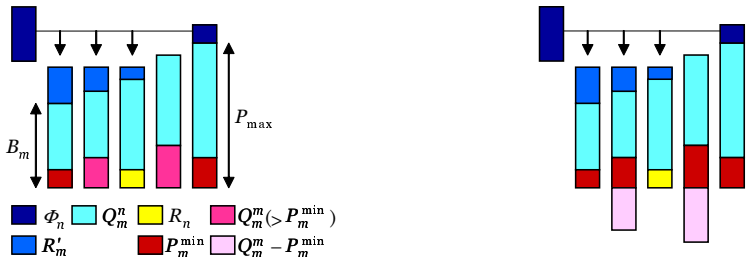
$$P_n^{\min} = \max(0, P_n - \sum_{m \in H(n)} (P_{\max} - P_m^{\min}))$$

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.

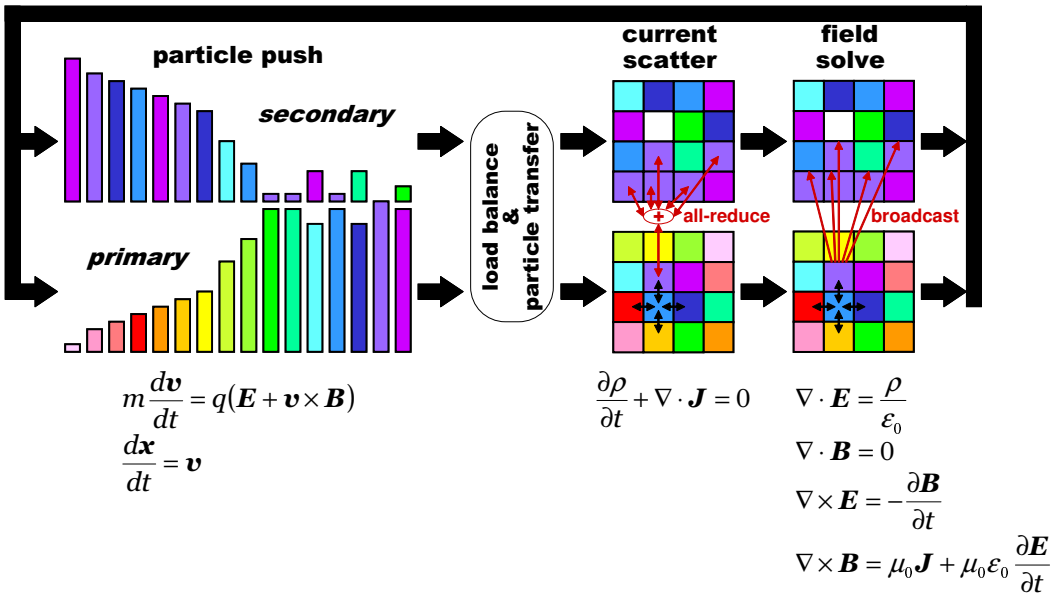


Figure 5: Typical 3D PIC simulator with OhHelp.

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^p_n, 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^p_n, 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

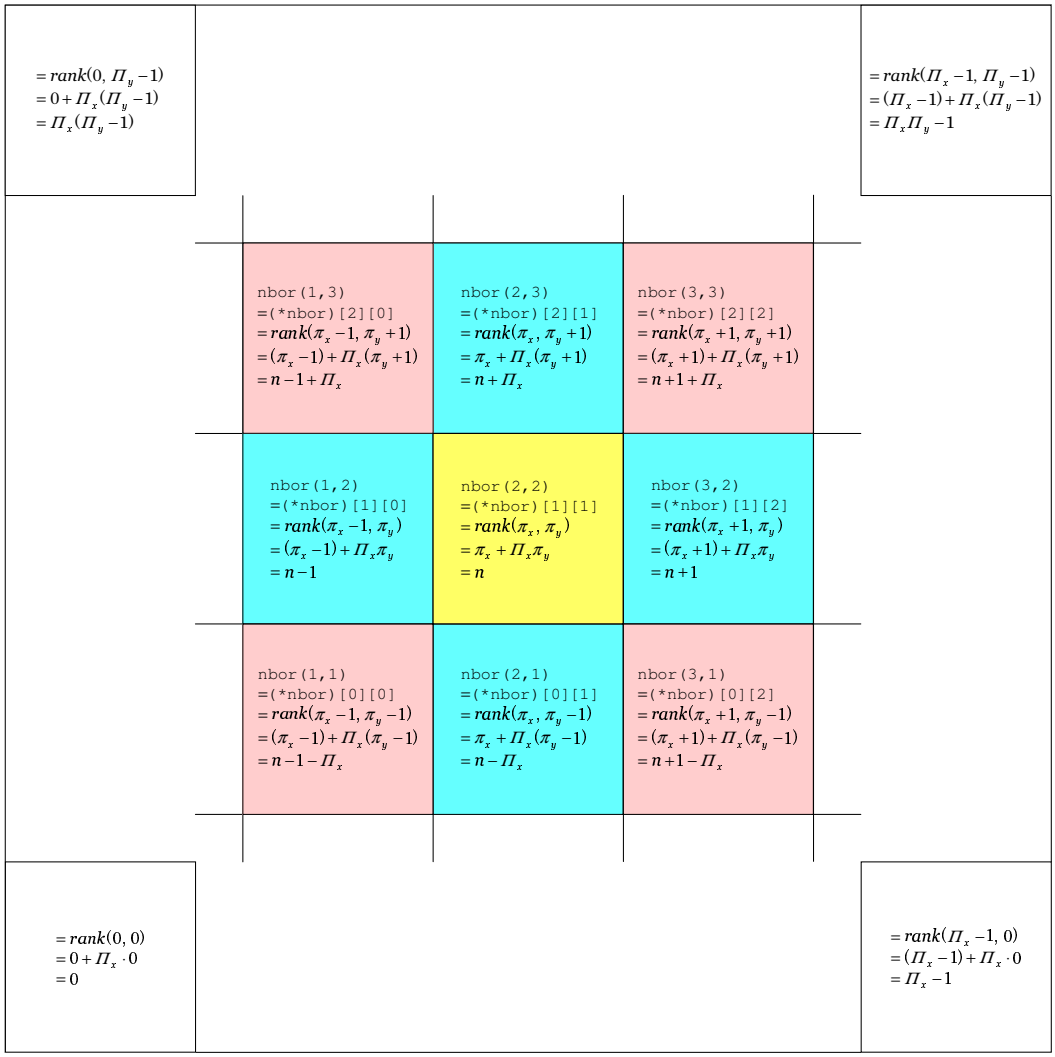


Figure 6: `nbor` and its default setting in $\Pi_x \times \Pi_y$ node coordinate system given by `pcoord`.

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.

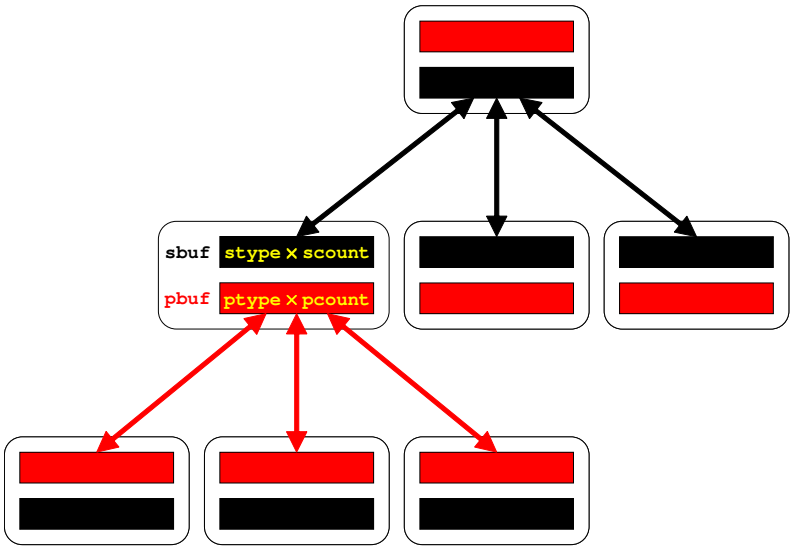


Figure 7: Red-black collective communication by `oh1_broadcast()`, `oh1_all_reduce()` and `oh1_reduce()`.

`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`

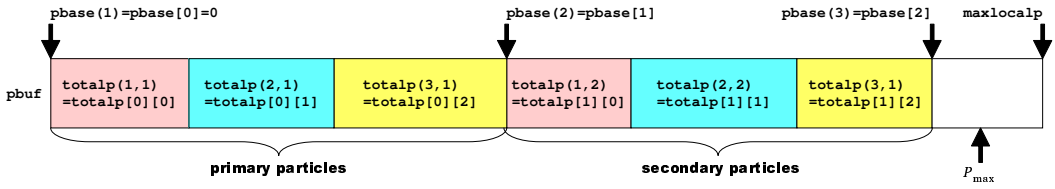


Figure 8: Particle buffer and related variables.

`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.

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

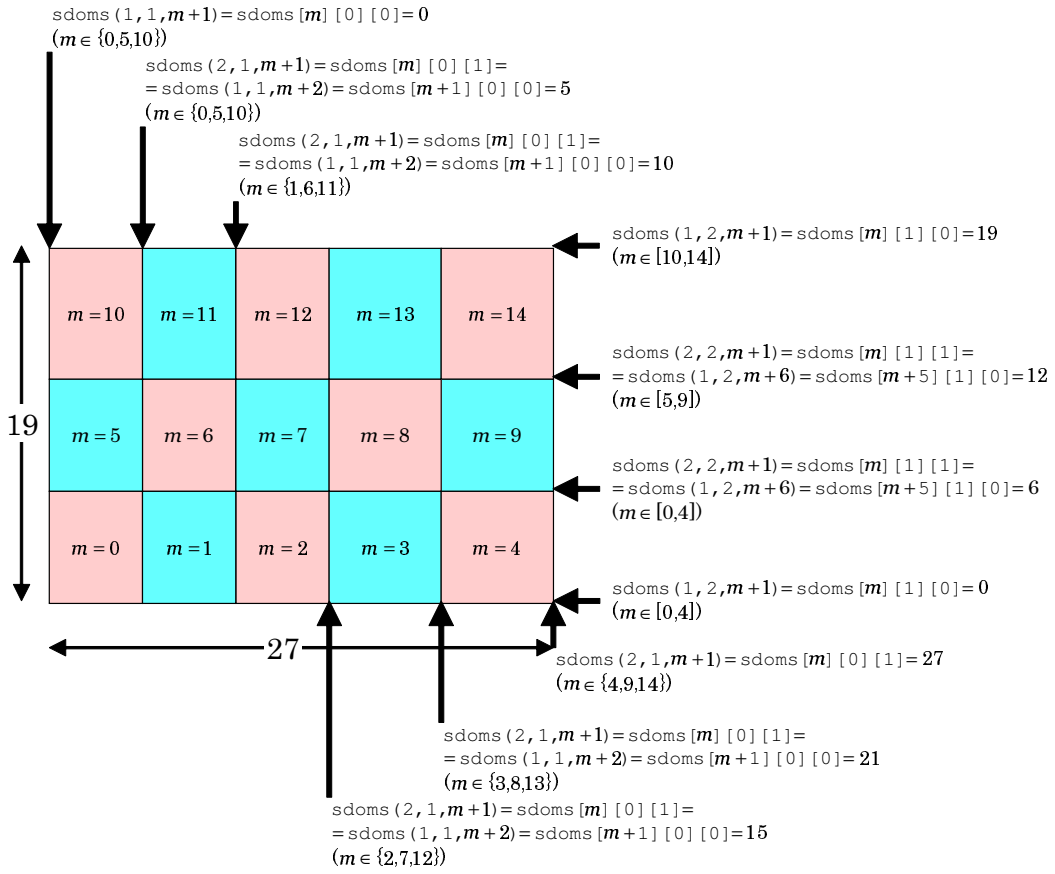


Figure 9: `sdoms` and its default setting for space domain of 27×19 given by `scoord` and node coordinate system of 5×3 given by `pcoord`.

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

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

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 $[\Delta_1^l,\ \Delta_1^u - 1] \times \ldots \times [\Delta_D^l,\ \Delta_D^u - 1]$.

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

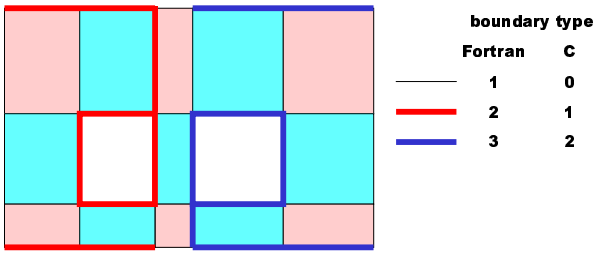


Figure 10: Complicated subdomains and their boundaries with walls and holes.

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.

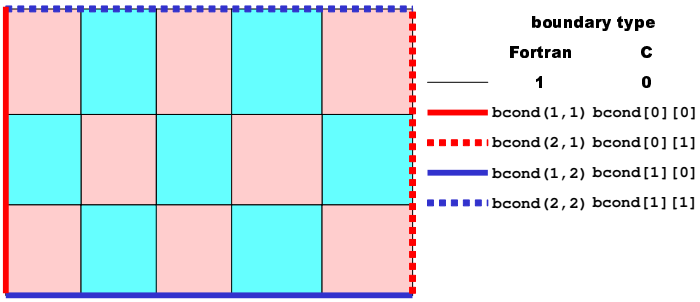


Figure 11: Default setting of subdomain boundaries.

`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)`.

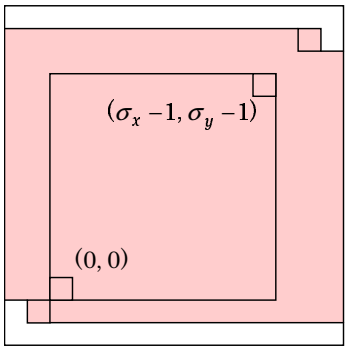


Figure 12: Type f field-array of $(\sigma_x + 5) \times (\sigma_y + 5)$ for a subdomain of $[0, \sigma_x - 1] \times [0, \sigma_y - 1]$ and its elements (painted) broadcasted by `oh3_bcast_field()` with setting of $e_l^b(f) = -1$ and $e_u^b(f) = 2$.

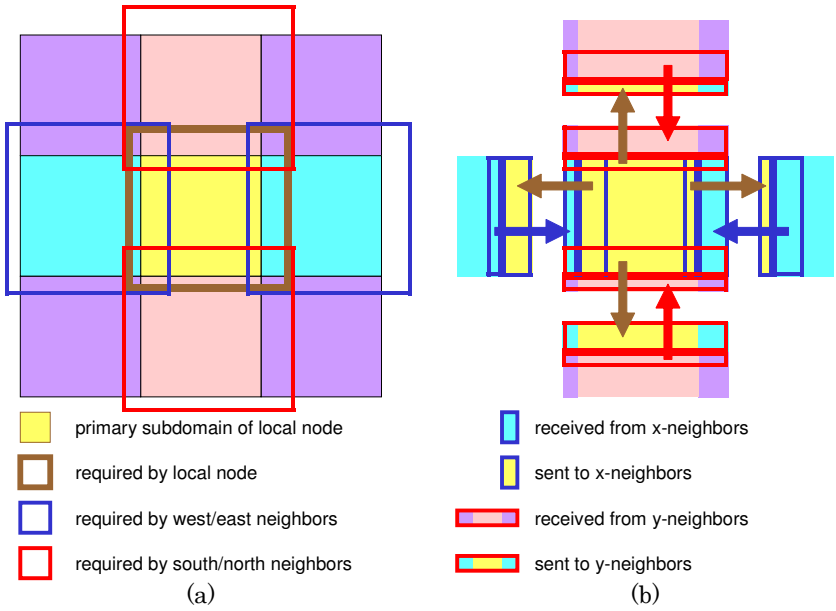


Figure 13: Field-array with downward communication $(e_f, e_t, s) = (0, 0, 2)$ and upward communication $(e_f, e_t, s) = (-1, -1, 1)$.

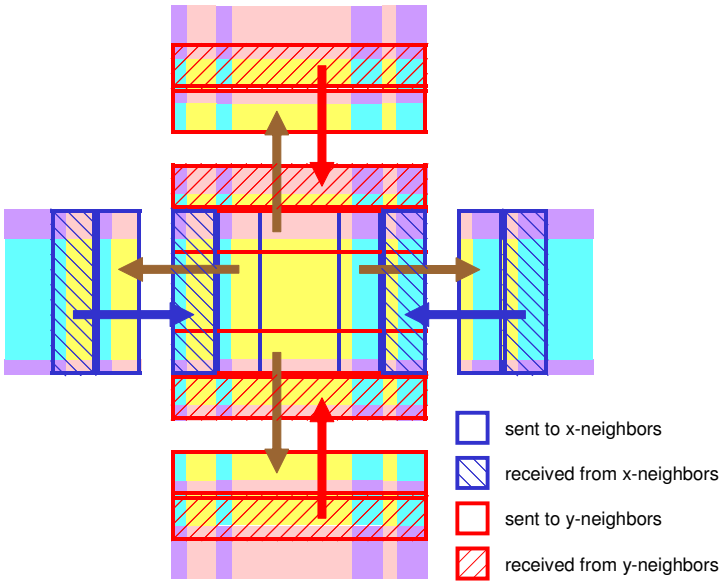


Figure 14: Field-array with downward communication $(e_f, e_t, s) = (-1, 2, 3)$ and upward communication $(e_f, e_t, s) = (-1, -4, 3)$.

`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.

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

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 $e^r_l(f)$ and $e^r_u(f)$ 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 $e^r_l(f)$ and $e^r_u(f)$ 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.,

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

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 $(2t\texttt{+}k)$-th ($k \in \{0, 1\}$) call of `oh4p_transbound()` and in $(0, 0, 0)$ are in

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

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 $\phi_d^\beta = \texttt{fsizes}(\beta, d, F{+}1)$ or $\phi_d^\beta = \texttt{fsizes}[F][d][\beta]$ 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. $\mathtt{h}(x, y, z, s, c) > 0$. For C coded simulators, `oh4s_transbound()` will let $\mathtt{i}[c][s][z][y][x]$ 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 $\mathtt{h}[c][s][z][y][x] > 0$. 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.

$$\begin{array}{lll} s^l_{x,y,z} = \texttt{ctype(2,2)} & n_l = \texttt{ctype(3,2)} & d^l_{x,y,z} = s^l_{x,y,z} + n_l \\ s^u_{x,y,z} = \sigma_{x,y,z} + \texttt{ctype(2,1)} & n_u = \texttt{ctype(3,1)} & d^u_{x,y,z} = s^u_{x,y,z} - n_u \end{array}$$

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
```

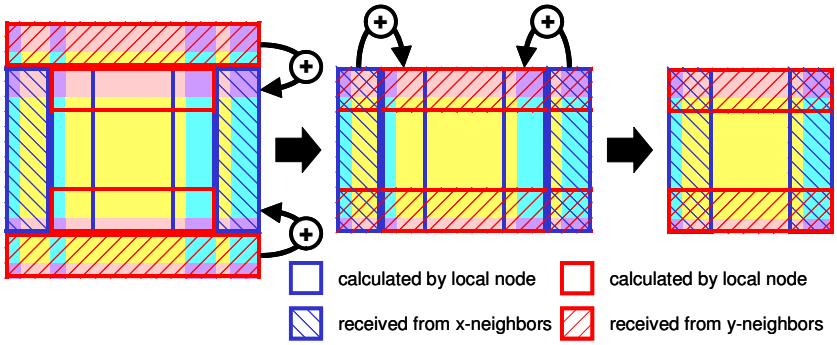


Figure 15: Adding boundary planes of current density vectors.

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.

$$\begin{array}{rll} s^l_{x,y,z} = \texttt{ctype[1][1]} & n_l = \texttt{ctype[1][2]} & d^l_{x,y,z} = s^l_{x,y,z} + n_l \\ s^u_{x,y,z} = \sigma_{x,y,z} + \texttt{ctype[0][1]} & n_u = \texttt{ctype[0][2]} & d^u_{x,y,z} = s^u_{x,y,z} - n_u \end{array}$$

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.

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

Underlined number refers to the page where the specification of corresponding entry is described.

G

H

I

J

L

M

N

O

T

V

Z