



OpenMP Application Programming Interface

Version 5.0 November 2018

Copyright ©1997-2018 OpenMP Architecture Review Board.

Permission to copy without fee all or part of this material is granted, provided the OpenMP Architecture Review Board copyright notice and the title of this document appear. Notice is given that copying is by permission of the OpenMP Architecture Review Board.

This page intentionally left blank.

Contents

1 Introduction	1
1.1 Scope	1
1.2 Glossary	2
1.2.1 Threading Concepts	2
1.2.2 OpenMP Language Terminology	2
1.2.3 Loop Terminology	8
1.2.4 Synchronization Terminology	9
1.2.5 Tasking Terminology	10
1.2.6 Data Terminology	12
1.2.7 Implementation Terminology	17
1.2.8 Tool Terminology	17
1.3 Execution Model	20
1.4 Memory Model	23
1.4.1 Structure of the OpenMP Memory Model	23
1.4.2 Device Data Environments	24
1.4.3 Memory Management	25
1.4.4 The Flush Operation	25
1.4.5 Flush Synchronization and <i>Happens Before</i>	27
1.4.6 OpenMP Memory Consistency	28
1.5 Tool Interfaces	29
1.5.1 OMPT	29
1.5.2 OMPD	30

1.6	OpenMP Compliance	31
1.7	Normative References	31
1.8	Organization of this Document	34
2	Directives	37
2.1	Directive Format	38
2.1.1	Fixed Source Form Directives	41
2.1.2	Free Source Form Directives	41
2.1.3	Stand-Alone Directives	42
2.1.4	Array Shaping	43
2.1.5	Array Sections	44
2.1.6	Iterators	47
2.2	Conditional Compilation	49
2.2.1	Fixed Source Form Conditional Compilation Sentinels	50
2.2.2	Free Source Form Conditional Compilation Sentinel	50
2.3	Variant Directives	51
2.3.1	OpenMP Context	51
2.3.2	Context Selectors	53
2.3.3	Matching and Scoring Context Selectors	55
2.3.4	Metadirectives	56
2.3.5	declare variant Directive	58
2.4	requires Directive	60
2.5	Internal Control Variables	63
2.5.1	ICV Descriptions	64
2.5.2	ICV Initialization	66
2.5.3	Modifying and Retrieving ICV Values	68
2.5.4	How ICVs are Scoped	70
2.5.4.1	How the Per-Data Environment ICVs Work	72
2.5.5	ICV Override Relationships	72
2.6	parallel Construct	74
2.6.1	Determining the Number of Threads for a parallel Region	78
2.6.2	Controlling OpenMP Thread Affinity	80
2.7	teams Construct	82

2.8	Worksharing Constructs	86
2.8.1	sections Construct	86
2.8.2	single Construct	89
2.8.3	workshare Construct	92
2.9	Loop-Related Directives	95
2.9.1	Canonical Loop Form	95
2.9.2	Worksharing-Loop Construct	101
2.9.2.1	Determining the Schedule of a Worksharing-Loop	109
2.9.3	SIMD Directives	110
2.9.3.1	simd Construct	110
2.9.3.2	Worksharing-Loop SIMD Construct	114
2.9.3.3	declare simd Directive	116
2.9.4	distribute Loop Constructs	120
2.9.4.1	distribute Construct	120
2.9.4.2	distribute simd Construct	123
2.9.4.3	Distribute Parallel Worksharing-Loop Construct	125
2.9.4.4	Distribute Parallel Worksharing-Loop SIMD Construct	126
2.9.5	loop Construct	128
2.9.6	scan Directive	132
2.10	Tasking Constructs	135
2.10.1	task Construct	135
2.10.2	taskloop Construct	140
2.10.3	taskloop simd Construct	146
2.10.4	taskyield Construct	147
2.10.5	Initial Task	148
2.10.6	Task Scheduling	149
2.11	Memory Management Directives	152
2.11.1	Memory Spaces	152
2.11.2	Memory Allocators	152
2.11.3	allocate Directive	156
2.11.4	allocate Clause	158
2.12	Device Directives	160
2.12.1	Device Initialization	160

2.12.2	target data Construct	161
2.12.3	target enter data Construct	164
2.12.4	target exit data Construct	166
2.12.5	target Construct	170
2.12.6	target update Construct	176
2.12.7	declare target Directive	180
2.13	Combined Constructs	185
2.13.1	Parallel Worksharing-Loop Construct	185
2.13.2	parallel loop Construct	186
2.13.3	parallel sections Construct	188
2.13.4	parallel workshare Construct	189
2.13.5	Parallel Worksharing-Loop SIMD Construct	190
2.13.6	parallel master Construct	191
2.13.7	master taskloop Construct	192
2.13.8	master taskloop simd Construct	194
2.13.9	parallel master taskloop Construct	195
2.13.10	parallel master taskloop simd Construct	196
2.13.11	teams distribute Construct	197
2.13.12	teams distribute simd Construct	198
2.13.13	Teams Distribute Parallel Worksharing-Loop Construct	200
2.13.14	Teams Distribute Parallel Worksharing-Loop SIMD Construct	201
2.13.15	teams loop Construct	202
2.13.16	target parallel Construct	203
2.13.17	Target Parallel Worksharing-Loop Construct	205
2.13.18	Target Parallel Worksharing-Loop SIMD Construct	206
2.13.19	target parallel loop Construct	208
2.13.20	target simd Construct	209
2.13.21	target teams Construct	210
2.13.22	target teams distribute Construct	211
2.13.23	target teams distribute simd Construct	213
2.13.24	target teams loop Construct	214
2.13.25	Target Teams Distribute Parallel Worksharing-Loop Construct	215
2.13.26	Target Teams Distribute Parallel Worksharing-Loop SIMD Construct	216

2.14	Clauses on Combined and Composite Constructs	218
2.15	if Clause	220
2.16	master Construct	221
2.17	Synchronization Constructs and Clauses	223
2.17.1	critical Construct	223
2.17.2	barrier Construct	226
2.17.3	Implicit Barriers	228
2.17.4	Implementation-Specific Barriers	230
2.17.5	taskwait Construct	230
2.17.6	taskgroup Construct	232
2.17.7	atomic Construct	234
2.17.8	flush Construct	242
2.17.8.1	Implicit Flushes	246
2.17.9	ordered Construct	250
2.17.10	Depend Objects	254
2.17.10.1	depobj Construct	254
2.17.11	depend Clause	255
2.17.12	Synchronization Hints	260
2.18	Cancellation Constructs	263
2.18.1	cancel Construct	263
2.18.2	cancellation point Construct	267
2.19	Data Environment	269
2.19.1	Data-Sharing Attribute Rules	269
2.19.1.1	Variables Referenced in a Construct	270
2.19.1.2	Variables Referenced in a Region but not in a Construct	273
2.19.2	threadprivate Directive	274
2.19.3	List Item Privatization	279
2.19.4	Data-Sharing Attribute Clauses	282
2.19.4.1	default Clause	282
2.19.4.2	shared Clause	283
2.19.4.3	private Clause	285
2.19.4.4	firstprivate Clause	286
2.19.4.5	lastprivate Clause	288

2.19.4.6	linear Clause	290
2.19.5	Reduction Clauses and Directives	293
2.19.5.1	Properties Common To All Reduction Clauses	294
2.19.5.2	Reduction Scoping Clauses	299
2.19.5.3	Reduction Participating Clauses	300
2.19.5.4	reduction Clause	300
2.19.5.5	task_reduction Clause	303
2.19.5.6	in_reduction Clause	303
2.19.5.7	declare reduction Directive	304
2.19.6	Data Copying Clauses	309
2.19.6.1	copyin Clause	310
2.19.6.2	copyprivate Clause	312
2.19.7	Data-Mapping Attribute Rules, Clauses, and Directives	314
2.19.7.1	map Clause	315
2.19.7.2	defaultmap Clause	324
2.19.7.3	declare mapper Directive	326
2.20	Nesting of Regions	328

3 Runtime Library Routines 331

3.1	Runtime Library Definitions	332
3.2	Execution Environment Routines	334
3.2.1	omp_set_num_threads	334
3.2.2	omp_get_num_threads	335
3.2.3	omp_get_max_threads	336
3.2.4	omp_get_thread_num	337
3.2.5	omp_get_num_procs	338
3.2.6	omp_in_parallel	339
3.2.7	omp_set_dynamic	340
3.2.8	omp_get_dynamic	341
3.2.9	omp_get_cancellation	342
3.2.10	omp_set_nested	343
3.2.11	omp_get_nested	344
3.2.12	omp_set_schedule	345
3.2.13	omp_get_schedule	347

3.2.14	<code>omp_get_thread_limit</code>	348
3.2.15	<code>omp_get_supported_active_levels</code>	349
3.2.16	<code>omp_set_max_active_levels</code>	350
3.2.17	<code>omp_get_max_active_levels</code>	351
3.2.18	<code>omp_get_level</code>	352
3.2.19	<code>omp_get_ancestor_thread_num</code>	353
3.2.20	<code>omp_get_team_size</code>	354
3.2.21	<code>omp_get_active_level</code>	355
3.2.22	<code>omp_in_final</code>	356
3.2.23	<code>omp_get_proc_bind</code>	357
3.2.24	<code>omp_get_num_places</code>	358
3.2.25	<code>omp_get_place_num_procs</code>	359
3.2.26	<code>omp_get_place_proc_ids</code>	360
3.2.27	<code>omp_get_place_num</code>	362
3.2.28	<code>omp_get_partition_num_places</code>	362
3.2.29	<code>omp_get_partition_place_nums</code>	363
3.2.30	<code>omp_set_affinity_format</code>	364
3.2.31	<code>omp_get_affinity_format</code>	366
3.2.32	<code>omp_display_affinity</code>	367
3.2.33	<code>omp_capture_affinity</code>	368
3.2.34	<code>omp_set_default_device</code>	369
3.2.35	<code>omp_get_default_device</code>	370
3.2.36	<code>omp_get_num_devices</code>	371
3.2.37	<code>omp_get_device_num</code>	372
3.2.38	<code>omp_get_num_teams</code>	373
3.2.39	<code>omp_get_team_num</code>	374
3.2.40	<code>omp_is_initial_device</code>	375
3.2.41	<code>omp_get_initial_device</code>	376
3.2.42	<code>omp_get_max_task_priority</code>	377
3.2.43	<code>omp_pause_resource</code>	378
3.2.44	<code>omp_pause_resource_all</code>	380
3.3	Lock Routines	381
3.3.1	<code>omp_init_lock</code> and <code>omp_init_nest_lock</code>	384

3.3.2	<code>omp_init_lock_with_hint</code> and <code>omp_init_nest_lock_with_hint</code>	385
3.3.3	<code>omp_destroy_lock</code> and <code>omp_destroy_nest_lock</code>	387
3.3.4	<code>omp_set_lock</code> and <code>omp_set_nest_lock</code>	388
3.3.5	<code>omp_unset_lock</code> and <code>omp_unset_nest_lock</code>	390
3.3.6	<code>omp_test_lock</code> and <code>omp_test_nest_lock</code>	392
3.4	Timing Routines	394
3.4.1	<code>omp_get_wtime</code>	394
3.4.2	<code>omp_get_wtick</code>	395
3.5	Event Routine	396
3.5.1	<code>omp_fulfill_event</code>	396
3.6	Device Memory Routines	397
3.6.1	<code>omp_target_alloc</code>	397
3.6.2	<code>omp_target_free</code>	399
3.6.3	<code>omp_target_is_present</code>	400
3.6.4	<code>omp_target_memcpy</code>	400
3.6.5	<code>omp_target_memcpy_rect</code>	402
3.6.6	<code>omp_target_associate_ptr</code>	403
3.6.7	<code>omp_target_disassociate_ptr</code>	405
3.7	Memory Management Routines	406
3.7.1	Memory Management Types	406
3.7.2	<code>omp_init_allocator</code>	409
3.7.3	<code>omp_destroy_allocator</code>	410
3.7.4	<code>omp_set_default_allocator</code>	411
3.7.5	<code>omp_get_default_allocator</code>	412
3.7.6	<code>omp_alloc</code>	413
3.7.7	<code>omp_free</code>	414
3.8	Tool Control Routine	415
4	OMPT Interface	419
4.1	OMPT Interfaces Definitions	419
4.2	Activating a First-Party Tool	420
4.2.1	<code>ompt_start_tool</code>	420
4.2.2	Determining Whether a First-Party Tool Should be Initialized	421

4.2.3	Initializing a First-Party Tool	423
4.2.3.1	Binding Entry Points in the OMPT Callback Interface	424
4.2.4	Monitoring Activity on the Host with OMPT	425
4.2.5	Tracing Activity on Target Devices with OMPT	427
4.3	Finalizing a First-Party Tool	432
4.4	OMPT Data Types	433
4.4.1	Tool Initialization and Finalization	433
4.4.2	Callbacks	434
4.4.3	Tracing	435
4.4.3.1	Record Type	435
4.4.3.2	Native Record Kind	435
4.4.3.3	Native Record Abstract Type	436
4.4.3.4	Record Type	436
4.4.4	Miscellaneous Type Definitions	438
4.4.4.1	<code>ompt_callback_t</code>	438
4.4.4.2	<code>ompt_set_result_t</code>	438
4.4.4.3	<code>ompt_id_t</code>	439
4.4.4.4	<code>ompt_data_t</code>	440
4.4.4.5	<code>ompt_device_t</code>	441
4.4.4.6	<code>ompt_device_time_t</code>	441
4.4.4.7	<code>ompt_buffer_t</code>	441
4.4.4.8	<code>ompt_buffer_cursor_t</code>	442
4.4.4.9	<code>ompt_dependence_t</code>	442
4.4.4.10	<code>ompt_thread_t</code>	443
4.4.4.11	<code>ompt_scope_endpoint_t</code>	443
4.4.4.12	<code>ompt_dispatch_t</code>	444
4.4.4.13	<code>ompt_sync_region_t</code>	444
4.4.4.14	<code>ompt_target_data_op_t</code>	444
4.4.4.15	<code>ompt_work_t</code>	445
4.4.4.16	<code>ompt_mutex_t</code>	445
4.4.4.17	<code>ompt_native_mon_flag_t</code>	446
4.4.4.18	<code>ompt_task_flag_t</code>	446
4.4.4.19	<code>ompt_task_status_t</code>	447

4.4.4.20	<code>ompt_target_t</code>	448
4.4.4.21	<code>ompt_parallel_flag_t</code>	448
4.4.4.22	<code>ompt_target_map_flag_t</code>	449
4.4.4.23	<code>ompt_dependence_type_t</code>	450
4.4.4.24	<code>ompt_cancel_flag_t</code>	450
4.4.4.25	<code>ompt_hwid_t</code>	451
4.4.4.26	<code>ompt_state_t</code>	452
4.4.4.27	<code>ompt_frame_t</code>	454
4.4.4.28	<code>ompt_frame_flag_t</code>	455
4.4.4.29	<code>ompt_wait_id_t</code>	456
4.5	OMPT Tool Callback Signatures and Trace Records	457
4.5.1	Initialization and Finalization Callback Signature	457
4.5.1.1	<code>ompt_initialize_t</code>	457
4.5.1.2	<code>ompt_finalize_t</code>	458
4.5.2	Event Callback Signatures and Trace Records	459
4.5.2.1	<code>ompt_callback_thread_begin_t</code>	459
4.5.2.2	<code>ompt_callback_thread_end_t</code>	460
4.5.2.3	<code>ompt_callback_parallel_begin_t</code>	461
4.5.2.4	<code>ompt_callback_parallel_end_t</code>	463
4.5.2.5	<code>ompt_callback_work_t</code>	464
4.5.2.6	<code>ompt_callback_dispatch_t</code>	465
4.5.2.7	<code>ompt_callback_task_create_t</code>	467
4.5.2.8	<code>ompt_callback_dependences_t</code>	468
4.5.2.9	<code>ompt_callback_task_dependence_t</code>	470
4.5.2.10	<code>ompt_callback_task_schedule_t</code>	470
4.5.2.11	<code>ompt_callback_implicit_task_t</code>	471
4.5.2.12	<code>ompt_callback_master_t</code>	473
4.5.2.13	<code>ompt_callback_sync_region_t</code>	474
4.5.2.14	<code>ompt_callback_mutex_acquire_t</code>	476
4.5.2.15	<code>ompt_callback_mutex_t</code>	477
4.5.2.16	<code>ompt_callback_nest_lock_t</code>	479
4.5.2.17	<code>ompt_callback_flush_t</code>	480
4.5.2.18	<code>ompt_callback_cancel_t</code>	481

4.5.2.19	<code>ompt_callback_device_initialize_t</code>	482
4.5.2.20	<code>ompt_callback_device_finalize_t</code>	484
4.5.2.21	<code>ompt_callback_device_load_t</code>	484
4.5.2.22	<code>ompt_callback_device_unload_t</code>	486
4.5.2.23	<code>ompt_callback_buffer_request_t</code>	486
4.5.2.24	<code>ompt_callback_buffer_complete_t</code>	487
4.5.2.25	<code>ompt_callback_target_data_op_t</code>	488
4.5.2.26	<code>ompt_callback_target_t</code>	490
4.5.2.27	<code>ompt_callback_target_map_t</code>	492
4.5.2.28	<code>ompt_callback_target_submit_t</code>	494
4.5.2.29	<code>ompt_callback_control_tool_t</code>	495
4.6	OMPT Runtime Entry Points for Tools	497
4.6.1	Entry Points in the OMPT Callback Interface	497
4.6.1.1	<code>ompt_enumerate_states_t</code>	498
4.6.1.2	<code>ompt_enumerate_mutex_impls_t</code>	499
4.6.1.3	<code>ompt_set_callback_t</code>	500
4.6.1.4	<code>ompt_get_callback_t</code>	502
4.6.1.5	<code>ompt_get_thread_data_t</code>	503
4.6.1.6	<code>ompt_get_num_procs_t</code>	503
4.6.1.7	<code>ompt_get_num_places_t</code>	504
4.6.1.8	<code>ompt_get_place_proc_ids_t</code>	505
4.6.1.9	<code>ompt_get_place_num_t</code>	506
4.6.1.10	<code>ompt_get_partition_place_nums_t</code>	507
4.6.1.11	<code>ompt_get_proc_id_t</code>	508
4.6.1.12	<code>ompt_get_state_t</code>	508
4.6.1.13	<code>ompt_get_parallel_info_t</code>	510
4.6.1.14	<code>ompt_get_task_info_t</code>	512
4.6.1.15	<code>ompt_get_task_memory_t</code>	514
4.6.1.16	<code>ompt_get_target_info_t</code>	515
4.6.1.17	<code>ompt_get_num_devices_t</code>	516
4.6.1.18	<code>ompt_get_unique_id_t</code>	517
4.6.1.19	<code>ompt_finalize_tool_t</code>	517

4.6.2	Entry Points in the OMPT Device Tracing Interface	518
4.6.2.1	<code>ompt_get_device_num_procs_t</code>	518
4.6.2.2	<code>ompt_get_device_time_t</code>	519
4.6.2.3	<code>ompt_translate_time_t</code>	520
4.6.2.4	<code>ompt_set_trace_ompt_t</code>	521
4.6.2.5	<code>ompt_set_trace_native_t</code>	522
4.6.2.6	<code>ompt_start_trace_t</code>	523
4.6.2.7	<code>ompt_pause_trace_t</code>	524
4.6.2.8	<code>ompt_flush_trace_t</code>	525
4.6.2.9	<code>ompt_stop_trace_t</code>	526
4.6.2.10	<code>ompt_advance_buffer_cursor_t</code>	527
4.6.2.11	<code>ompt_get_record_type_t</code>	528
4.6.2.12	<code>ompt_get_record_ompt_t</code>	529
4.6.2.13	<code>ompt_get_record_native_t</code>	530
4.6.2.14	<code>ompt_get_record_abstract_t</code>	531
4.6.3	Lookup Entry Points: <code>ompt_function_lookup_t</code>	531

5 OMPD Interface 533

5.1	OMPD Interfaces Definitions	534
5.2	Activating an OMPD Tool	534
5.2.1	Enabling the Runtime for OMPD	534
5.2.2	<code>ompd_dll_locations</code>	535
5.2.3	<code>ompd_dll_locations_valid</code>	536
5.3	OMPD Data Types	536
5.3.1	Size Type	536
5.3.2	Wait ID Type	537
5.3.3	Basic Value Types	537
5.3.4	Address Type	538
5.3.5	Frame Information Type	538
5.3.6	System Device Identifiers	539
5.3.7	Native Thread Identifiers	539
5.3.8	OMPD Handle Types	540
5.3.9	OMPD Scope Types	541
5.3.10	ICV ID Type	542

5.3.11	Tool Context Types	542
5.3.12	Return Code Types	543
5.3.13	Primitive Type Sizes	544
5.4	OMPD Tool Callback Interface	545
5.4.1	Memory Management of OMPD Library	545
5.4.1.1	<code>ompd_callback_memory_alloc_fn_t</code>	546
5.4.1.2	<code>ompd_callback_memory_free_fn_t</code>	546
5.4.2	Context Management and Navigation	547
5.4.2.1	<code>ompd_callback_get_thread_context_for_thread_id</code> <code>_fn_t</code>	547
5.4.2.2	<code>ompd_callback_sizeof_fn_t</code>	549
5.4.3	Accessing Memory in the OpenMP Program or Runtime	549
5.4.3.1	<code>ompd_callback_symbol_addr_fn_t</code>	550
5.4.3.2	<code>ompd_callback_memory_read_fn_t</code>	551
5.4.3.3	<code>ompd_callback_memory_write_fn_t</code>	553
5.4.4	Data Format Conversion: <code>ompd_callback_device_host_fn_t</code> . . .	554
5.4.5	Output: <code>ompd_callback_print_string_fn_t</code>	556
5.4.6	The Callback Interface	556
5.5	OMPD Tool Interface Routines	558
5.5.1	Per OMPD Library Initialization and Finalization	558
5.5.1.1	<code>ompd_initialize</code>	558
5.5.1.2	<code>ompd_get_api_version</code>	559
5.5.1.3	<code>ompd_get_version_string</code>	560
5.5.1.4	<code>ompd_finalize</code>	561
5.5.2	Per OpenMP Process Initialization and Finalization	562
5.5.2.1	<code>ompd_process_initialize</code>	562
5.5.2.2	<code>ompd_device_initialize</code>	563
5.5.2.3	<code>ompd_rel_address_space_handle</code>	564
5.5.3	Thread and Signal Safety	565
5.5.4	Address Space Information	565
5.5.4.1	<code>ompd_get_omp_version</code>	565
5.5.4.2	<code>ompd_get_omp_version_string</code>	566

5.5.5	Thread Handles	567
5.5.5.1	<code>ompd_get_thread_in_parallel</code>	567
5.5.5.2	<code>ompd_get_thread_handle</code>	568
5.5.5.3	<code>ompd_rel_thread_handle</code>	569
5.5.5.4	<code>ompd_thread_handle_compare</code>	570
5.5.5.5	<code>ompd_get_thread_id</code>	570
5.5.6	Parallel Region Handles	571
5.5.6.1	<code>ompd_get_curr_parallel_handle</code>	571
5.5.6.2	<code>ompd_get_enclosing_parallel_handle</code>	572
5.5.6.3	<code>ompd_get_task_parallel_handle</code>	573
5.5.6.4	<code>ompd_rel_parallel_handle</code>	574
5.5.6.5	<code>ompd_parallel_handle_compare</code>	575
5.5.7	Task Handles	576
5.5.7.1	<code>ompd_get_curr_task_handle</code>	576
5.5.7.2	<code>ompd_get_generating_task_handle</code>	577
5.5.7.3	<code>ompd_get_scheduling_task_handle</code>	578
5.5.7.4	<code>ompd_get_task_in_parallel</code>	579
5.5.7.5	<code>ompd_rel_task_handle</code>	580
5.5.7.6	<code>ompd_task_handle_compare</code>	580
5.5.7.7	<code>ompd_get_task_function</code>	581
5.5.7.8	<code>ompd_get_task_frame</code>	582
5.5.7.9	<code>ompd_enumerate_states</code>	583
5.5.7.10	<code>ompd_get_state</code>	585
5.5.8	Display Control Variables	586
5.5.8.1	<code>ompd_get_display_control_vars</code>	586
5.5.8.2	<code>ompd_rel_display_control_vars</code>	587
5.5.9	Accessing Scope-Specific Information	588
5.5.9.1	<code>ompd_enumerate_icvs</code>	588
5.5.9.2	<code>ompd_get_icv_from_scope</code>	590
5.5.9.3	<code>ompd_get_icv_string_from_scope</code>	591
5.5.9.4	<code>ompd_get_tool_data</code>	592
5.6	Runtime Entry Points for OMPD	594
5.6.1	Beginning Parallel Regions	594

5.6.2	Ending Parallel Regions	595
5.6.3	Beginning Task Regions	595
5.6.4	Ending Task Regions	596
5.6.5	Beginning OpenMP Threads	597
5.6.6	Ending OpenMP Threads	597
5.6.7	Initializing OpenMP Devices	598
5.6.8	Finalizing OpenMP Devices	599
6	Environment Variables	601
6.1	OMP_SCHEDULE	601
6.2	OMP_NUM_THREADS	602
6.3	OMP_DYNAMIC	603
6.4	OMP_PROC_BIND	604
6.5	OMP_PLACES	605
6.6	OMP_STACKSIZE	607
6.7	OMP_WAIT_POLICY	608
6.8	OMP_MAX_ACTIVE_LEVELS	608
6.9	OMP_NESTED	609
6.10	OMP_THREAD_LIMIT	610
6.11	OMP_CANCELLATION	610
6.12	OMP_DISPLAY_ENV	611
6.13	OMP_DISPLAY_AFFINITY	612
6.14	OMP_AFFINITY_FORMAT	613
6.15	OMP_DEFAULT_DEVICE	615
6.16	OMP_MAX_TASK_PRIORITY	615
6.17	OMP_TARGET_OFFLOAD	615
6.18	OMP_TOOL	616
6.19	OMP_TOOL_LIBRARIES	617
6.20	OMP_DEBUG	617
6.21	OMP_ALLOCATOR	618
A	OpenMP Implementation-Defined Behaviors	619
B	Features History	627
B.1	Deprecated Features	627

B.2	Version 4.5 to 5.0 Differences	627
B.3	Version 4.0 to 4.5 Differences	631
B.4	Version 3.1 to 4.0 Differences	633
B.5	Version 3.0 to 3.1 Differences	634
B.6	Version 2.5 to 3.0 Differences	635
Index		639

List of Figures

2.1	Determining the schedule for a Worksharing-Loop	109
4.1	First-Party Tool Activation Flow Chart	422

List of Tables

1.1	Map-Type Decay of Map Type Combinations	16
2.1	ICV Initial Values	66
2.2	Ways to Modify and to Retrieve ICV Values	68
2.3	Scopes of ICVs	70
2.4	ICV Override Relationships	72
2.5	schedule Clause <i>kind</i> Values	104
2.6	schedule Clause <i>modifier</i> Values	106
2.7	ompt_callback_task_create callback flags evaluation	139
2.8	Predefined Memory Spaces	152
2.9	Allocator Traits	153
2.10	Predefined Allocators	155
2.11	Implicitly Declared C/C++ <i>reduction-identifiers</i>	294
2.12	Implicitly Declared Fortran <i>reduction-identifiers</i>	295
3.1	Standard Tool Control Commands	417
4.1	OMPT Callback Interface Runtime Entry Point Names and Their Type Signatures	426
4.2	Valid Return Codes of ompt_set_callback for Each Callback	428
4.3	OMPT Tracing Interface Runtime Entry Point Names and Their Type Signatures	430
5.1	Mapping of Scope Type and OMPD Handles	542
5.2	OMPD-specific ICVs	589
6.1	Defined Abstract Names for OMP_PLACES	605
6.2	Available Field Types for Formatting OpenMP Thread Affinity Information	613

CHAPTER 1

Introduction

The collection of compiler directives, library routines, and environment variables described in this document collectively define the specification of the OpenMP Application Program Interface (OpenMP API) for parallelism in C, C++ and Fortran programs.

This specification provides a model for parallel programming that is portable across architectures from different vendors. Compilers from numerous vendors support the OpenMP API. More information about the OpenMP API can be found at the following web site

`http://www.openmp.org`

The directives, library routines, environment variables, and tool support defined in this document allow users to create, to manage, to debug and to analyze parallel programs while permitting portability. The directives extend the C, C++ and Fortran base languages with single program multiple data (SPMD) constructs, tasking constructs, device constructs, worksharing constructs, and synchronization constructs, and they provide support for sharing, mapping and privatizing data. The functionality to control the runtime environment is provided by library routines and environment variables. Compilers that support the OpenMP API often include a command line option to the compiler that activates and allows interpretation of all OpenMP directives.

1.1 Scope

The OpenMP API covers only user-directed parallelization, wherein the programmer explicitly specifies the actions to be taken by the compiler and runtime system in order to execute the program in parallel. OpenMP-compliant implementations are not required to check for data dependencies, data conflicts, race conditions, or deadlocks, any of which may occur in conforming programs. In addition, compliant implementations are not required to check for code sequences that cause a program to be classified as non-conforming. Application developers are responsible for correctly

using the OpenMP API to produce a conforming program. The OpenMP API does not cover compiler-generated automatic parallelization.

1.2 Glossary

1.2.1 Threading Concepts

thread An execution entity with a stack and associated static memory, called *threadprivate memory*.

OpenMP thread A *thread* that is managed by the OpenMP implementation.

thread number A number that the OpenMP implementation assigns to an OpenMP thread. For threads within the same team, zero identifies the master thread and consecutive numbers identify the other threads of this team.

idle thread An *OpenMP thread* that is not currently part of any **parallel** region.

thread-safe routine A routine that performs the intended function even when executed concurrently (by more than one *thread*).

processor Implementation-defined hardware unit on which one or more *OpenMP threads* can execute.

device An implementation-defined logical execution engine.

COMMENT: A *device* could have one or more *processors*.

host device The *device* on which the *OpenMP program* begins execution.

target device A device onto which code and data may be offloaded from the *host device*.

parent device For a given **target** region, the device on which the corresponding **target** construct was encountered.

1.2.2 OpenMP Language Terminology

base language A programming language that serves as the foundation of the OpenMP specification.

COMMENT: See Section 1.7 on page 31 for a listing of current *base languages* for the OpenMP API.

1	base program	A program written in a <i>base language</i> .
2	program order	An ordering of operations performed by the same thread as determined by the
3		execution sequence of operations specified by the <i>base language</i> .
4		COMMENT: For C11 and C++11, <i>program order</i> corresponds to the
5		<i>sequenced before</i> relation between operations performed by the same
6		thread.
7	structured block	For C/C++, an executable statement, possibly compound, with a single entry at the
8		top and a single exit at the bottom, or an OpenMP <i>construct</i> .
9		For Fortran, a block of executable statements with a single entry at the top and a
10		single exit at the bottom, or an OpenMP <i>construct</i> .
11		COMMENT: See Section 2.1 on page 38 for restrictions on <i>structured</i>
12		<i>blocks</i> .
13	compilation unit	For C/C++, a translation unit.
14		For Fortran, a program unit.
15	enclosing context	For C/C++, the innermost scope enclosing an OpenMP <i>directive</i> .
16		For Fortran, the innermost scoping unit enclosing an OpenMP <i>directive</i> .
17	directive	For C/C++, a #pragma , and for Fortran, a comment, that specifies <i>OpenMP</i>
18		<i>program</i> behavior.
19		COMMENT: See Section 2.1 on page 38 for a description of OpenMP
20		<i>directive</i> syntax.
21	metadirective	A <i>directive</i> that conditionally resolves to another <i>directive</i> at compile time.
22	white space	A non-empty sequence of space and/or horizontal tab characters.
23	OpenMP program	A program that consists of a <i>base program</i> that is annotated with OpenMP <i>directives</i>
24		or that calls OpenMP API runtime library routines
25	conforming program	An <i>OpenMP program</i> that follows all rules and restrictions of the OpenMP
26		specification.
27	declarative directive	An OpenMP <i>directive</i> that may only be placed in a declarative context. A <i>declarative</i>
28		<i>directive</i> results in one or more declarations only; it is not associated with the
29		immediate execution of any user code.
30	executable directive	An OpenMP <i>directive</i> that is not declarative. That is, it may be placed in an
31		executable context.
32	stand-alone directive	An OpenMP <i>executable directive</i> that has no associated user code except for that
33		which appears in clauses in the directive.

1	construct	An OpenMP <i>executable directive</i> (and for Fortran, the paired end directive , if any) and the associated statement, loop or <i>structured block</i> , if any, not including the code in any called routines. That is, the lexical extent of an <i>executable directive</i> .
4	combined construct	A construct that is a shortcut for specifying one construct immediately nested inside another construct. A combined construct is semantically identical to that of explicitly specifying the first construct containing one instance of the second construct and no other statements.
8	composite construct	A construct that is composed of two constructs but does not have identical semantics to specifying one of the constructs immediately nested inside the other. A composite construct either adds semantics not included in the constructs from which it is composed or the nesting of the one construct inside the other is not conforming.
12	combined target construct	A <i>combined construct</i> that is composed of a target construct along with another construct.
14	region	All code encountered during a specific instance of the execution of a given <i>construct</i> or of an OpenMP library routine. A <i>region</i> includes any code in called routines as well as any implicit code introduced by the OpenMP implementation. The generation of a <i>task</i> at the point where a <i>task generating construct</i> is encountered is a part of the <i>region</i> of the <i>encountering thread</i> . However, an <i>explicit task region</i> corresponding to a <i>task generating construct</i> is not part of the <i>region</i> of the <i>encountering thread</i> unless it is an <i>included task region</i> . The point where a target or teams directive is encountered is a part of the <i>region</i> of the <i>encountering thread</i> , but the <i>region</i> corresponding to the target or teams directive is not.
23		COMMENTS:
24		A <i>region</i> may also be thought of as the dynamic or runtime extent of a <i>construct</i> or of an OpenMP library routine.
26		During the execution of an <i>OpenMP program</i> , a <i>construct</i> may give rise to many <i>regions</i> .
28	active parallel region	A parallel <i>region</i> that is executed by a <i>team</i> consisting of more than one <i>thread</i> .
29	inactive parallel region	A parallel <i>region</i> that is executed by a <i>team</i> of only one <i>thread</i> .
30	active target region	A target <i>region</i> that is executed on a <i>device</i> other than the <i>device</i> that encountered the target <i>construct</i> .
32	inactive target region	A target <i>region</i> that is executed on the same <i>device</i> that encountered the target <i>construct</i> .

1	sequential part	All code encountered during the execution of an <i>initial task region</i> that is not part of
2		a parallel region corresponding to a parallel construct or a task region
3		corresponding to a task construct.
4		COMMENTS:
5		A <i>sequential part</i> is enclosed by an <i>implicit parallel region</i> .
6		Executable statements in called routines may be in both a <i>sequential part</i>
7		and any number of explicit parallel regions at different points in the
8		program execution.
9	master thread	An <i>OpenMP thread</i> that has <i>thread</i> number 0. A <i>master thread</i> may be an <i>initial</i>
10		<i>thread</i> or the <i>thread</i> that encounters a parallel construct, creates a <i>team</i> ,
11		generates a set of <i>implicit tasks</i> , and then executes one of those <i>tasks</i> as <i>thread</i>
12		number 0.
13	parent thread	The <i>thread</i> that encountered the parallel construct and generated a parallel
14		<i>region</i> is the <i>parent thread</i> of each of the <i>threads</i> in the <i>team</i> of that parallel
15		<i>region</i> . The <i>master thread</i> of a parallel region is the same <i>thread</i> as its <i>parent</i>
16		<i>thread</i> with respect to any resources associated with an <i>OpenMP thread</i> .
17	child thread	When a <i>thread</i> encounters a parallel construct, each of the <i>threads</i> in the
18		generated parallel region's <i>team</i> are <i>child threads</i> of the encountering <i>thread</i> .
19		The target or teams region's <i>initial thread</i> is not a <i>child thread</i> of the <i>thread</i> that
20		encountered the target or teams construct.
21	ancestor thread	For a given <i>thread</i> , its <i>parent thread</i> or one of its <i>parent thread's</i> <i>ancestor threads</i> .
22	descendent thread	For a given <i>thread</i> , one of its <i>child threads</i> or one of its <i>child threads'</i> <i>descendent</i>
23		<i>threads</i> .
24	team	A set of one or more <i>threads</i> participating in the execution of a parallel region.
25		COMMENTS:
26		For an <i>active parallel region</i> , the <i>team</i> comprises the <i>master thread</i> and at
27		least one additional <i>thread</i> .
28		For an <i>inactive parallel region</i> , the <i>team</i> comprises only the <i>master thread</i> .
29	league	The set of <i>teams</i> created by a teams construct.
30	contention group	An <i>initial thread</i> and its <i>descendent threads</i> .
31	implicit parallel region	An <i>inactive parallel region</i> that is not generated from a parallel construct.
32		<i>Implicit parallel regions</i> surround the whole <i>OpenMP program</i> , all target regions,
33		and all teams regions.
34	initial thread	The <i>thread</i> that executes an <i>implicit parallel region</i> .

1	initial team	The <i>team</i> that comprises an <i>initial thread</i> executing an <i>implicit parallel region</i> .
2	nested construct	A <i>construct</i> (lexically) enclosed by another <i>construct</i> .
3	closely nested construct	A <i>construct</i> nested inside another <i>construct</i> with no other <i>construct</i> nested between
4		them.
5	nested region	A <i>region</i> (dynamically) enclosed by another <i>region</i> . That is, a <i>region</i> generated from
6		the execution of another <i>region</i> or one of its <i>nested regions</i> .
7		COMMENT: Some nestings are <i>conforming</i> and some are not. See
8		Section 2.20 on page 328 for the restrictions on nesting.
9	closely nested region	A <i>region nested</i> inside another <i>region</i> with no parallel <i>region nested</i> between
10		them.
11	strictly nested region	A <i>region nested</i> inside another <i>region</i> with no other <i>region nested</i> between them.
12	all threads	All OpenMP <i>threads</i> participating in the <i>OpenMP program</i> .
13	current team	All <i>threads</i> in the <i>team</i> executing the innermost enclosing parallel <i>region</i> .
14	encountering thread	For a given <i>region</i> , the <i>thread</i> that encounters the corresponding <i>construct</i> .
15	all tasks	All <i>tasks</i> participating in the <i>OpenMP program</i> .
16	current team tasks	All <i>tasks</i> encountered by the corresponding <i>team</i> . The <i>implicit tasks</i> constituting the
17		parallel <i>region</i> and any <i>descendent tasks</i> encountered during the execution of
18		these <i>implicit tasks</i> are included in this set of tasks.
19	generating task	For a given <i>region</i> , the task for which execution by a <i>thread</i> generated the <i>region</i> .
20	binding thread set	The set of <i>threads</i> that are affected by, or provide the context for, the execution of a
21		<i>region</i> .
22		The <i>binding thread</i> set for a given <i>region</i> can be <i>all threads</i> on a <i>device</i> , <i>all threads</i>
23		in a <i>contention group</i> , <i>all master threads</i> executing an enclosing teams <i>region</i> , the
24		<i>current team</i> , or the <i>encountering thread</i> .
25		COMMENT: The <i>binding thread</i> set for a particular <i>region</i> is described in
26		its corresponding subsection of this specification.
27	binding task set	The set of <i>tasks</i> that are affected by, or provide the context for, the execution of a
28		<i>region</i> .
29		The <i>binding task</i> set for a given <i>region</i> can be <i>all tasks</i> , the <i>current team tasks</i> , <i>all</i>
30		<i>tasks of the current team</i> that are generated in the <i>region</i> , the <i>binding implicit task</i> , or
31		the <i>generating task</i> .
32		COMMENT: The <i>binding task</i> set for a particular <i>region</i> (if applicable) is
33		described in its corresponding subsection of this specification.

1	binding region	The enclosing <i>region</i> that determines the execution context and limits the scope of
2		the effects of the bound <i>region</i> is called the <i>binding region</i> .
3		<i>Binding region</i> is not defined for <i>regions</i> for which the <i>binding thread</i> set is <i>all</i>
4		<i>threads</i> or the <i>encountering thread</i> , nor is it defined for <i>regions</i> for which the <i>binding</i>
5		<i>task set</i> is <i>all tasks</i> .
6		COMMENTS:
7		The <i>binding region</i> for an ordered <i>region</i> is the innermost enclosing
8		<i>loop region</i> .
9		The <i>binding region</i> for a taskwait <i>region</i> is the innermost enclosing
10		<i>task region</i> .
11		The <i>binding region</i> for a cancel <i>region</i> is the innermost enclosing
12		<i>region</i> corresponding to the <i>construct-type-clause</i> of the cancel
13		construct.
14		The <i>binding region</i> for a cancellation point <i>region</i> is the
15		innermost enclosing <i>region</i> corresponding to the <i>construct-type-clause</i> of
16		the cancellation point construct.
17		For all other <i>regions</i> for which the <i>binding thread</i> set is the <i>current team</i>
18		or the <i>binding task set</i> is the <i>current team tasks</i> , the <i>binding region</i> is the
19		innermost enclosing parallel <i>region</i> .
20		For <i>regions</i> for which the <i>binding task set</i> is the <i>generating task</i> , the
21		<i>binding region</i> is the <i>region</i> of the <i>generating task</i> .
22		A parallel <i>region</i> need not be <i>active</i> nor explicit to be a <i>binding</i>
23		<i>region</i> .
24		A <i>task region</i> need not be explicit to be a <i>binding region</i> .
25		A <i>region</i> never binds to any <i>region</i> outside of the innermost enclosing
26		parallel <i>region</i> .
27	orphaned construct	A <i>construct</i> that gives rise to a <i>region</i> for which the <i>binding thread</i> set is the <i>current</i>
28		<i>team</i> , but is not nested within another <i>construct</i> giving rise to the <i>binding region</i> .
29	worksharing construct	A <i>construct</i> that defines units of work, each of which is executed exactly once by one
30		of the <i>threads</i> in the <i>team</i> executing the <i>construct</i> .
31		For C/C++, <i>worksharing constructs</i> are for , sections , and single .
32		For Fortran, <i>worksharing constructs</i> are do , sections , single and
33		workshare .
34	device construct	An OpenMP <i>construct</i> that accepts the device clause.

device routine	A function (for C/C+ and Fortran) or subroutine (for Fortran) that can be executed on a <i>target device</i> , as part of a target region.
place	An unordered set of <i>processors</i> on a device.
place list	The ordered list that describes all OpenMP <i>places</i> available to the execution environment.
place partition	An ordered list that corresponds to a contiguous interval in the OpenMP <i>place list</i> . It describes the <i>places</i> currently available to the execution environment for a given parallel <i>region</i> .
place number	A number that uniquely identifies a <i>place</i> in the <i>place list</i> , with zero identifying the first <i>place</i> in the <i>place list</i> , and each consecutive whole number identifying the next <i>place</i> in the <i>place list</i> .
thread affinity	A binding of <i>threads</i> to <i>places</i> within the current <i>place partition</i> .
SIMD instruction	A single machine instruction that can operate on multiple data elements.
SIMD lane	A software or hardware mechanism capable of processing one data element from a <i>SIMD instruction</i> .
SIMD chunk	A set of iterations executed concurrently, each by a <i>SIMD lane</i> , by a single <i>thread</i> by means of <i>SIMD instructions</i> .
memory	A storage resource to store and to retrieve variables accessible by OpenMP threads.
memory space	A representation of storage resources from which <i>memory</i> can be allocated or deallocated. More than one memory space may exist.
memory allocator	An OpenMP object that fulfills requests to allocate and to deallocate <i>memory</i> for program variables from the storage resources of its associated <i>memory space</i> .
handle	An opaque reference that uniquely identifies an abstraction.

1.2.3 Loop Terminology

loop-associated directive	An OpenMP <i>executable</i> directive for which the associated user code must be a loop nest that is a <i>structured block</i> .
associated loop(s)	The loop(s) controlled by a <i>loop-associated directive</i> . COMMENT: If the <i>loop-associated directive</i> contains a collapse or an ordered (<i>n</i>) clause then it may have more than one <i>associated loop</i> .
sequential loop	A loop that is not associated with any OpenMP <i>loop-associated directive</i> .

SIMD loop	A loop that includes at least one <i>SIMD chunk</i> .
non-rectangular loop nest	A loop nest for which the iteration count of a loop inside the loop nest is the not same for all occurrences of the loop in the loop nest.
doacross loop nest	A loop nest that has cross-iteration dependence. An iteration is dependent on one or more lexicographically earlier iterations.
	COMMENT: The ordered clause parameter on a worksharing-loop directive identifies the loop(s) associated with the <i>doacross loop nest</i> .

1.2.4 Synchronization Terminology

barrier	A point in the execution of a program encountered by a <i>team of threads</i> , beyond which no <i>thread</i> in the team may execute until all <i>threads</i> in the <i>team</i> have reached the barrier and all <i>explicit tasks</i> generated by the <i>team</i> have executed to completion. If <i>cancellation</i> has been requested, threads may proceed to the end of the canceled <i>region</i> even if some threads in the team have not reached the <i>barrier</i> .
cancellation	An action that cancels (that is, aborts) an OpenMP <i>region</i> and causes executing <i>implicit</i> or <i>explicit</i> tasks to proceed to the end of the canceled <i>region</i> .
cancellation point	A point at which implicit and explicit tasks check if cancellation has been requested. If cancellation has been observed, they perform the <i>cancellation</i> .
	COMMENT: For a list of cancellation points, see Section 2.18.1 on page 263.
flush	An operation that a <i>thread</i> performs to enforce consistency between its view and other <i>threads'</i> view of memory.
flush property	Properties that determine the manner in which a <i>flush</i> operation enforces memory consistency. These properties are: <ul style="list-style-type: none"> • <i>strong</i>: flushes a set of variables from the current thread's temporary view of the memory to the memory; • <i>release</i>: orders memory operations that precede the flush before memory operations performed by a different thread with which it synchronizes; • <i>acquire</i>: orders memory operations that follow the flush after memory operations performed by a different thread that synchronizes with it.
	COMMENT: Any <i>flush</i> operation has one or more <i>flush properties</i> .
strong flush	A <i>flush</i> operation that has the <i>strong flush property</i> .

1	release flush	A <i>flush</i> operation that has the <i>release flush property</i> .
2	acquire flush	A <i>flush</i> operation that has the <i>acquire flush property</i> .
3	atomic operation	An operation that is specified by an atomic construct and atomically accesses
4		and/or modifies a specific storage location.
5	atomic read	An <i>atomic operation</i> that is specified by an atomic construct on which the read
6		clause is present.
7	atomic write	An <i>atomic operation</i> that is specified by an atomic construct on which the write
8		clause is present.
9	atomic update	An <i>atomic operation</i> that is specified by an atomic construct on which the
10		update clause is present.
11	atomic captured	An <i>atomic operation</i> that is specified by an atomic construct on which the
12	update	capture clause is present.
13	read-modify-write	An <i>atomic operation</i> that reads and writes to a given storage location.
14		COMMENT: All <i>atomic update</i> and <i>atomic captured update</i> operations
15		are <i>read-modify-write</i> operations.
16	sequentially consistent atomic construct	An atomic construct for which the seq_cst clause is specified.
17	non-sequentially consistent atomic construct	An atomic construct for which the seq_cst clause is not specified
18	sequentially consistent atomic operation	An <i>atomic operation</i> that is specified by a <i>sequentially consistent atomic construct</i> .

19 1.2.5 Tasking Terminology

20	task	A specific instance of executable code and its data environment that the OpenMP
21		implementation can schedule for execution by threads.
22	task region	A <i>region</i> consisting of all code encountered during the execution of a <i>task</i> .
23		COMMENT: A parallel <i>region</i> consists of one or more implicit <i>task</i>
24		<i>regions</i> .
25	implicit task	A <i>task</i> generated by an <i>implicit parallel region</i> or generated when a parallel
26		<i>construct</i> is encountered during execution.

1	binding implicit task	The <i>implicit task</i> of the current thread team assigned to the encountering thread.
2	explicit task	A <i>task</i> that is not an <i>implicit task</i> .
3	initial task	An <i>implicit task</i> associated with an <i>implicit parallel region</i> .
4	current task	For a given <i>thread</i> , the <i>task</i> corresponding to the <i>task region</i> in which it is executing.
5	child task	A <i>task</i> is a <i>child task</i> of its generating <i>task region</i> . A <i>child task region</i> is not part of
6		its generating <i>task region</i> .
7	sibling tasks	<i>Tasks</i> that are <i>child tasks</i> of the same <i>task region</i> .
8	descendent task	A <i>task</i> that is the <i>child task</i> of a <i>task region</i> or of one of its <i>descendent task regions</i> .
9	task completion	<i>Task completion</i> occurs when the end of the <i>structured block</i> associated with the
10		<i>construct</i> that generated the <i>task</i> is reached.
11		COMMENT: Completion of the <i>initial task</i> that is generated when the
12		program begins occurs at program exit.
13	task scheduling point	A point during the execution of the current <i>task region</i> at which it can be suspended
14		to be resumed later; or the point of <i>task completion</i> , after which the executing thread
15		may switch to a different <i>task region</i> .
16		COMMENT: For a list of <i>task scheduling points</i> , see Section 2.10.6 on
17		page 149.
18	task switching	The act of a <i>thread</i> switching from the execution of one <i>task</i> to another <i>task</i> .
19	tied task	A <i>task</i> that, when its <i>task region</i> is suspended, can be resumed only by the same
20		<i>thread</i> that suspended it. That is, the <i>task</i> is tied to that <i>thread</i> .
21	untied task	A <i>task</i> that, when its <i>task region</i> is suspended, can be resumed by any <i>thread</i> in the
22		team. That is, the <i>task</i> is not tied to any <i>thread</i> .
23	unddeferred task	A <i>task</i> for which execution is not deferred with respect to its generating <i>task region</i> .
24		That is, its generating <i>task region</i> is suspended until execution of the structured block
25		associated with the <i>unddeferred task</i> is completed.
26	included task	A <i>task</i> for which execution is sequentially included in the generating <i>task region</i> .
27		That is, an <i>included task</i> is <i>unddeferred</i> and executed by the <i>encountering thread</i> .
28	merged task	A <i>task</i> for which the <i>data environment</i> , inclusive of ICVs, is the same as that of its
29		generating <i>task region</i> .
30	mergeable task	A <i>task</i> that may be a <i>merged task</i> if it is an <i>unddeferred task</i> or an <i>included task</i> .
31	final task	A <i>task</i> that forces all of its <i>child tasks</i> to become <i>final</i> and <i>included tasks</i> .

1	task dependence	An ordering relation between two <i>sibling tasks</i> : the <i>dependent task</i> and a previously
2		generated <i>predecessor task</i> . The <i>task dependence</i> is fulfilled when the <i>predecessor</i>
3		<i>task</i> has completed.
4	dependent task	A <i>task</i> that because of a <i>task dependence</i> cannot be executed until its <i>predecessor</i>
5		<i>tasks</i> have completed.
6	mutually exclusive tasks	<i>Tasks</i> that may be executed in any order, but not at the same time.
7	predecessor task	A <i>task</i> that must complete before its <i>dependent tasks</i> can be executed.
8	task synchronization construct	A taskwait , taskgroup , or a barrier <i>construct</i> .
9	task generating construct	A <i>construct</i> that generates one or more <i>explicit tasks</i> .
10	target task	A <i>mergeable</i> and <i>untied task</i> that is generated by a target , target enter
11		data , target exit data , or target update <i>construct</i> .
12	taskgroup set	A set of tasks that are logically grouped by a taskgroup <i>region</i> .

13 1.2.6 Data Terminology

14	variable	A named data storage block, for which the value can be defined and redefined during
15		the execution of a program.
16		COMMENT: An array element or structure element is a variable that is
17		part of another variable.
18	scalar variable	For C/C++, a scalar variable, as defined by the base language.
19		For Fortran, a scalar variable with intrinsic type, as defined by the base language,
20		excluding character type.
21	aggregate variable	A variable, such as an array or structure, composed of other variables.
22	array section	A designated subset of the elements of an array that is specified using a subscript
23		notation that can select more than one element.
24	array item	An array, an array section, or an array element.
25	shape-operator	For C/C++, an array shaping operator that reinterprets a pointer expression as an
26		array with one or more specified dimensions.

1	implicit array	For C/C++, the set of array elements of non-array type <i>T</i> that may be accessed by
2		applying a sequence of [] operators to a given pointer that is either a pointer to type <i>T</i>
3		or a pointer to a multidimensional array of elements of type <i>T</i> .
4		For Fortran, the set of array elements for a given array pointer.
5		COMMENT: For C/C++, the implicit array for pointer <i>p</i> with type <i>T</i>
6		(*)[10] consists of all accessible elements <i>p</i> [<i>i</i>][<i>j</i>], for all <i>i</i> and <i>j</i> =0..9.
7	base pointer	For C/C++, an lvalue pointer expression that is used by a given lvalue expression or
8		array section to refer indirectly to its storage, where the lvalue expression or array
9		section is part of the implicit array for that lvalue pointer expression.
10		For Fortran, a data pointer that appears last in the designator for a given variable or
11		array section, where the variable or array section is part of the pointer target for that
12		data pointer.
13		COMMENT: For the array section
14		(* <i>p0</i>). <i>x0</i> [<i>k1</i>]. <i>p1</i> -> <i>p2</i> [<i>k2</i>]. <i>x1</i> [<i>k3</i>]. <i>x2</i> [4][0: <i>n</i>], where identifiers <i>pi</i> have a
15		pointer type declaration and identifiers <i>xi</i> have an array type declaration,
16		the <i>base pointer</i> is: (* <i>p0</i>). <i>x0</i> [<i>k1</i>]. <i>p1</i> -> <i>p2</i> .
17	named pointer	For C/C++, the <i>base pointer</i> of a given lvalue expression or array section, or the <i>base</i>
18		<i>pointer</i> of one of its <i>named pointers</i> .
19		For Fortran, the <i>base pointer</i> of a given variable or array section, or the <i>base pointer</i>
20		of one of its <i>named pointers</i> .
21		COMMENT: For the array section
22		(* <i>p0</i>). <i>x0</i> [<i>k1</i>]. <i>p1</i> -> <i>p2</i> [<i>k2</i>]. <i>x1</i> [<i>k3</i>]. <i>x2</i> [4][0: <i>n</i>], where identifiers <i>pi</i> have a
23		pointer type declaration and identifiers <i>xi</i> have an array type declaration,
24		the <i>named pointers</i> are: <i>p0</i> , (* <i>p0</i>). <i>x0</i> [<i>k1</i>]. <i>p1</i> , and (* <i>p0</i>). <i>x0</i> [<i>k1</i>]. <i>p1</i> -> <i>p2</i> .
25	containing array	For C/C++, a non-subscripted array (a <i>containing array</i>) that appears in a given
26		lvalue expression or array section, where the lvalue expression or array section is part
27		of that <i>containing array</i> .
28		For Fortran, an array (a <i>containing array</i>) without the POINTER attribute and
29		without a subscript list that appears in the designator of a given variable or array
30		section, where the variable or array section is part of that <i>containing array</i> .
31		COMMENT: For the array section
32		(* <i>p0</i>). <i>x0</i> [<i>k1</i>]. <i>p1</i> -> <i>p2</i> [<i>k2</i>]. <i>x1</i> [<i>k3</i>]. <i>x2</i> [4][0: <i>n</i>], where identifiers <i>pi</i> have a
33		pointer type declaration and identifiers <i>xi</i> have an array type declaration,
34		the <i>containing arrays</i> are: (* <i>p0</i>). <i>x0</i> [<i>k1</i>]. <i>p1</i> -> <i>p2</i> [<i>k2</i>]. <i>x1</i> and
35		(* <i>p0</i>). <i>x0</i> [<i>k1</i>]. <i>p1</i> -> <i>p2</i> [<i>k2</i>]. <i>x1</i> [<i>k3</i>]. <i>x2</i> .

base array For C/C++, a *containing array* of a given lvalue expression or array section that does not appear in the expression of any of its other *containing arrays*.

For Fortran, a *containing array* of a given variable or array section that does not appear in the designator of any of its other *containing arrays*.

COMMENT: For the array section
 (*p0).x0[k1].p1->p2[k2].x1[k3].x2[4][0:n], where identifiers *pi* have a
 pointer type declaration and identifiers *xi* have an array type declaration,
 the *base array* is: (*p0).x0[k1].p1->p2[k2].x1[k3].x2.

named array For C/C++, a *containing array* of a given lvalue expression or array section, or a *containing array* of one of its *named pointers*.

For Fortran, a *containing array* of a given variable or array section, or a *containing array* of one of its *named pointers*.

COMMENT: For the array section
 (*p0).x0[k1].p1->p2[k2].x1[k3].x2[4][0:n], where identifiers *pi* have a
 pointer type declaration and identifiers *xi* have an array type declaration,
 the *named arrays* are: (*p0).x0, (*p0).x0[k1].p1->p2[k2].x1, and
 (*p0).x0[k1].p1->p2[k2].x1[k3].x2.

base expression The *base array* of a given array section or array element, if it exists; otherwise, the *base pointer* of the array section or array element.

COMMENT: For the array section
 (*p0).x0[k1].p1->p2[k2].x1[k3].x2[4][0:n], where identifiers *pi* have a
 pointer type declaration and identifiers *xi* have an array type declaration,
 the *base expression* is: (*p0).x0[k1].p1->p2[k2].x1[k3].x2.

More examples for C/C++:

- The *base expression* for *x[i]* and for *x[i:n]* is *x*, if *x* is an array or pointer.
- The *base expression* for *x[5][i]* and for *x[5][i:n]* is *x*, if *x* is a pointer to an array or *x* is 2-dimensional array.
- The *base expression* for *y[5][i]* and for *y[5][i:n]* is *y[5]*, if *y* is an array of pointers or *y* is a pointer to a pointer.

Examples for Fortran:

- The *base expression* for *x(i)* and for *x(i:j)* is *x*.

attached pointer A pointer variable in a device data environment to which the effect of a **map** clause assigns the address of an object, minus some offset, that is created in the device data environment. The pointer is an attached pointer for the remainder of its lifetime in the device data environment.

1	simply contiguous	An array section that statically can be determined to have contiguous storage or that,
2	array section	in Fortran, has the CONTIGUOUS attribute.
3	structure	A structure is a variable that contains one or more variables.
4		For C/C++: Implemented using struct types.
5		For C++: Implemented using class types.
6		For Fortran: Implemented using derived types.
7	private variable	With respect to a given set of <i>task regions</i> or <i>SIMD lanes</i> that bind to the same
8		parallel region , a <i>variable</i> for which the name provides access to a different
9		block of storage for each <i>task region</i> or <i>SIMD lane</i> .
10		A <i>variable</i> that is part of another variable (as an array or structure element) cannot be
11		made private independently of other components.
12	shared variable	With respect to a given set of <i>task regions</i> that bind to the same parallel region , a
13		<i>variable</i> for which the name provides access to the same block of storage for each
14		<i>task region</i> .
15		A <i>variable</i> that is part of another variable (as an array or structure element) cannot be
16		<i>shared</i> independently of the other components, except for static data members of
17		C++ classes.
18	threadprivate variable	A <i>variable</i> that is replicated, one instance per <i>thread</i> , by the OpenMP
19		implementation. Its name then provides access to a different block of storage for each
20		<i>thread</i> .
21		A <i>variable</i> that is part of another variable (as an array or structure element) cannot be
22		made <i>threadprivate</i> independently of the other components, except for static data
23		members of C++ classes.
24	threadprivate memory	The set of <i>threadprivate variables</i> associated with each <i>thread</i> .
25	data environment	The <i>variables</i> associated with the execution of a given <i>region</i> .
26	device data environment	The initial <i>data environment</i> associated with a device.
27	device address	An <i>implementation-defined</i> reference to an address in a <i>device data environment</i> .
28	device pointer	A <i>variable</i> that contains a <i>device address</i> .
29	mapped variable	An original <i>variable</i> in a <i>data environment</i> with a corresponding <i>variable</i> in a device
30		<i>data environment</i> .
31		COMMENT: The original and corresponding <i>variables</i> may share storage.

TABLE 1.1: Map-Type Decay of Map Type Combinations

	alloc	to	from	tofrom	release	delete
alloc	alloc	alloc	alloc	alloc	release	delete
to	alloc	to	alloc	to	release	delete
from	alloc	alloc	from	from	release	delete
tofrom	alloc	to	from	tofrom	release	delete

map-type decay The process used to determine the final map type when mapping a variable with a user defined mapper. Table 1.1 shows the final map type that the combination of the two map types determines.

mappable type A type that is valid for a *mapped variable*. If a type is composed from other types (such as the type of an array or structure element) and any of the other types are not mappable then the type is not mappable.

COMMENT: Pointer types are *mappable* but the memory block to which the pointer refers is not *mapped*.

For C, the type must be a complete type.

For C++, the type must be a complete type.

In addition, for class types:

- All member functions accessed in any **target** region must appear in a **declare target** directive.

For Fortran, no restrictions on the type except that for derived types:

- All type-bound procedures accessed in any target region must appear in a **declare target** directive.

defined For *variables*, the property of having a valid value.

For C, for the contents of *variables*, the property of having a valid value.

For C++, for the contents of *variables* of POD (plain old data) type, the property of having a valid value.

For *variables* of non-POD class type, the property of having been constructed but not subsequently destructed.

For Fortran, for the contents of *variables*, the property of having a valid value. For the allocation or association status of *variables*, the property of having a valid status.

COMMENT: Programs that rely upon *variables* that are not *defined* are *non-conforming programs*.

class type For C++, *variables* declared with one of the **class**, **struct**, or **union** keywords.

1 1.2.7 Implementation Terminology

2	supporting n active	Implies allowing an <i>active parallel region</i> to be enclosed by $n-1$ <i>active parallel</i>
3	levels of parallelism	<i>regions</i> .
4	supporting the	Supporting at least one active level of parallelism.
	OpenMP API	
5	supporting nested	Supporting more than one active level of parallelism.
	parallelism	
6	internal control	A conceptual variable that specifies runtime behavior of a set of <i>threads</i> or <i>tasks</i> in
7	variable	an <i>OpenMP program</i> .
8		COMMENT: The acronym ICV is used interchangeably with the term
9		<i>internal control variable</i> in the remainder of this specification.
10	compliant	An implementation of the OpenMP specification that compiles and executes any
11	implementation	<i>conforming program</i> as defined by the specification.
12		COMMENT: A <i>compliant implementation</i> may exhibit <i>unspecified</i>
13		<i>behavior</i> when compiling or executing a <i>non-conforming program</i> .
14	unspecified behavior	A behavior or result that is not specified by the OpenMP specification or not known
15		prior to the compilation or execution of an <i>OpenMP program</i> .
16		Such <i>unspecified behavior</i> may result from:
17		• Issues documented by the OpenMP specification as having <i>unspecified behavior</i> .
18		• A <i>non-conforming program</i> .
19		• A <i>conforming program</i> exhibiting an <i>implementation-defined</i> behavior.
20	implementation defined	Behavior that must be documented by the implementation, and is allowed to vary
21		among different <i>compliant implementations</i> . An implementation is allowed to define
22		this behavior as <i>unspecified</i> .
23		COMMENT: All features that have <i>implementation-defined</i> behavior are
24		documented in Appendix A.
25	deprecated	For a construct, clause, or other feature, the property that it is normative in the
26		current specification but is considered obsolescent and will be removed in the future.

27 1.2.8 Tool Terminology

28	tool	Executable code, distinct from application or runtime code, that can observe and/or
29		modify the execution of an application.

1	first-party tool	A tool that executes in the address space of the program that it is monitoring.
2	third-party tool	A tool that executes as a separate process from the process that it is monitoring and
3		potentially controlling.
4	activated tool	A <i>first-party tool</i> that successfully completed its initialization.
5	event	A point of interest in the execution of a thread.
6	native thread	A thread defined by an underlying thread implementation.
7	tool callback	A function that a tool provides to an OpenMP implementation to invoke when an
8		associated event occurs.
9	registering a callback	Providing a <i>tool callback</i> to an OpenMP implementation.
10	dispatching a callback	Processing a callback when an associated <i>event</i> occurs in a manner consistent with
11	at an event	the return code provided when a <i>first-party tool</i> registered the callback.
12	thread state	An enumeration type that describes the current OpenMP activity of a <i>thread</i> . A
13		<i>thread</i> can be in only one state at any time.
14	wait identifier	A unique opaque handle associated with each data object (for example, a lock) used
15		by the OpenMP runtime to enforce mutual exclusion that may cause a thread to wait
16		actively or passively.
17	frame	A storage area on a thread's stack associated with a procedure invocation. A frame
18		includes space for one or more saved registers and often also includes space for saved
19		arguments, local variables, and padding for alignment.
20	canonical frame	An address associated with a procedure <i>frame</i> on a call stack that was the value of the
21	address	stack pointer immediately prior to calling the procedure for which the invocation is
22		represented by the frame.
23	runtime entry point	A function interface provided by an OpenMP runtime for use by a tool. A runtime
24		entry point is typically not associated with a global function symbol.
25	trace record	A data structure in which to store information associated with an occurrence of an
26		<i>event</i> .
27	native trace record	A <i>trace record</i> for an OpenMP device that is in a device-specific format.
28	signal	A software interrupt delivered to a <i>thread</i> .
29	signal handler	A function called asynchronously when a <i>signal</i> is delivered to a <i>thread</i> .
30	async signal safe	The guarantee that interruption by <i>signal</i> delivery will not interfere with a set of
31		operations. An async signal safe <i>runtime entry point</i> is safe to call from a <i>signal</i>
32		<i>handler</i> .

1	code block	A contiguous region of memory that contains code of an OpenMP program to be
2		executed on a device.
3	OMPT	An interface that helps a <i>first-party tool</i> monitor the execution of an OpenMP
4		program.
5	OMPT interface state	A state that indicates the permitted interactions between a first-party tool and the
6		OpenMP implementation.
7	OMPT active	An <i>OMPT interface state</i> in which the OpenMP implementation is prepared to accept
8		runtime calls from a <i>first party tool</i> and it dispatches any registered callbacks and in
9		which a first-party tool can invoke <i>runtime entry points</i> if not otherwise restricted.
10	OMPT pending	An <i>OMPT interface state</i> in which the OpenMP implementation can only call
11		functions to initialize a <i>first party tool</i> and in which a <i>first-party tool</i> cannot invoke
12		<i>runtime entry points</i> .
13	OMPT inactive	An <i>OMPT interface state</i> in which the OpenMP implementation will not make any
14		callbacks and in which a <i>first-party tool</i> cannot invoke <i>runtime entry points</i> .
15	OMPD	An interface that helps a <i>third-party tool</i> inspect the OpenMP state of a program that
16		has begun execution.
17	OMPD library	A dynamically loadable library that implements the <i>OMPD</i> interface.
18	image file	An executable or shared library.
19	address space	A collection of logical, virtual, or physical memory address ranges that contain code,
20		stack, and/or data. Address ranges within an address space need not be contiguous.
21		An address space consists of one or more <i>segments</i> .
22	segment	A portion of an address space associated with a set of address ranges.
23	OpenMP architecture	The architecture on which an OpenMP <i>region</i> executes.
24	tool architecture	The architecture on which an <i>OMPD</i> tool executes.
25	OpenMP process	A collection of one or more <i>threads</i> and <i>address spaces</i> . A process may contain
26		<i>threads</i> and <i>address spaces</i> for multiple <i>OpenMP architectures</i> . At least one thread
27		in an OpenMP process is an OpenMP <i>thread</i> . A process may be live or a core file.
28	address space handle	A <i>handle</i> that refers to an <i>address space</i> within an OpenMP process.
29	thread handle	A <i>handle</i> that refers to an OpenMP <i>thread</i> .
30	parallel handle	A <i>handle</i> that refers to an OpenMP parallel <i>region</i> .
31	task handle	A <i>handle</i> that refers to an OpenMP task <i>region</i> .
32	descendent handle	An output <i>handle</i> that is returned from the <i>OMPD</i> library in a function that accepts
33		an input <i>handle</i> : the output <i>handle</i> is a descendent of the input <i>handle</i> .

ancestor handle An input *handle* that is passed to the *OMPD* library in a function that returns an output *handle*: the input *handle* is an ancestor of the output *handle*. For a given *handle*, the ancestors of the *handle* are also the ancestors of the handle's descendent.

COMMENT: A *handle* cannot be used by the tool in an *OMPD* call if any ancestor of the *handle* has been released, except for *OMPD* calls that release the *handle*.

tool context An opaque reference provided by a tool to an *OMPD* library. A *tool context* uniquely identifies an abstraction.

address space context A *tool context* that refers to an *address space* within a process.

thread context A *tool context* that refers to a *native thread*.

native thread identifier An identifier for a native thread defined by a thread implementation.

1.3 Execution Model

The OpenMP API uses the fork-join model of parallel execution. Multiple threads of execution perform tasks defined implicitly or explicitly by OpenMP directives. The OpenMP API is intended to support programs that will execute correctly both as parallel programs (multiple threads of execution and a full OpenMP support library) and as sequential programs (directives ignored and a simple OpenMP stubs library). However, it is possible and permitted to develop a program that executes correctly as a parallel program but not as a sequential program, or that produces different results when executed as a parallel program compared to when it is executed as a sequential program. Furthermore, using different numbers of threads may result in different numeric results because of changes in the association of numeric operations. For example, a serial addition reduction may have a different pattern of addition associations than a parallel reduction. These different associations may change the results of floating-point addition.

An OpenMP program begins as a single thread of execution, called an initial thread. An initial thread executes sequentially, as if the code encountered is part of an implicit task region, called an initial task region, that is generated by the implicit parallel region surrounding the whole program.

The thread that executes the implicit parallel region that surrounds the whole program executes on the *host device*. An implementation may support other *target devices*. If supported, one or more devices are available to the host device for offloading code and data. Each device has its own threads that are distinct from threads that execute on another device. Threads cannot migrate from one device to another device. The execution model is host-centric such that the host device offloads **target** regions to target devices.

When a **target** construct is encountered, a new *target task* is generated. The *target task* region encloses the **target** region. The *target task* is complete after the execution of the **target** region is complete.

When a *target task* executes, the enclosed **target** region is executed by an initial thread. The initial thread may execute on a *target device*. The initial thread executes sequentially, as if the target region is part of an initial task region that is generated by an implicit parallel region. If the target device does not exist or the implementation does not support the target device, all **target** regions associated with that device execute on the host device.

The implementation must ensure that the **target** region executes as if it were executed in the data environment of the target device unless an **if** clause is present and the **if** clause expression evaluates to *false*.

The **teams** construct creates a *league of teams*, where each team is an initial team that comprises an initial thread that executes the **teams** region. Each initial thread executes sequentially, as if the code encountered is part of an initial task region that is generated by an implicit parallel region associated with each team.

If a construct creates a data environment, the data environment is created at the time the construct is encountered. The description of a construct defines whether it creates a data environment.

When any thread encounters a **parallel** construct, the thread creates a team of itself and zero or more additional threads and becomes the master of the new team. A set of implicit tasks, one per thread, is generated. The code for each task is defined by the code inside the **parallel** construct. Each task is assigned to a different thread in the team and becomes tied; that is, it is always executed by the thread to which it is initially assigned. The task region of the task being executed by the encountering thread is suspended, and each member of the new team executes its implicit task. There is an implicit barrier at the end of the **parallel** construct. Only the master thread resumes execution beyond the end of the **parallel** construct, resuming the task region that was suspended upon encountering the **parallel** construct. Any number of **parallel** constructs can be specified in a single program.

parallel regions may be arbitrarily nested inside each other. If nested parallelism is disabled, or is not supported by the OpenMP implementation, then the new team that is created by a thread encountering a **parallel** construct inside a **parallel** region will consist only of the encountering thread. However, if nested parallelism is supported and enabled, then the new team can consist of more than one thread. A **parallel** construct may include a **proc_bind** clause to specify the places to use for the threads in the team within the **parallel** region.

When any team encounters a worksharing construct, the work inside the construct is divided among the members of the team, and executed cooperatively instead of being executed by every thread. There is a default barrier at the end of each worksharing construct unless the **nowait** clause is present. Redundant execution of code by every thread in the team resumes after the end of the worksharing construct.

When any thread encounters a *task generating construct*, one or more explicit tasks are generated. Execution of explicitly generated tasks is assigned to one of the threads in the current team, subject to the thread's availability to execute work. Thus, execution of the new task could be immediate, or deferred until later according to task scheduling constraints and thread availability. Threads are allowed to suspend the current task region at a task scheduling point in order to execute a different task. If the suspended task region is for a tied task, the initially assigned thread later resumes execution of the suspended task region. If the suspended task region is for an untied task, then any thread may resume its execution. Completion of all explicit tasks bound to a given parallel region is guaranteed before the master thread leaves the implicit barrier at the end of the region. Completion of a subset of all explicit tasks bound to a given parallel region may be specified through the use of task synchronization constructs. Completion of all explicit tasks bound to the implicit parallel region is guaranteed by the time the program exits.

When any thread encounters a **simd** construct, the iterations of the loop associated with the construct may be executed concurrently using the SIMD lanes that are available to the thread.

When a **loop** construct is encountered, the iterations of the loop associated with the construct are executed in the context of its encountering thread(s), as determined according to its binding region. If the **loop** region binds to a **teams** region, the region is encountered by the set of master threads that execute the **teams** region. If the **loop** region binds to a **parallel** region, the region is encountered by the team of threads executing the **parallel** region. Otherwise, the region is encountered by a single thread.

If the **loop** region binds to a **teams** region, the encountering threads may continue execution after the **loop** region without waiting for all iterations to complete; the iterations are guaranteed to complete before the end of the **teams** region. Otherwise, all iterations must complete before the encountering thread(s) continue execution after the **loop** region. All threads that encounter the **loop** construct may participate in the execution of the iterations. Only one of these threads may execute any given iteration.

The **cancel** construct can alter the previously described flow of execution in an OpenMP region. The effect of the **cancel** construct depends on its *construct-type-clause*. If a task encounters a **cancel** construct with a **taskgroup** *construct-type-clause*, then the task activates cancellation and continues execution at the end of its **task** region, which implies completion of that task. Any other task in that **taskgroup** that has begun executing completes execution unless it encounters a **cancellation point** construct, in which case it continues execution at the end of its **task** region, which implies its completion. Other tasks in that **taskgroup** region that have not begun execution are aborted, which implies their completion.

For all other *construct-type-clause* values, if a thread encounters a **cancel** construct, it activates cancellation of the innermost enclosing region of the type specified and the thread continues execution at the end of that region. Threads check if cancellation has been activated for their region at cancellation points and, if so, also resume execution at the end of the canceled region.

If cancellation has been activated regardless of *construct-type-clause*, threads that are waiting inside a barrier other than an implicit barrier at the end of the canceled region exit the barrier and

resume execution at the end of the canceled region. This action can occur before the other threads reach that barrier.

Synchronization constructs and library routines are available in the OpenMP API to coordinate tasks and data access in **parallel** regions. In addition, library routines and environment variables are available to control or to query the runtime environment of OpenMP programs.

The OpenMP specification makes no guarantee that input or output to the same file is synchronous when executed in parallel. In this case, the programmer is responsible for synchronizing input and output processing with the assistance of OpenMP synchronization constructs or library routines. For the case where each thread accesses a different file, no synchronization by the programmer is necessary.

1.4 Memory Model

1.4.1 Structure of the OpenMP Memory Model

The OpenMP API provides a relaxed-consistency, shared-memory model. All OpenMP threads have access to a place to store and to retrieve variables, called the *memory*. In addition, each thread is allowed to have its own *temporary view* of the memory. The temporary view of memory for each thread is not a required part of the OpenMP memory model, but can represent any kind of intervening structure, such as machine registers, cache, or other local storage, between the thread and the memory. The temporary view of memory allows the thread to cache variables and thereby to avoid going to memory for every reference to a variable. Each thread also has access to another type of memory that must not be accessed by other threads, called *threadprivate memory*.

A directive that accepts data-sharing attribute clauses determines two kinds of access to variables used in the directive's associated structured block: shared and private. Each variable referenced in the structured block has an original variable, which is the variable by the same name that exists in the program immediately outside the construct. Each reference to a shared variable in the structured block becomes a reference to the original variable. For each private variable referenced in the structured block, a new version of the original variable (of the same type and size) is created in memory for each task or SIMD lane that contains code associated with the directive. Creation of the new version does not alter the value of the original variable. However, the impact of attempts to access the original variable during the region corresponding to the directive is unspecified; see Section 2.19.4.3 on page 285 for additional details. References to a private variable in the structured block refer to the private version of the original variable for the current task or SIMD lane. The relationship between the value of the original variable and the initial or final value of the private version depends on the exact clause that specifies it. Details of this issue, as well as other issues with privatization, are provided in Section 2.19 on page 269.

The minimum size at which a memory update may also read and write back adjacent variables that are part of another variable (as array or structure elements) is implementation defined but is no larger than required by the base language.

A single access to a variable may be implemented with multiple load or store instructions and, thus, is not guaranteed to be atomic with respect to other accesses to the same variable. Accesses to variables smaller than the implementation defined minimum size or to C or C++ bit-fields may be implemented by reading, modifying, and rewriting a larger unit of memory, and may thus interfere with updates of variables or fields in the same unit of memory.

If multiple threads write without synchronization to the same memory unit, including cases due to atomicity considerations as described above, then a data race occurs. Similarly, if at least one thread reads from a memory unit and at least one thread writes without synchronization to that same memory unit, including cases due to atomicity considerations as described above, then a data race occurs. If a data race occurs then the result of the program is unspecified.

A private variable in a task region that subsequently generates an inner nested **parallel** region is permitted to be made shared by implicit tasks in the inner **parallel** region. A private variable in a task region can also be shared by an explicit task region generated during its execution. However, it is the programmer's responsibility to ensure through synchronization that the lifetime of the variable does not end before completion of the explicit task region sharing it. Any other access by one task to the private variables of another task results in unspecified behavior.

1.4.2 Device Data Environments

When an OpenMP program begins, an implicit **target data** region for each device surrounds the whole program. Each device has a device data environment that is defined by its implicit **target data** region. Any **declare target** directives and the directives that accept data-mapping attribute clauses determine how an original variable in a data environment is mapped to a corresponding variable in a device data environment.

When an original variable is mapped to a device data environment and a corresponding variable is not present in the device data environment, a new corresponding variable (of the same type and size as the original variable) is created in the device data environment. Conversely, the original variable becomes the new variable's corresponding variable in the device data environment of the device that performs the mapping operation.

The corresponding variable in the device data environment may share storage with the original variable. Writes to the corresponding variable may alter the value of the original variable. The impact of this possibility on memory consistency is discussed in Section 1.4.6 on page 28. When a task executes in the context of a device data environment, references to the original variable refer to the corresponding variable in the device data environment. If an original variable is not currently mapped and a corresponding variable does not exist in the device data environment then accesses to

the original variable result in unspecified behavior unless the **unified_shared_memory** clause is specified on a **requires** directive for the compilation unit.

The relationship between the value of the original variable and the initial or final value of the corresponding variable depends on the *map-type*. Details of this issue, as well as other issues with mapping a variable, are provided in Section 2.19.7.1 on page 315.

The original variable in a data environment and the corresponding variable(s) in one or more device data environments may share storage. Without intervening synchronization data races can occur.

1.4.3 Memory Management

The host device, and target devices that an implementation may support, have attached storage resources where program variables are stored. These resources can have different traits. A memory space in an OpenMP program represents a set of these storage resources. Memory spaces are defined according to a set of traits, and a single resource may be exposed as multiple memory spaces with different traits or may be part of multiple memory spaces. In any device, at least one memory space is guaranteed to exist.

An OpenMP program can use a *memory allocator* to allocate *memory* in which to store variables. This *memory* will be allocated from the storage resources of the *memory space* associated with the memory allocator. Memory allocators are also used to deallocate previously allocated *memory*. When an OpenMP memory allocator is not used to allocate memory, OpenMP does not prescribe the storage resource for the allocation; the memory for the variables may be allocated in any storage resource.

1.4.4 The Flush Operation

The memory model has relaxed-consistency because a thread's temporary view of memory is not required to be consistent with memory at all times. A value written to a variable can remain in the thread's temporary view until it is forced to memory at a later time. Likewise, a read from a variable may retrieve the value from the thread's temporary view, unless it is forced to read from memory. OpenMP flush operations are used to enforce consistency between a thread's temporary view of memory and memory, or between multiple threads' view of memory.

If a flush operation is a strong flush, it enforces consistency between a thread's temporary view and memory. A strong flush operation is applied to a set of variables called the *flush-set*. A strong flush restricts reordering of memory operations that an implementation might otherwise do.

Implementations must not reorder the code for a memory operation for a given variable, or the code

for a flush operation for the variable, with respect to a strong flush operation that refers to the same variable.

If a thread has performed a write to its temporary view of a shared variable since its last strong flush of that variable, then when it executes another strong flush of the variable, the strong flush does not complete until the value of the variable has been written to the variable in memory. If a thread performs multiple writes to the same variable between two strong flushes of that variable, the strong flush ensures that the value of the last write is written to the variable in memory. A strong flush of a variable executed by a thread also causes its temporary view of the variable to be discarded, so that if its next memory operation for that variable is a read, then the thread will read from memory and capture the value in its temporary view. When a thread executes a strong flush, no later memory operation by that thread for a variable involved in that strong flush is allowed to start until the strong flush completes. The completion of a strong flush executed by a thread is defined as the point at which all writes to the flush-set performed by the thread before the strong flush are visible in memory to all other threads, and at which that thread's temporary view of the flush-set is discarded.

A strong flush operation provides a guarantee of consistency between a thread's temporary view and memory. Therefore, a strong flush can be used to guarantee that a value written to a variable by one thread may be read by a second thread. To accomplish this, the programmer must ensure that the second thread has not written to the variable since its last strong flush of the variable, and that the following sequence of events are completed in this specific order:

1. The value is written to the variable by the first thread;
2. The variable is flushed, with a strong flush, by the first thread;
3. The variable is flushed, with a strong flush, by the second thread; and
4. The value is read from the variable by the second thread.

If a flush operation is a release flush or acquire flush, it can enforce consistency between the views of memory of two synchronizing threads. A release flush guarantees that any prior operation that writes or reads a shared variable will appear to be completed before any operation that writes or reads the same shared variable and follows an acquire flush with which the release flush synchronizes (see Section 1.4.5 on page 27 for more details on flush synchronization). A release flush will propagate the values of all shared variables in its temporary view to memory prior to the thread performing any subsequent atomic operation that may establish a synchronization. An acquire flush will discard any value of a shared variable in its temporary view to which the thread has not written since last performing a release flush, so that it may subsequently read a value propagated by a release flush that synchronizes with it. Therefore, release and acquire flushes may also be used to guarantee that a value written to a variable by one thread may be read by a second thread. To accomplish this, the programmer must ensure that the second thread has not written to the variable since its last acquire flush, and that the following sequence of events happen in this specific order:

1. The value is written to the variable by the first thread;
2. The first thread performs a release flush;

3. The second thread performs an acquire flush; and
4. The value is read from the variable by the second thread.

Note – OpenMP synchronization operations, described in Section 2.17 on page 223 and in Section 3.3 on page 381, are recommended for enforcing this order. Synchronization through variables is possible but is not recommended because the proper timing of flushes is difficult.

The flush properties that define whether a flush operation is a strong flush, a release flush, or an acquire flush are not mutually disjoint. A flush operation may be a strong flush and a release flush; it may be a strong flush and an acquire flush; it may be a release flush and an acquire flush; or it may be all three.

1.4.5 Flush Synchronization and *Happens Before*

OpenMP supports thread synchronization with the use of release flushes and acquire flushes. For any such synchronization, a release flush is the source of the synchronization and an acquire flush is the sink of the synchronization, such that the release flush *synchronizes with* the acquire flush.

A release flush has one or more associated *release sequences* that define the set of modifications that may be used to establish a synchronization. A release sequence starts with an atomic operation that follows the release flush and modifies a shared variable and additionally includes any read-modify-write atomic operations that read a value taken from some modification in the release sequence. The following rules determine the atomic operation that starts an associated release sequence.

- If a release flush is performed on entry to an atomic operation, that atomic operation starts its release sequence.
- If a release flush is performed in an implicit **flush** region, an atomic operation that is provided by the implementation and that modifies an internal synchronization variable, starts its release sequence.
- If a release flush is performed by an explicit **flush** region, any atomic operation that modifies a shared variable and follows the **flush** region in its thread's program order starts an associated release sequence.

An acquire flush is associated with one or more prior atomic operations that read a shared variable and that may be used to establish a synchronization. The following rules determine the associated atomic operation that may establish a synchronization.

- If an acquire flush is performed on exit from an atomic operation, that atomic operation is its associated atomic operation.

- If an acquire flush is performed in an implicit **flush** region, an atomic operation that is provided by the implementation and that reads an internal synchronization variable is its associated atomic operation.
- If an acquire flush is performed by an explicit **flush** region, any atomic operation that reads a shared variable and precedes the **flush** region in its thread's program order is an associated atomic operation.

A release flush synchronizes with an acquire flush if an atomic operation associated with the acquire flush reads a value written by a modification from a release sequence associated with the release flush.

An operation *X simply happens before* an operation *Y* if any of the following conditions are satisfied:

1. *X* and *Y* are performed by the same thread, and *X* precedes *Y* in the thread's program order;
2. *X* synchronizes with *Y* according to the flush synchronization conditions explained above or according to the base language's definition of *synchronizes with*, if such a definition exists; or
3. There exists another operation *Z*, such that *X* simply happens before *Z* and *Z* simply happens before *Y*.

An operation *X happens before* an operation *Y* if any of the following conditions are satisfied:

1. *X* happens before *Y* according to the base language's definition of *happens before*, if such a definition exists; or
2. *X* simply happens before *Y*.

A variable with an initial value is treated as if the value is stored to the variable by an operation that happens before all operations that access or modify the variable in the program.

1.4.6 OpenMP Memory Consistency

The following rules guarantee the observable completion order of memory operations, as seen by all threads.

- If two operations performed by different threads are sequentially consistent atomic operations or they are strong flushes that flush the same variable, then they must be completed as if in some sequential order, seen by all threads.
- If two operations performed by the same thread are sequentially consistent atomic operations or they access, modify, or, with a strong flush, flush the same variable, then they must be completed as if in that thread's program order, as seen by all threads.
- If two operations are performed by different threads and one happens before the other, then they must be completed as if in that *happens before* order, as seen by all threads, if:

- both operations access or modify the same variable;
 - both operations are strong flushes that flush the same variable; or
 - both operations are sequentially consistent atomic operations.
- Any two atomic memory operations from different **atomic** regions must be completed as if in the same order as the strong flushes implied in their respective regions, as seen by all threads.

The flush operation can be specified using the **flush** directive, and is also implied at various locations in an OpenMP program: see Section 2.17.8 on page 242 for details.

Note – Since flush operations by themselves cannot prevent data races, explicit flush operations are only useful in combination with non-sequentially consistent atomic directives.

OpenMP programs that:

- Do not use non-sequentially consistent atomic directives;
 - Do not rely on the accuracy of a *false* result from **omp_test_lock** and **omp_test_nest_lock**; and
 - Correctly avoid data races as required in Section 1.4.1 on page 23,
- behave as though operations on shared variables were simply interleaved in an order consistent with the order in which they are performed by each thread. The relaxed consistency model is invisible for such programs, and any explicit flush operations in such programs are redundant.

1.5 Tool Interfaces

The OpenMP API includes two tool interfaces, OMPT and OMPD, to enable development of high-quality, portable, tools that support monitoring, performance, or correctness analysis and debugging of OpenMP programs developed using any implementation of the OpenMP API,

1.5.1 OMPT

The OMPT interface, which is intended for *first-party* tools, provides the following:

- A mechanism to initialize a first-party tool;

- Routines that enable a tool to determine the capabilities of an OpenMP implementation;
- Routines that enable a tool to examine OpenMP state information associated with a thread;
- Mechanisms that enable a tool to map implementation-level calling contexts back to their source-level representations;
- A callback interface that enables a tool to receive notification of OpenMP *events*;
- A tracing interface that enables a tool to trace activity on OpenMP target devices; and
- A runtime library routine that an application can use to control a tool.

OpenMP implementations may differ with respect to the *thread states* that they support, the mutual exclusion implementations that they employ, and the OpenMP events for which tool callbacks are invoked. For some OpenMP events, OpenMP implementations must guarantee that a registered callback will be invoked for each occurrence of the event. For other OpenMP events, OpenMP implementations are permitted to invoke a registered callback for some or no occurrences of the event; for such OpenMP events, however, OpenMP implementations are encouraged to invoke tool callbacks on as many occurrences of the event as is practical. Section 4.2.4 specifies the subset of OMPT callbacks that an OpenMP implementation must support for a minimal implementation of the OMPT interface.

An implementation of the OpenMP API may differ from the abstract execution model described by its specification. The ability of tools that use the OMPT interface to observe such differences does not constrain implementations of the OpenMP API in any way.

With the exception of the `omp_control_tool` runtime library routine for tool control, all other routines in the OMPT interface are intended for use only by tools and are not visible to applications. For that reason, a Fortran binding is provided only for `omp_control_tool`; all other OMPT functionality is described with C syntax only.

1.5.2 OMPD

The OMPD interface is intended for *third-party* tools, which run as separate processes. An OpenMP implementation must provide an OMPD library that can be dynamically loaded and used by a third-party tool. A third-party tool, such as a debugger, uses the OMPD library to access OpenMP state of a program that has begun execution. OMPD defines the following:

- An interface that an OMPD library exports, which a tool can use to access OpenMP state of a program that has begun execution;
- A callback interface that a tool provides to the OMPD library so that the library can use it to access the OpenMP state of a program that has begun execution; and

- A small number of symbols that must be defined by an OpenMP implementation to help the tool find the correct OMPD library to use for that OpenMP implementation and to facilitate notification of events.

Section 5 describes OMPD in detail.

1.6 OpenMP Compliance

The OpenMP API defines constructs that operate in the context of the base language that is supported by an implementation. If the implementation of the base language does not support a language construct that appears in this document, a compliant OpenMP implementation is not required to support it, with the exception that for Fortran, the implementation must allow case insensitivity for directive and API routines names, and must allow identifiers of more than six characters. An implementation of the OpenMP API is compliant if and only if it compiles and executes all other conforming programs, and supports the tool interface, according to the syntax and semantics laid out in Chapters 1, 2, 3, 4 and 5. Appendices A, B, C, and D, as well as sections designated as Notes (see Section 1.8 on page 34) are for information purposes only and are not part of the specification.

All library, intrinsic and built-in routines provided by the base language must be thread-safe in a compliant implementation. In addition, the implementation of the base language must also be thread-safe. For example, **ALLOCATE** and **DEALLOCATE** statements must be thread-safe in Fortran. Unsynchronized concurrent use of such routines by different threads must produce correct results (although not necessarily the same as serial execution results, as in the case of random number generation routines).

Starting with Fortran 90, variables with explicit initialization have the **SAVE** attribute implicitly. This is not the case in Fortran 77. However, a compliant OpenMP Fortran implementation must give such a variable the **SAVE** attribute, regardless of the underlying base language version.

Appendix A lists certain aspects of the OpenMP API that are implementation defined. A compliant implementation must define and document its behavior for each of the items in Appendix A.

1.7 Normative References

- ISO/IEC 9899:1990, *Information Technology - Programming Languages - C*.

This OpenMP API specification refers to ISO/IEC 9899:1990 as C90.

- ISO/IEC 9899:1999, *Information Technology - Programming Languages - C*.

This OpenMP API specification refers to ISO/IEC 9899:1999 as C99.

- ISO/IEC 9899:2011, *Information Technology - Programming Languages - C*.

This OpenMP API specification refers to ISO/IEC 9899:2011 as C11. While future versions of the OpenMP specification are expected to address the following features, currently their use may result in unspecified behavior.

- Supporting the noreturn property
- Adding alignment support
- Creation of complex value
- Threads for the C standard library
- Thread-local storage
- Parallel memory sequencing model
- Atomic

- ISO/IEC 14882:1998, *Information Technology - Programming Languages - C++*.

This OpenMP API specification refers to ISO/IEC 14882:1998 as C++98.

- ISO/IEC 14882:2011, *Information Technology - Programming Languages - C++*.

This OpenMP API specification refers to ISO/IEC 14882:2011 as C++11. While future versions of the OpenMP specification are expected to address the following features, currently their use may result in unspecified behavior.

- Alignment support
- Standard layout types
- Allowing move constructs to throw
- Defining move special member functions
- Concurrency
- Data-dependency ordering: atomics and memory model
- Additions to the standard library
- Thread-local storage
- Dynamic initialization and destruction with concurrency
- C++11 library

- 1 • ISO/IEC 14882:2014, *Information Technology - Programming Languages - C++*.
- 2 This OpenMP API specification refers to ISO/IEC 14882:2014 as C++14. While future versions
- 3 of the OpenMP specification are expected to address the following features, currently their use
- 4 may result in unspecified behavior.
- 5 – Sized deallocation
- 6 – What signal handlers can do
- 7 • ISO/IEC 14882:2017, *Information Technology - Programming Languages - C++*.
- 8 This OpenMP API specification refers to ISO/IEC 14882:2017 as C++17.
- 9 • ISO/IEC 1539:1980, *Information Technology - Programming Languages - Fortran*.
- 10 This OpenMP API specification refers to ISO/IEC 1539:1980 as Fortran 77.
- 11 • ISO/IEC 1539:1991, *Information Technology - Programming Languages - Fortran*.
- 12 This OpenMP API specification refers to ISO/IEC 1539:1991 as Fortran 90.
- 13 • ISO/IEC 1539-1:1997, *Information Technology - Programming Languages - Fortran*.
- 14 This OpenMP API specification refers to ISO/IEC 1539-1:1997 as Fortran 95.
- 15 • ISO/IEC 1539-1:2004, *Information Technology - Programming Languages - Fortran*.
- 16 This OpenMP API specification refers to ISO/IEC 1539-1:2004 as Fortran 2003.
- 17 • ISO/IEC 1539-1:2010, *Information Technology - Programming Languages - Fortran*.
- 18 This OpenMP API specification refers to ISO/IEC 1539-1:2010 as Fortran 2008. While future
- 19 versions of the OpenMP specification are expected to address the following features, currently
- 20 their use may result in unspecified behavior.
- 21 – Submodules
- 22 – Coarrays
- 23 – DO CONCURRENT
- 24 – Allocatable components of recursive type
- 25 – Pointer initialization
- 26 – Value attribute is permitted for any nonallocatable nonpointer nonarray
- 27 – Simply contiguous arrays rank remapping to rank>1 target
- 28 – Polymorphic assignment
- 29 – Accessing real and imaginary parts
- 30 – Pointer function reference is a variable

- Recursive I/O
- The BLOCK construct
- EXIT statement (to terminate a non-DO construct)
- ERROR STOP
- Internal procedure as an actual argument
- Generic resolution by proceduriness
- Generic resolution by pointer vs. allocatable
- Impure elemental procedures

Where this OpenMP API specification refers to C, C++ or Fortran, reference is made to the base language supported by the implementation.

1.8 Organization of this Document

The remainder of this document is structured as follows:

- Chapter 2 “Directives”
- Chapter 3 “Runtime Library Routines”
- Chapter 4 “OMPT Interface”
- Chapter 5 “OMPD Interface”
- Chapter 6 “Environment Variables”
- Appendix A “OpenMP Implementation-Defined Behaviors”
- Appendix B “Features History”

Some sections of this document only apply to programs written in a certain base language. Text that applies only to programs for which the base language is C or C++ is shown as follows:

▼ C / C++ ▼

C/C++ specific text...

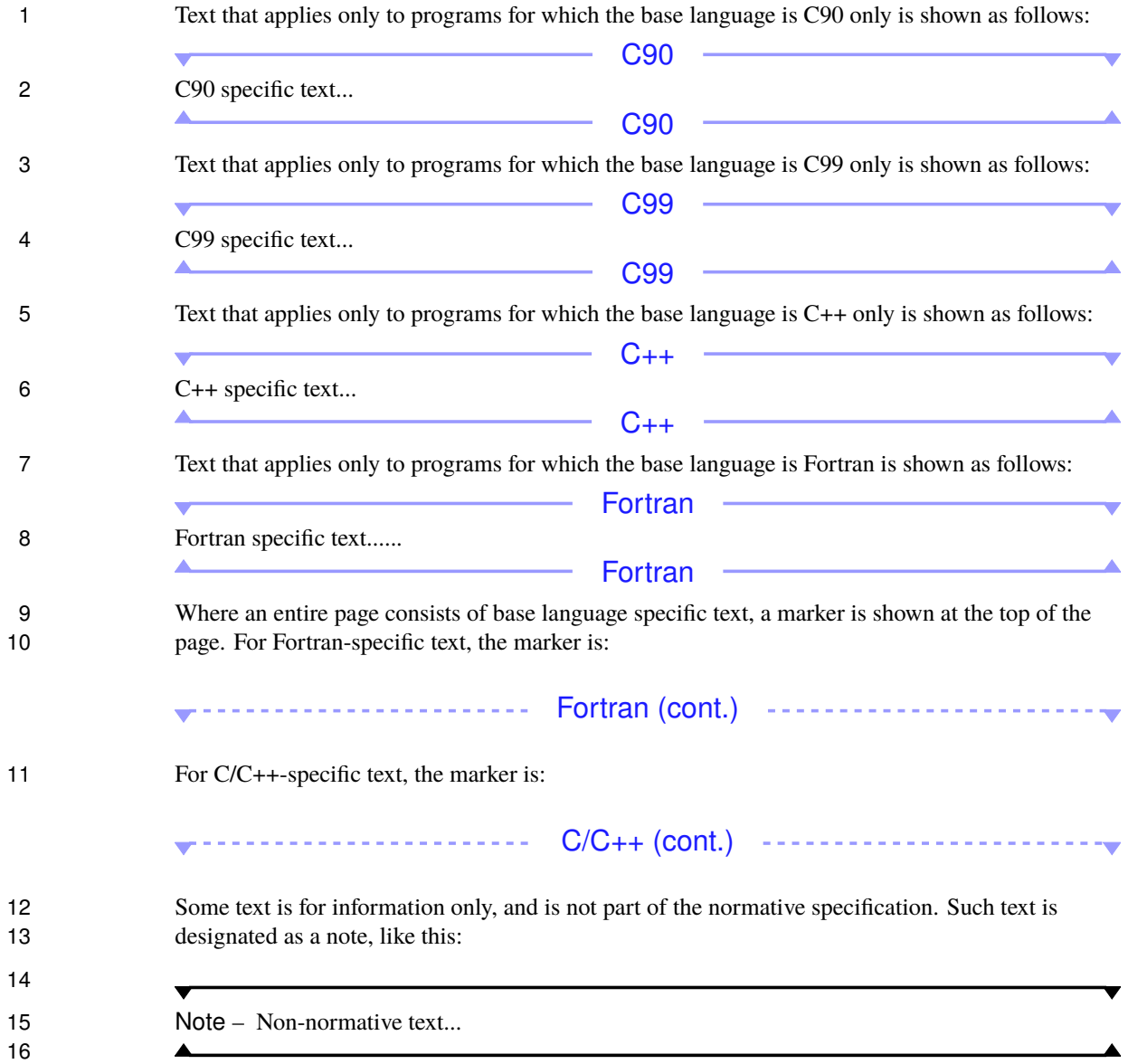
C / C++

Text that applies only to programs for which the base language is C only is shown as follows:

_____ C _____

C specific text...

 C 



This page intentionally left blank

CHAPTER 2

Directives

This chapter describes the syntax and behavior of OpenMP directives.

C / C++

In C/C++, OpenMP directives are specified by using the **#pragma** mechanism provided by the C and C++ standards.

C / C++

Fortran

In Fortran, OpenMP directives are specified by using special comments that are identified by unique sentinels. Also, a special comment form is available for conditional compilation.

Fortran

Compilers can therefore ignore OpenMP directives and conditionally compiled code if support of the OpenMP API is not provided or enabled. A compliant implementation must provide an option or interface that ensures that underlying support of all OpenMP directives and OpenMP conditional compilation mechanisms is enabled. In the remainder of this document, the phrase *OpenMP compilation* is used to mean a compilation with these OpenMP features enabled.

Fortran

Restrictions

The following restriction applies to all OpenMP directives:

- OpenMP directives, except **simd** and any declarative directive, may not appear in pure procedures.
- OpenMP directives may not appear in the WHERE and FORALL constructs.

Fortran

2.1 Directive Format

C / C++

OpenMP directives for C/C++ are specified with **#pragma** directives. The syntax of an OpenMP directive is as follows:

```
#pragma omp directive-name [clause[ [, ] clause] ... ] new-line
```

Each directive starts with **#pragma omp**. The remainder of the directive follows the conventions of the C and C++ standards for compiler directives. In particular, white space can be used before and after the **#**, and sometimes white space must be used to separate the words in a directive. Preprocessing tokens following **#pragma omp** are subject to macro replacement.

Some OpenMP directives may be composed of consecutive **#pragma** directives if specified in their syntax.

Directives are case-sensitive.

Each of the expressions used in the OpenMP syntax inside of the clauses must be a valid *assignment-expression* of the base language unless otherwise specified.

C / C++

C++

Directives may not appear in **constexpr** functions or in constant expressions. Variadic parameter packs cannot be expanded into a directive or its clauses except as part of an expression argument to be evaluated by the base language, such as into a function call inside an **if** clause.

C++

Fortran

OpenMP directives for Fortran are specified as follows:

```
sentinel directive-name [clause[ [, ] clause]...]
```

All OpenMP compiler directives must begin with a directive *sentinel*. The format of a sentinel differs between fixed form and free form source files, as described in Section 2.1.1 on page 41 and Section 2.1.2 on page 41.

Directives are case insensitive. Directives cannot be embedded within continued statements, and statements cannot be embedded within directives.

Each of the expressions used in the OpenMP syntax inside of the clauses must be a valid *expression* of the base language unless otherwise specified.

In order to simplify the presentation, free form is used for the syntax of OpenMP directives for Fortran in the remainder of this document, except as noted.

Fortran

Only one *directive-name* can be specified per directive (note that this includes combined directives, see Section 2.13 on page 185). The order in which clauses appear on directives is not significant. Clauses on directives may be repeated as needed, subject to the restrictions listed in the description of each clause.

Some clauses accept a *list*, an *extended-list*, or a *locator-list*. A *list* consists of a comma-separated collection of one or more *list items*. An *extended-list* consists of a comma-separated collection of one or more *extended list items*. A *locator-list* consists of a comma-separated collection of one or more *locator list items*.

C / C++

A *list item* is a variable or an array section. An *extended list item* is a *list item* or a function name. A *locator list item* is any lvalue expression, including variables, or an array section.

C / C++

Fortran

A *list item* is a variable, array section or common block name (enclosed in slashes). An *extended list item* is a *list item* or a procedure name. A *locator list item* is a *list item*.

When a named common block appears in a *list*, it has the same meaning as if every explicit member of the common block appeared in the list. An explicit member of a common block is a variable that is named in a **COMMON** statement that specifies the common block name and is declared in the same scoping unit in which the clause appears.

Although variables in common blocks can be accessed by use association or host association, common block names cannot. As a result, a common block name specified in a data-sharing attribute, a data copying or a data-mapping attribute clause must be declared to be a common block in the same scoping unit in which the clause appears.

If a list item that appears in a directive or clause is an optional dummy argument that is not present, the directive or clause for that list item is ignored.

If the variable referenced inside a construct is an optional dummy argument that is not present, any explicitly determined, implicitly determined, or predetermined data-sharing and data-mapping attribute rules for that variable are ignored. Otherwise, if the variable is an optional dummy argument that is present, it is present inside the construct.

Fortran

For all base languages, a *list item*, an *extended list item*, or a *locator list item* is subject to the restrictions specified in Section 2.1.5 on page 44 and in each of the sections describing clauses and directives for which the *list*, the *extended-list*, or the *locator-list* appears.

Some executable directives include a structured block. A structured block:

- may contain infinite loops where the point of exit is never reached;
- may halt due to an IEEE exception;

C / C++

- may contain calls to **exit()**, **_Exit()**, **quick_exit()**, **abort()** or functions with a **_Noreturn** specifier (in C) or a **noreturn** attribute (in C/C++);
- may be an expression statement, iteration statement, selection statement, or try block, provided that the corresponding compound statement obtained by enclosing it in **{** and **}** would be a structured block; and

C / C++

Fortran

- may contain **STOP** statements.

Fortran

Restrictions

Restrictions to structured blocks are as follows:

- Entry to a structured block must not be the result of a branch.
- The point of exit cannot be a branch out of the structured block.

C / C++

- The point of entry to a structured block must not be a call to **setjmp()**.
- **longjmp()** and **throw()** must not violate the entry/exit criteria.

C / C++

2.1.1 Fixed Source Form Directives

The following sentinels are recognized in fixed form source files:

!\$omp | c\$omp | *\$omp

Sentinels must start in column 1 and appear as a single word with no intervening characters. Fortran fixed form line length, white space, continuation, and column rules apply to the directive line. Initial directive lines must have a space or a zero in column 6, and continuation directive lines must have a character other than a space or a zero in column 6.

Comments may appear on the same line as a directive. The exclamation point initiates a comment when it appears after column 6. The comment extends to the end of the source line and is ignored. If the first non-blank character after the directive sentinel of an initial or continuation directive line is an exclamation point, the line is ignored.

Note – In the following example, the three formats for specifying the directive are equivalent (the first line represents the position of the first 9 columns):

```
c23456789
!$omp parallel do shared(a,b,c)

c$omp parallel do
c$omp+shared(a,b,c)

c$omp paralleldoshared(a,b,c)
```

2.1.2 Free Source Form Directives

The following sentinel is recognized in free form source files:

!\$omp

The sentinel can appear in any column as long as it is preceded only by white space. It must appear as a single word with no intervening white space. Fortran free form line length, white space, and continuation rules apply to the directive line. Initial directive lines must have a space after the sentinel. Continued directive lines must have an ampersand (&) as the last non-blank character on the line, prior to any comment placed inside the directive. Continuation directive lines can have an ampersand after the directive sentinel with optional white space before and after the ampersand.

Comments may appear on the same line as a directive. The exclamation point (!) initiates a comment. The comment extends to the end of the source line and is ignored. If the first non-blank character after the directive sentinel is an exclamation point, the line is ignored.

One or more blanks or horizontal tabs are optional to separate adjacent keywords in *directive-names* unless otherwise specified.

Note – In the following example the three formats for specifying the directive are equivalent (the first line represents the position of the first 9 columns):

```
!23456789
    !$omp parallel do &
        !$omp shared(a,b,c)

    !$omp parallel &
    !$omp&do shared(a,b,c)

!$omp paralleldo shared(a,b,c)
```

Fortran

2.1.3 Stand-Alone Directives

Summary

Stand-alone directives are executable directives that have no associated user code.

Description

Stand-alone directives do not have any associated executable user code. Instead, they represent executable statements that typically do not have succinct equivalent statements in the base language. There are some restrictions on the placement of a stand-alone directive within a program. A stand-alone directive may be placed only at a point where a base language executable statement is allowed.

1 **Restrictions**

C / C++

- 2 • A stand-alone directive may not be used in place of the statement following an **if**, **while**, **do**,
3 **switch**, or **label**.

C / C++

Fortran

- 4 • A stand-alone directive may not be used as the action statement in an **if** statement or as the
5 executable statement following a label if the label is referenced in the program.

Fortran

C / C++

6 **2.1.4 Array Shaping**

7 If an expression has a type of pointer to T , then a shape-operator can be used to specify the extent of
8 that pointer. In other words, the shape-operator is used to reinterpret, as an n -dimensional array, the
9 region of memory to which that expression points.

10 Formally, the syntax of the shape-operator is as follows:

11 | *shaped-expression* := ([s_1] [s_2] . . . [s_n]) *cast-expression*

12 The result of applying the shape-operator to an expression is an lvalue expression with an
13 n -dimensional array type with dimensions $s_1 \times s_2 \dots \times s_n$ and element type T .

14 The precedence of the shape-operator is the same as a type cast.

15 Each s_i is an integral type expression that must evaluate to a positive integer.

16 **Restrictions**

17 Restrictions to the shape-operator are as follows:

- 18 • The type T must be a complete type.
19 • The shape-operator can appear only in clauses where it is explicitly allowed.
20 • The result of a shape-operator must be a named array of a list item.
21 • The type of the expression upon which a shape-operator is applied must be a pointer type.

C++

- 22 • If the type T is a reference to a type T' , then the type will be considered to be T' for all purposes
23 of the designated array.

C++

C / C++

2.1.5 Array Sections

An array section designates a subset of the elements in an array.

C / C++

To specify an array section in an OpenMP construct, array subscript expressions are extended with the following syntax:

```
[ lower-bound : length : stride ] or  
[ lower-bound : length : ] or  
[ lower-bound : length ] or  
[ lower-bound : : stride ] or  
[ lower-bound : : ] or  
[ lower-bound : ] or  
[ : length : stride ] or  
[ : length : ] or  
[ : length ] or  
[ : : stride ]  
[ : : ]  
[ : ]
```

The array section must be a subset of the original array.

Array sections are allowed on multidimensional arrays. Base language array subscript expressions can be used to specify length-one dimensions of multidimensional array sections.

Each of the *lower-bound*, *length*, and *stride* expressions if specified must be an integral type *expression* of the base language. When evaluated they represent a set of integer values as follows:

{ *lower-bound*, *lower-bound* + *stride*, *lower-bound* + 2 * *stride*, ..., *lower-bound* + ((*length* - 1) * *stride*) }

The *length* must evaluate to a non-negative integer.

The *stride* must evaluate to a positive integer.

When the size of the array dimension is not known, the *length* must be specified explicitly.

When the *stride* is absent it defaults to 1.

When the *length* is absent it defaults to $\lceil (size - lower-bound) / stride \rceil$, where *size* is the size of the array dimension.

When the *lower-bound* is absent it defaults to 0.

The precedence of a subscript operator that uses the array section syntax is the same as the precedence of a subscript operator that does not use the array section syntax.

Note – The following are examples of array sections:

```

a[0:6]
a[0:6:1]
a[1:10]
a[1:]
a[:10:2]
b[10][:][:]
b[10][:][:0]
c[42][0:6][:]
c[42][0:6:2][:]
c[1:10][42][0:6]
S.c[:100]
p->y[:10]
this->a[:N]
(p+10) [:N]

```

Assume **a** is declared to be a 1-dimensional array with dimension size 11. The first two examples are equivalent, and the third and fourth examples are equivalent. The fifth example specifies a stride of 2 and therefore is not contiguous.

Assume **b** is declared to be a pointer to a 2-dimensional array with dimension sizes 10 and 10. The sixth example refers to all elements of the 2-dimensional array given by **b**[10]. The seventh example is a zero-length array section.

Assume **c** is declared to be a 3-dimensional array with dimension sizes 50, 50, and 50. The eighth example is contiguous, while the ninth and tenth examples are not contiguous.

The final four examples show array sections that are formed from more general base expressions.

The following are examples that are non-conforming array sections:

```

s[:10].x
p[:10]->y
*(xp[:10])

```

For all three examples, a base language operator is applied in an undefined manner to an array section. The only operator that may be applied to an array section is a subscript operator for which the array section appears as the postfix expression.

C / C++

Fortran

Fortran has built-in support for array sections although some restrictions apply to their use, as enumerated in the following section.

Fortran

Restrictions

Restrictions to array sections are as follows:

- An array section can appear only in clauses where it is explicitly allowed.
- A *stride* expression may not be specified unless otherwise stated.

C / C++

- An element of an array section with a non-zero size must have a complete type.
- The base expression of an array section must have an array or pointer type.
- If a consecutive sequence of array subscript expressions appears in an array section, and the first subscript expression in the sequence uses the extended array section syntax defined in this section, then only the last subscript expression in the sequence may select array elements that have a pointer type.

C / C++

C++

- If the type of the base expression of an array section is a reference to a type *T*, then the type will be considered to be *T* for all purposes of the array section.
- An array section cannot be used in an overloaded `[]` operator.

C++

Fortran

- If a stride expression is specified, it must be positive.
- The upper bound for the last dimension of an assumed-size dummy array must be specified.
- If a list item is an array section with vector subscripts, the first array element must be the lowest in the array element order of the array section.
- If a list item is an array section, the last *part-ref* of the list item must have a section subscript list.

Fortran

1 2.1.6 Iterators

2 Iterators are identifiers that expand to multiple values in the clause on which they appear.

3 The syntax of the **iterator** modifier is as follows:

4 **iterator** (*iterators-definition*)

5 where *iterators-definition* is one of the following:

6 *iterator-specifier* [, *iterators-definition*]

7 where *iterator-specifier* is one of the following:

8 [*iterator-type*] *identifier* = *range-specification*

9 where:

10 • *identifier* is a base language identifier.

▼ C / C++ ▼

11 • *iterator-type* is a type name.

▲ C / C++ ▲

▼ Fortran ▼

12 • *iterator-type* is a type specifier.

▲ Fortran ▲

13 • *range-specification* is of the form *begin* : *end* [: *step*], where *begin* and *end* are expressions for
14 which their types can be converted to *iterator-type* and *step* is an integral expression.

▼ C / C++ ▼

15 In an *iterator-specifier*, if the *iterator-type* is not specified then the type of that iterator is of **int**
16 type.

▲ C / C++ ▲

▼ Fortran ▼

17 In an *iterator-specifier*, if the *iterator-type* is not specified then the type of that iterator is default
18 integer.

▲ Fortran ▲

19 In a *range-specification*, if the *step* is not specified its value is implicitly defined to be 1.

20 An iterator only exists in the context of the clause in which it appears. An iterator also hides all
21 accessible symbols with the same name in the context of the clause.

22 The use of a variable in an expression that appears in the *range-specification* causes an implicit
23 reference to the variable in all enclosing constructs.

C / C++

The values of the iterator are the set of values i_0, \dots, i_{N-1} where:

- $i_0 = (\text{iterator-type}) \text{ begin}$,
- $i_j = (\text{iterator-type}) (i_{j-1} + \text{step})$, and
- if $\text{step} > 0$,
 - $i_0 < (\text{iterator-type}) \text{ end}$,
 - $i_{N-1} < (\text{iterator-type}) \text{ end}$, and
 - $(\text{iterator-type}) (i_{N-1} + \text{step}) \geq (\text{iterator-type}) \text{ end}$;
- if $\text{step} < 0$,
 - $i_0 > (\text{iterator-type}) \text{ end}$,
 - $i_{N-1} > (\text{iterator-type}) \text{ end}$, and
 - $(\text{iterator-type}) (i_{N-1} + \text{step}) \leq (\text{iterator-type}) \text{ end}$.

C / C++

Fortran

The values of the iterator are the set of values i_1, \dots, i_N where:

- $i_1 = \text{begin}$,
- $i_j = i_{j-1} + \text{step}$, and
- if $\text{step} > 0$,
 - $i_1 \leq \text{end}$,
 - $i_N \leq \text{end}$, and
 - $i_N + \text{step} > \text{end}$;
- if $\text{step} < 0$,
 - $i_1 \geq \text{end}$,
 - $i_N \geq \text{end}$, and
 - $i_N + \text{step} < \text{end}$.





Fortran

The set of values will be empty if no possible value complies with the conditions above.

For those clauses that contain expressions that contain iterator identifiers, the effect is as if the list item is instantiated within the clause for each value of the iterator in the set defined above, substituting each occurrence of the iterator identifier in the expression with the iterator value. If the set of values of the iterator is empty then the effect is as if the clause was not specified.

The behavior is unspecified if $i_j + step$ cannot be represented in *iterator-type* in any of the $i_j + step$ computations for any $0 \leq j < N$ in C/C++ or $0 < j \leq N$ in Fortran.

Restrictions

- An expression that contains an iterator identifier can only appear in clauses that explicitly allow expressions that contain iterators.
 - The *iterator-type* must not declare a new type.
- 
C / C++
- The *iterator-type* must be an integral or pointer type.
 - The *iterator-type* must not be **const** qualified.
- 
C / C++
- 
Fortran
- The *iterator-type* must be an integer type.
- 
Fortran
- If the *step* expression of a *range-specification* equals zero, the behavior is unspecified.
 - Each iterator identifier can only be defined once in an *iterators-definition*.
 - Iterators cannot appear in the *range-specification*.

2.2 Conditional Compilation

In implementations that support a preprocessor, the **_OPENMP** macro name is defined to have the decimal value *yyyymm* where *yyyy* and *mm* are the year and month designations of the version of the OpenMP API that the implementation supports.

If a **#define** or a **#undef** preprocessing directive in user code defines or undefines the **_OPENMP** macro name, the behavior is unspecified.


Fortran

The OpenMP API requires Fortran lines to be compiled conditionally, as described in the following sections.

2.2.1 Fixed Source Form Conditional Compilation Sentinels

The following conditional compilation sentinels are recognized in fixed form source files:

!\$ | *\$ | c\$

To enable conditional compilation, a line with a conditional compilation sentinel must satisfy the following criteria:

- The sentinel must start in column 1 and appear as a single word with no intervening white space;
- After the sentinel is replaced with two spaces, initial lines must have a space or zero in column 6 and only white space and numbers in columns 1 through 5;
- After the sentinel is replaced with two spaces, continuation lines must have a character other than a space or zero in column 6 and only white space in columns 1 through 5.

If these criteria are met, the sentinel is replaced by two spaces. If these criteria are not met, the line is left unchanged.

Note – In the following example, the two forms for specifying conditional compilation in fixed source form are equivalent (the first line represents the position of the first 9 columns):

```
c23456789
!$ 10 iam = omp_get_thread_num() +
!$   &           index

#ifdef _OPENMP
    10 iam = omp_get_thread_num() +
        &           index
#endif
```

2.2.2 Free Source Form Conditional Compilation Sentinel

The following conditional compilation sentinel is recognized in free form source files:

!\$

To enable conditional compilation, a line with a conditional compilation sentinel must satisfy the following criteria:

- The sentinel can appear in any column but must be preceded only by white space;
- The sentinel must appear as a single word with no intervening white space;

- Initial lines must have a space after the sentinel;
 - Continued lines must have an ampersand as the last non-blank character on the line, prior to any comment appearing on the conditionally compiled line.
- Continuation lines can have an ampersand after the sentinel, with optional white space before and after the ampersand. If these criteria are met, the sentinel is replaced by two spaces. If these criteria are not met, the line is left unchanged.

Note – In the following example, the two forms for specifying conditional compilation in free source form are equivalent (the first line represents the position of the first 9 columns):

```
c23456789
!$ iam = omp_get_thread_num() +      &
!$&   index

#ifdef _OPENMP
    iam = omp_get_thread_num() +      &
    index
#endif
```

Fortran

2.3 Variant Directives

2.3.1 OpenMP Context

At any point in a program, an OpenMP context exists that defines traits that describe the active OpenMP constructs, the execution devices, and functionality supported by the implementation. The traits are grouped into trait sets. The following trait sets exist: *construct*, *device* and *implementation*.

The *construct* set is composed of the directive names, each being a trait, of all enclosing constructs at that point in the program up to a **target** construct. Combined and composite constructs are added to the set as distinct constructs in the same nesting order specified by the original construct. The set is ordered by nesting level in ascending order. Specifically, the ordering of the set of constructs is c_1, \dots, c_N , where c_1 is the construct at the outermost nesting level and c_N is the construct at the innermost nesting level. In addition, if the point in the program is not enclosed by a **target** construct, the following rules are applied in order:

- 1 1. For functions with a **declare simd** directive, the *simd* trait is added to the beginning of the
2 set as c_1 for any generated SIMD versions so the total size of the set is increased by 1.
- 3 2. For functions that are determined to be function variants by a **declare variant** directive,
4 the selectors c_1, \dots, c_M of the **construct** selector set are added in the same order to the
5 beginning of the set as c_1, \dots, c_M so the total size of the set is increased by M .
- 6 3. For functions within a **declare target** block, the *target* trait is added to the beginning of the
7 set as c_1 for any versions of the function that are generated for **target** regions so the total size
8 of the set is increased by 1.

9 The *simd* trait can be further defined with properties that match the clauses accepted by the
10 **declare simd** directive with the same name and semantics. The *simd* trait must define at least
11 the *simdlen* property and one of the *inbranch* or *notinbranch* properties.

12 The *device* set includes traits that define the characteristics of the device being targeted by the
13 compiler at that point in the program. At least the following traits must be defined:

- 14 • The *kind(kind-name-list)* trait specifies the general kind of the device. The following *kind-name*
15 values are defined:
 - 16 – *host*, which specifies that the device is the host device;
 - 17 – *nohost*, which specifies that the devices is not the host device; and
 - 18 – the values defined in the “OpenMP Context Definitions” document, which is available at
19 <http://www.openmp.org/>.
- 20 • The *isa(isa-name-list)* trait specifies the Instruction Set Architectures supported by the device.
21 The accepted *isa-name* values are implementation defined.
- 22 • The *arch(arch-name-list)* trait specifies the architectures supported by the device. The accepted
23 *arch-name* values are implementation defined.

24 The *implementation* set includes traits that describe the functionality supported by the OpenMP
25 implementation at that point in the program. At least the following traits can be defined:

- 26 • The *vendor(vendor-name-list)* trait, which specifies the vendor identifiers of the implementation.
27 OpenMP defined values for *vendor-name* are defined in the “OpenMP Context Definitions”
28 document, which is available at <http://www.openmp.org/>.
- 29 • The *extension(extension-name-list)* trait, which specifies vendor specific extensions to the
30 OpenMP specification. The accepted *extension-name* values are implementation defined.
- 31 • A trait with a name that is identical to the name of any clause that can be supplied to the
32 **requires** directive.

Implementations can define further traits in the *device* and *implementation* sets. All implementation defined traits must follow the following syntax:

```

identifier[(context-element[, context-element[, ...]])]

context-element:
    identifier[(context-element[, context-element[, ...]])]
    or
    context-value

context-value:
    constant string
    or
    constant integer expression

```

where *identifier* is a base language identifier.

2.3.2 Context Selectors

Context selectors are used to define the properties of an OpenMP context that a directive or clause can match. OpenMP defines different sets of selectors, each containing different selectors.

The syntax to define a *context-selector-specification* is the following:

```

trait-set-selector[, trait-set-selector[, ...]]

trait-set-selector:
    trait-set-selector-name={trait-selector[, trait-selector[, ...]]}

trait-selector:
    trait-selector-name[( [trait-score: ] trait-property[, trait-property[, ...]])]

trait-score:
    score (score-expression)

```

The **construct** selector set defines the *construct* traits that should be active in the OpenMP context. The following selectors can be defined in the **construct** set: **target**; **teams**; **parallel**; **for** (in C/C++); **do** (in Fortran); and **simd**. The properties of each selector are the same properties that are defined for the corresponding trait. The **construct** selector is an ordered list c_1, \dots, c_N .

The **device** and **implementation** selector sets define the traits that should be active in the corresponding trait set of the OpenMP context. The same traits defined in the corresponding traits

sets can be used as selectors with the same properties. The **kind** selector of the **device** selector set can also be set to the value **any**, which is as if no **kind** selector was specified.

The **user** selector set defines the **condition** selector that provides additional user-defined conditions.

C

The **condition**(*boolean-expr*) selector defines a *constant expression* that must evaluate to true for the selector to be true.

C

C++

The **condition**(*boolean-expr*) selector defines a **constexpr** expression that must evaluate to true for the selector to be true.

C++

Fortran

The **condition**(*logical-expr*) selector defines a *constant expression* that must evaluate to true for the selector to be true.

Fortran

A *score-expression* must be an constant integer expression.

Implementations can allow further selectors to be specified. Implementations can ignore specified selectors that are not those described in this section.

Restrictions

- Each *trait-set-selector-name* can only be specified once.
- Each *trait-selector-name* can only be specified once.
- A *trait-score* cannot be specified in traits from the **construct** or **device** *trait-selector-sets*.

2.3.3 Matching and Scoring Context Selectors

A given context selector is compatible with a given OpenMP context if the following conditions are satisfied:

- All selectors in the **user** set of the context selector are true;
- All selectors in the **construct**, **device**, and **implementation** sets of the context selector appear in the corresponding trait set of the OpenMP context;
- For each selector in the context selector, its properties are a subset of the properties of the corresponding trait of the OpenMP context; and
- Selectors in the **construct** set of the context selector appear in the same relative order as their corresponding traits in the *construct* trait set of the OpenMP context.

Some properties of the **simd** selector have special rules to match the properties of the *simd* trait:

- The **simdlen**(*N*) property of the selector matches the *simdlen*(*M*) trait of the OpenMP context if $M \% N$ equals zero; and
- The **aligned**(*list*:*N*) property of the selector matches the *aligned*(*list*:*M*) trait of the OpenMP context if $N \% M$ equals zero.

Among compatible context selectors, a score is computed using the following algorithm:

1. Each trait that appears in the *construct* trait set in the OpenMP context is given the value 2^{p-1} where *p* is the position of the construct trait, *c_p*, in the set;
2. The **kind**, **arch**, and **isa** selectors are given the values 2^l , 2^{l+1} and 2^{l+2} , respectively, where *l* is the number of traits in the *construct* set;
3. Traits for which a *trait-score* is specified are given the value specified by the *trait-score score-expression*;
4. The values given to any additional selectors allowed by the implementation are implemented defined;
5. Other selectors are given a value of zero; and
6. A context selector that is a strict subset of another context selector has a score of zero. For other context selectors, the final score is the sum of the values of all specified selectors plus 1. If the traits that correspond to the **construct** selectors appear multiple times in the OpenMP context, the highest valued subset of traits that contains all selectors in the same order are used.

1 2.3.4 Metadirectives

2 Summary

3 A metadirective is a directive that can specify multiple directive variants of which one may be
4 conditionally selected to replace the metadirective based on the enclosing OpenMP context.

5 Syntax

▼ C / C++ ▼

6 The syntax of a metadirective takes one of the following forms:

7 **#pragma omp metadirective** [*clause*[[,] *clause*] ...] *new-line*

8 or

9 **#pragma omp begin metadirective** [*clause*[[,] *clause*] ...] *new-line*
10 *stmt(s)*
11 **#pragma omp end metadirective**

12 where *clause* is one of the following:

13 **when** (*context-selector-specification* : [*directive-variant*])
14 **default** (*directive-variant*)

▲ C / C++ ▲
▼ Fortran ▼

15 The syntax of a metadirective takes one of the following forms:

16 **!\$omp metadirective** [*clause*[[,] *clause*] ...]

17 or

18 **!\$omp begin metadirective** [*clause*[[,] *clause*] ...]
19 *stmt(s)*
20 **!\$omp end metadirective**

21 where *clause* is one of the following:

22 **when** (*context-selector-specification* : [*directive-variant*])
23 **default** (*directive-variant*)

▲ Fortran ▲

24 In the **when** clause, *context-selector-specification* specifies a context selector (see Section 2.3.2).

25 In the **when** and **default** clauses, *directive-variant* has the following form and specifies a
26 directive variant that specifies an OpenMP directive with clauses that apply to it.

27 *directive-name* [*clause*[[,] *clause*] ...]

Description

A metadirective is a directive that behaves as if it is either ignored or replaced by the directive variant specified in one of the **when** or **default** clauses that appears on the metadirective.

The OpenMP context for a given metadirective is defined according to Section 2.3.1. For each **when** clause that appears on a metadirective, the specified directive variant, if present, is a candidate to replace the metadirective if the corresponding context selector is compatible with the OpenMP context according to the matching rules defined in Section 2.3.3. If only one compatible context selector specified by a **when** clause has the highest score and it specifies a directive variant, the directive variant will replace the metadirective. If more than one **when** clause specifies a compatible context selector that has the highest computed score and at least one specifies a directive variant, the first directive variant specified in the lexical order of those **when** clauses will replace the metadirective.

If no context selector from any **when** clause is compatible with the OpenMP context and a **default** clause is present, the directive variant specified in the **default** clause will replace the metadirective.

If a directive variant is not selected to replace a metadirective according to the above rules, the metadirective has no effect on the execution of the program.

The **begin metadirective** directive behaves identically to the **metadirective** directive, except that the directive syntax for the specified directive variants must accept a paired **end directive**. For any directive variant that is selected to replace the **begin metadirective** directive, the **end metadirective** directive will be implicitly replaced by its paired **end directive** to demarcate the statements that are affected by or are associated with the directive variant. If no directive variant is selected to replace the **begin metadirective** directive, its paired **end metadirective** directive is ignored.

Restrictions

Restrictions to metadirectives are as follows:

- The directive variant appearing in a **when** or **default** clause must not specify a **metadirective**, **begin metadirective**, or **end metadirective** directive.
- The context selector that appears in a **when** clause must not specify any properties for the **simd** selector.
- Any replacement that occurs for a metadirective must not result in a non-conforming OpenMP program.
- Any directive variant that is specified by a **when** or **default** clause on a **begin metadirective** directive must be an OpenMP directive that has a paired **end directive**, and the **begin metadirective** directive must have a paired **end metadirective** directive.
- The **default** clause may appear at most once on a metadirective.

1 2.3.5 declare variant Directive

2 Summary

3 The **declare variant** directive declares a specialized variant of a base function and specifies
4 the context in which that specialized variant is used. The **declare variant** directive is a
5 declarative directive.

6 Syntax

▼ C / C++ ▼

7 The syntax of the **declare variant** directive is as follows:

```
8 #pragma omp declare variant (variant-func-id) clause new-line  
9 [#pragma omp declare variant (variant-func-id) clause new-line]  
10 [ ... ]  
11 function definition or declaration
```

12 where *clause* is one of the following:

```
13 match (context-selector-specification)
```

14 and where *variant-func-id* is the name of a function variant that is either a base language identifier
15 or, for C++, a *template-id*.

▲ C / C++ ▲

▼ Fortran ▼

16 The syntax of the **declare variant** directive is as follows:

```
17 !$omp declare variant ([base-proc-name:]variant-proc-name) clause
```

18 where *clause* is one of the following:

```
19 match (context-selector-specification)
```

20 and where *variant-proc-name* is the name of a function variant that is a base language identifier.

▲ Fortran ▲

21 Description

22 The **declare variant** directive declares the *base function* to have the specified function
23 variant. The context selector in the **match** clause is associated with the variant.

The OpenMP context for a call to a given base function is defined according to Section 2.3.1. If the context selector that is associated with a declared function variant is compatible with the OpenMP context of a call to a base function according to the matching rules defined in Section 2.3.3 then a call to the variant is a candidate to replace the base function call. For any call to the base function for which candidate variants exist, the variant with the highest score is selected from all compatible variants. If multiple variants have the highest score, the selected variant is implementation defined. If a compatible variant exists, the call to the base function is replaced with a call to the selected variant. If no compatible variants exist then the call to the base function is not changed.

Different **declare variant** directives may be specified for different declarations of the same base function.

Any differences that the specific OpenMP context requires in the prototype of the variant from the base function prototype are implementation defined.

C++

The function variant is determined by base language standard name lookup rules ([basic.lookup]) of *variant-func-id* with arguments that correspond to the argument types in the base function declaration.

The *variant-func-id* and any expressions inside of the **match** clause are interpreted as if they appeared at the scope of the trailing return type of the base function.

C++

Restrictions

Restrictions to the **declare variant** directive are as follows:

- Calling functions that a **declare variant** directive determined to be a function variant directly in an OpenMP context that is different from the one that the **construct** selector set of the context selector specifies is non-conforming.
- If a function is determined to be a function variant through more than one **declare variant** directive then the **construct** selector set of their context selectors must be the same.

C / C++

- If the function has any declarations, then the **declare variant** directives for any declarations that have one must be equivalent. If the function definition has a **declare variant**, it must also be equivalent. Otherwise, the result is unspecified.

C / C++

C++

- The **declare variant** directive cannot be specified for a virtual function.
- The type of the function variant must be compatible with the type of the base function after the implementation-defined transformation for its OpenMP context.

C++

Fortran

- *base-proc-name* must not be a generic name, procedure pointer, or entry name.
- If *base-proc-name* is omitted then the **declare variant** directive must appear in the specification part of a subroutine subprogram or a function subprogram.
- Any **declare variant** directive must appear in the specification part of a subroutine, subprogram, function subprogram, or interface body to which it applies.
- If a **declare variant** directive is specified in an interface block for a procedure then it must match a **declare variant** directive in the definition of the procedure.
- If a procedure is declared via a procedure declaration statement then the procedure *base-proc-name* should appear in the same specification.
- If a **declare variant** directive is specified for a procedure name with an explicit interface and a **declare variant** directive is also specified for the definition of the procedure, the two **declare variant** directives must match. Otherwise the result is unspecified.

Fortran

Cross References

- OpenMP Context Specification, see Section 2.3.1 on page 51.
- Context Selectors, see Section 2.3.2 on page 53.

2.4 requires Directive

Summary

The **requires** directive specifies the features that an implementation must provide in order for the code to compile and to execute correctly. The **requires** directive is a declarative directive.

Syntax

C / C++

The syntax of the **requires** directive is as follows:

```
#pragma omp requires clause[ [ [, ] clause ] ... ] new-line
```

C / C++

Fortran

The syntax of the **requires** directive is as follows:

```
!$omp requires clause[ [ [,] clause] ... ]
```

Fortran

Where *clause* is either one of the requirement clauses listed below or a clause of the form **ext_implementation-defined-requirement** for an implementation defined requirement clause.

```
reverse_offload
unified_address
unified_shared_memory
atomic_default_mem_order(seq_cst | acq_rel | relaxed)
dynamic_allocators
```

Description

The **requires** directive specifies features that an implementation must support for correct execution. The behavior that a requirement clause specifies may override the normal behavior specified elsewhere in this document. Whether an implementation supports the feature that a given requirement clause specifies is implementation defined.

The **requires** directive specifies requirements for the execution of all code in the current compilation unit.

Note – Use of this directive makes your code less portable. Users should be aware that not all devices or implementations support all requirements.

When the **reverse_offload** clause appears on a **requires** directive, the implementation guarantees that a **target** region, for which the **target** construct specifies a **device** clause in which the **ancestor** modifier appears, can execute on the parent device of an enclosing **target** region.

When the **unified_address** clause appears on a **requires** directive, the implementation guarantees that all devices accessible through OpenMP API routines and directives use a unified address space. In this address space, a pointer will always refer to the same location in memory from all devices accessible through OpenMP. The pointers returned by **omp_target_alloc** and accessed through **use_device_ptr** are guaranteed to be pointer values that can support pointer arithmetic while still being native device pointers. The **is_device_ptr** clause is not necessary for device pointers to be translated in **target** regions, and pointers found not present are not set to null but keep their original value. Memory local to a specific execution context may be exempt from this requirement, following the restrictions of locality to a given execution context, thread, or

contention group. Target devices may still have discrete memories and dereferencing a device pointer on the host device or host pointer on a target device remains unspecified behavior.

The **unified_shared_memory** clause implies the **unified_address** requirement, inheriting all of its behaviors. Additionally, memory in the device data environment of any device visible to OpenMP, including but not limited to the host, is considered part of the device data environment of all devices accessible through OpenMP except as noted below. Every device address allocated through OpenMP device memory routines is a valid host pointer. Memory local to an execution context as defined in **unified_address** above may remain part of distinct device data environments as long as the execution context is local to the device containing that environment.

The **unified_shared_memory** clause makes the **map** clause optional on **target** constructs and the **declare target** directive optional for static lifetime variables accessed inside **declare target** functions. Scalar variables are still firstprivate by default when referenced inside **target** constructs. Values stored into memory by one device may not be visible to another device until those two devices synchronize with each other or both devices synchronize with the host.

The **atomic_default_mem_order** clause specifies the default memory ordering behavior for **atomic** constructs that must be provided by an implementation. If the default memory ordering is specified as **seq_cst**, all **atomic** constructs on which *memory-order-clause* is not specified behave as if the **seq_cst** clause appears. If the default memory ordering is specified as **relaxed**, all **atomic** constructs on which *memory-order-clause* is not specified behave as if the **relaxed** clause appears.

If the default memory ordering is specified as **acq_rel**, **atomic** constructs on which *memory-order-clause* is not specified behave as if the **release** clause appears if the atomic write or atomic update operation is specified, as if the **acquire** clause appears if the atomic read operation is specified, and as if the **acq_rel** clause appears if the atomic captured update operation is specified.

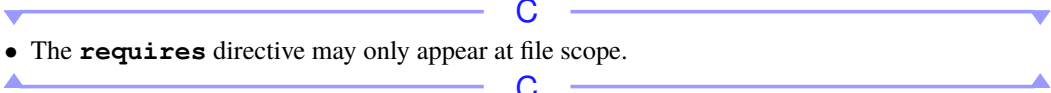
The **dynamic_allocators** clause removes certain restrictions on the use of memory allocators in **target** regions. It makes the **uses_allocators** clause optional on **target** constructs for the purpose of using allocators in the corresponding **target** regions. It allows calls to the **omp_init_allocator** and **omp_destroy_allocator** API routines in **target** regions. Finally, it allows default allocators to be used by **allocate** directives, **allocate** clauses, and **omp_alloc** API routines in **target** regions.

Implementers are allowed to include additional implementation defined requirement clauses. All implementation defined requirements should begin with **ext_**. Requirement names that do not start with **ext_** are reserved.

Restrictions

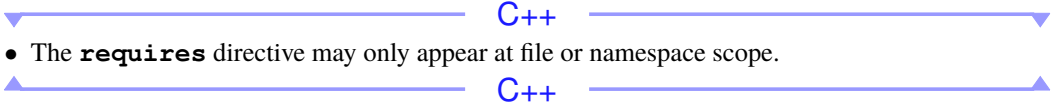
The restrictions for the **requires** directive are as follows:

- Each of the clauses can appear at most once on the directive.

- 1 • At most one **requires** directive with **atomic_default_mem_order** clause can appear in
 - 2 a single compilation unit.
 - 3 • A **requires** directive with a **unified_address**, **unified_shared_memory**, or
 - 4 **reverse_offload** clause must appear lexically before any device constructs or device
 - 5 routines.
 - 6 • A **requires** directive with any of the following clauses must appear in all *compilation units* of
 - 7 a program that contain device constructs or device routines or in none of them:
 - 8 – **reverse_offload**
 - 9 – **unified_address**
 - 10 – **unified_shared_memory**
 - 11 • The **requires** directive with **atomic_default_mem_order** clause may not appear
 - 12 lexically after any **atomic** construct on which *memory-order-clause* is not specified.
- 

▼ C ▼

13 • The **requires** directive may only appear at file scope.

▲ C ▲
- 

▼ C++ ▼

14 • The **requires** directive may only appear at file or namespace scope.

▲ C++ ▲

15 2.5 Internal Control Variables

16 An OpenMP implementation must act as if there are internal control variables (ICVs) that control

17 the behavior of an OpenMP program. These ICVs store information such as the number of threads

18 to use for future **parallel** regions, the schedule to use for worksharing loops and whether nested

19 parallelism is enabled or not. The ICVs are given values at various times (described below) during

20 the execution of the program. They are initialized by the implementation itself and may be given

21 values through OpenMP environment variables and through calls to OpenMP API routines. The

22 program can retrieve the values of these ICVs only through OpenMP API routines.

23 For purposes of exposition, this document refers to the ICVs by certain names, but an

24 implementation is not required to use these names or to offer any way to access the variables other

25 than through the ways shown in Section 2.5.2 on page 66.

1 2.5.1 ICV Descriptions

2 The following ICVs store values that affect the operation of **parallel** regions.

- 3 • *dyn-var* - controls whether dynamic adjustment of the number of threads is enabled for
4 encountered **parallel** regions. There is one copy of this ICV per data environment.
- 5 • *nthreads-var* - controls the number of threads requested for encountered **parallel** regions.
6 There is one copy of this ICV per data environment.
- 7 • *thread-limit-var* - controls the maximum number of threads participating in the contention
8 group. There is one copy of this ICV per data environment.
- 9 • *max-active-levels-var* - controls the maximum number of nested active **parallel** regions.
10 There is one copy of this ICV per device.
- 11 • *place-partition-var* - controls the place partition available to the execution environment for
12 encountered **parallel** regions. There is one copy of this ICV per implicit task.
- 13 • *active-levels-var* - the number of nested active **parallel** regions that enclose the current task
14 such that all of the **parallel** regions are enclosed by the outermost initial task region on the
15 current device. There is one copy of this ICV per data environment.
- 16 • *levels-var* - the number of nested parallel regions that enclose the current task such that all of the
17 **parallel** regions are enclosed by the outermost initial task region on the current device.
18 There is one copy of this ICV per data environment.
- 19 • *bind-var* - controls the binding of OpenMP threads to places. When binding is requested, the
20 variable indicates that the execution environment is advised not to move threads between places.
21 The variable can also provide default thread affinity policies. There is one copy of this ICV per
22 data environment.

23 The following ICVs store values that affect the operation of worksharing-loop regions.

- 24 • *run-sched-var* - controls the schedule that is used for worksharing-loop regions when the
25 **runtime** schedule kind is specified. There is one copy of this ICV per data environment.
- 26 • *def-sched-var* - controls the implementation defined default scheduling of worksharing-loop
27 regions. There is one copy of this ICV per device.

28 The following ICVs store values that affect program execution.

- 29 • *stacksize-var* - controls the stack size for threads that the OpenMP implementation creates. There
30 is one copy of this ICV per device.
- 31 • *wait-policy-var* - controls the desired behavior of waiting threads. There is one copy of this ICV
32 per device.
- 33 • *display-affinity-var* - controls whether to display thread affinity. There is one copy of this ICV for
34 the whole program.

- *affinity-format-var* - controls the thread affinity format when displaying thread affinity. There is one copy of this ICV per device.
- *cancel-var* - controls the desired behavior of the **cancel** construct and cancellation points. There is one copy of this ICV for the whole program.
- *default-device-var* - controls the default target device. There is one copy of this ICV per data environment.
- *target-offload-var* - controls the offloading behavior. There is one copy of this ICV for the whole program.
- *max-task-priority-var* - controls the maximum priority value that can be specified in the **priority** clause of the **task** construct. There is one copy of this ICV for the whole program.

The following ICVs store values that affect the operation of the OMPT tool interface.

- *tool-var* - controls whether an OpenMP implementation will try to register a tool. There is one copy of this ICV for the whole program.
- *tool-libraries-var* - specifies a list of absolute paths to tool libraries for OpenMP devices. There is one copy of this ICV for the whole program.

The following ICVs store values that affect the operation of the OMPD tool interface.

- *debug-var* - controls whether an OpenMP implementation will collect information that an OMPD library can access to satisfy requests from a tool. There is one copy of this ICV for the whole program.

The following ICVs store values that affect default memory allocation.

- *def-allocator-var* - controls the memory allocator to be used by memory allocation routines, directives and clauses when a memory allocator is not specified by the user. There is one copy of this ICV per implicit task.

1 **2.5.2 ICV Initialization**

TABLE 2.1: ICV Initial Values

ICV	Environment Variable	Initial value
<i>dyn-var</i>	OMP_DYNAMIC	See description below
<i>nthreads-var</i>	OMP_NUM_THREADS	Implementation defined
<i>run-sched-var</i>	OMP_SCHEDULE	Implementation defined
<i>def-sched-var</i>	(none)	Implementation defined
<i>bind-var</i>	OMP_PROC_BIND	Implementation defined
<i>stacksize-var</i>	OMP_STACKSIZE	Implementation defined
<i>wait-policy-var</i>	OMP_WAIT_POLICY	Implementation defined
<i>thread-limit-var</i>	OMP_THREAD_LIMIT	Implementation defined
<i>max-active-levels-var</i>	OMP_MAX_ACTIVE_LEVELS, OMP_NESTED	See description below
<i>active-levels-var</i>	(none)	<i>zero</i>
<i>levels-var</i>	(none)	<i>zero</i>
<i>place-partition-var</i>	OMP_PLACES	Implementation defined
<i>cancel-var</i>	OMP_CANCELLATION	<i>false</i>
<i>display-affinity-var</i>	OMP_DISPLAY_AFFINITY	<i>false</i>
<i>affinity-format-var</i>	OMP_AFFINITY_FORMAT	Implementation defined
<i>default-device-var</i>	OMP_DEFAULT_DEVICE	Implementation defined
<i>target-offload-var</i>	OMP_TARGET_OFFLOAD	DEFAULT
<i>max-task-priority-var</i>	OMP_MAX_TASK_PRIORITY	<i>zero</i>
<i>tool-var</i>	OMP_TOOL	<i>enabled</i>
<i>tool-libraries-var</i>	OMP_TOOL_LIBRARIES	<i>empty string</i>
<i>debug-var</i>	OMP_DEBUG	<i>disabled</i>
<i>def-allocator-var</i>	OMP_ALLOCATOR	Implementation defined

2 Table 2.1 shows the ICVs, associated environment variables, and initial values.

Description

- Each device has its own ICVs.
- The initial value of *dyn-var* is implementation defined if the implementation supports dynamic adjustment of the number of threads; otherwise, the initial value is *false*.
- The value of the *nthreads-var* ICV is a list.
- The value of the *bind-var* ICV is a list.
- The initial value of *max-active-levels-var* is the number of active levels of parallelism that the implementation supports if **OMP_NUM_THREADS** or **OMP_PROC_BIND** is set to a comma-separated list of more than one value. Otherwise, the initial value of *max-active-levels-var* is implementation defined.

The host and target device ICVs are initialized before any OpenMP API construct or OpenMP API routine executes. After the initial values are assigned, the values of any OpenMP environment variables that were set by the user are read and the associated ICVs for the host device are modified accordingly. The method for initializing a target device's ICVs is implementation defined.

Cross References

- **OMP_SCHEDULE** environment variable, see Section 6.1 on page 601.
- **OMP_NUM_THREADS** environment variable, see Section 6.2 on page 602.
- **OMP_DYNAMIC** environment variable, see Section 6.3 on page 603.
- **OMP_PROC_BIND** environment variable, see Section 6.4 on page 604.
- **OMP_PLACES** environment variable, see Section 6.5 on page 605.
- **OMP_STACKSIZE** environment variable, see Section 6.6 on page 607.
- **OMP_WAIT_POLICY** environment variable, see Section 6.7 on page 608.
- **OMP_MAX_ACTIVE_LEVELS** environment variable, see Section 6.8 on page 608.
- **OMP_NESTED** environment variable, see Section 6.9 on page 609.
- **OMP_THREAD_LIMIT** environment variable, see Section 6.10 on page 610.
- **OMP_CANCELLATION** environment variable, see Section 6.11 on page 610.
- **OMP_DISPLAY_AFFINITY** environment variable, see Section 6.13 on page 612.
- **OMP_AFFINITY_FORMAT** environment variable, see Section 6.14 on page 613.
- **OMP_DEFAULT_DEVICE** environment variable, see Section 6.15 on page 615.
- **OMP_MAX_TASK_PRIORITY** environment variable, see Section 6.16 on page 615.
- **OMP_TARGET_OFFLOAD** environment variable, see Section 6.17 on page 615.

- **OMP_TOOL** environment variable, see Section 6.18 on page 616.
- **OMP_TOOL_LIBRARIES** environment variable, see Section 6.19 on page 617.
- **OMP_DEBUG** environment variable, see Section 6.20 on page 617.
- **OMP_ALLOCATOR** environment variable, see Section 6.21 on page 618.

2.5.3 Modifying and Retrieving ICV Values

Table 2.2 shows the method for modifying and retrieving the values of ICVs through OpenMP API routines.

TABLE 2.2: Ways to Modify and to Retrieve ICV Values

ICV	Ways to Modify Value	Ways to Retrieve Value
<i>dyn-var</i>	<code>omp_set_dynamic()</code>	<code>omp_get_dynamic()</code>
<i>nthreads-var</i>	<code>omp_set_num_threads()</code>	<code>omp_get_max_threads()</code>
<i>run-sched-var</i>	<code>omp_set_schedule()</code>	<code>omp_get_schedule()</code>
<i>def-sched-var</i>	(none)	(none)
<i>bind-var</i>	(none)	<code>omp_get_proc_bind()</code>
<i>stacksize-var</i>	(none)	(none)
<i>wait-policy-var</i>	(none)	(none)
<i>thread-limit-var</i>	<code>thread_limit</code> clause	<code>omp_get_thread_limit()</code>
<i>max-active-levels-var</i>	<code>omp_set_max_active_levels()</code> , <code>omp_set_nested()</code>	<code>omp_get_max_active_levels()</code>
<i>active-levels-var</i>	(none)	<code>omp_get_active_level()</code>
<i>levels-var</i>	(none)	<code>omp_get_level()</code>
<i>place-partition-var</i>	(none)	See description below
<i>cancel-var</i>	(none)	<code>omp_get_cancellation()</code>
<i>display-affinity-var</i>	(none)	(none)
<i>affinity-format-var</i>	<code>omp_set_affinity_format()</code>	<code>omp_get_affinity_format()</code>

table continued on next page

table continued from previous page

ICV	Ways to Modify Value	Ways to Retrieve Value
<i>default-device-var</i>	omp_set_default_device()	omp_get_default_device()
<i>target-offload-var</i>	(none)	(none)
<i>max-task-priority-var</i>	(none)	omp_get_max_task_priority()
<i>tool-var</i>	(none)	(none)
<i>tool-libraries-var</i>	(none)	(none)
<i>debug-var</i>	(none)	(none)
<i>def-allocator-var</i>	omp_set_default_allocator()	omp_get_default_allocator()

Description

- The value of the *nthreads-var* ICV is a list. The runtime call **omp_set_num_threads** sets the value of the first element of this list, and **omp_get_max_threads** retrieves the value of the first element of this list.
- The value of the *bind-var* ICV is a list. The runtime call **omp_get_proc_bind** retrieves the value of the first element of this list.
- Detailed values in the *place-partition-var* ICV are retrieved using the runtime calls **omp_get_partition_num_places**, **omp_get_partition_place_nums**, **omp_get_place_num_procs**, and **omp_get_place_proc_ids**.

Cross References

- **thread_limit** clause of the **teams** construct, see Section 2.7 on page 82.
- **omp_set_num_threads** routine, see Section 3.2.1 on page 334.
- **omp_get_max_threads** routine, see Section 3.2.3 on page 336.
- **omp_set_dynamic** routine, see Section 3.2.7 on page 340.
- **omp_get_dynamic** routine, see Section 3.2.8 on page 341.
- **omp_get_cancellation** routine, see Section 3.2.9 on page 342.
- **omp_set_nested** routine, see Section 3.2.10 on page 343.
- **omp_get_nested** routine, see Section 3.2.11 on page 344.
- **omp_set_schedule** routine, see Section 3.2.12 on page 345.
- **omp_get_schedule** routine, see Section 3.2.13 on page 347.

- 1 • `omp_get_thread_limit` routine, see Section 3.2.14 on page 348.
- 2 • `omp_get_supported_active_levels`, see Section 3.2.15 on page 349.
- 3 • `omp_set_max_active_levels` routine, see Section 3.2.16 on page 350.
- 4 • `omp_get_max_active_levels` routine, see Section 3.2.17 on page 351.
- 5 • `omp_get_level` routine, see Section 3.2.18 on page 352.
- 6 • `omp_get_active_level` routine, see Section 3.2.21 on page 355.
- 7 • `omp_get_proc_bind` routine, see Section 3.2.23 on page 357.
- 8 • `omp_get_place_num_procs` routine, see Section 3.2.25 on page 359.
- 9 • `omp_get_place_proc_ids` routine, see Section 3.2.26 on page 360.
- 10 • `omp_get_partition_num_places` routine, see Section 3.2.28 on page 362.
- 11 • `omp_get_partition_place_nums` routine, see Section 3.2.29 on page 363.
- 12 • `omp_set_affinity_format` routine, see Section 3.2.30 on page 364.
- 13 • `omp_get_affinity_format` routine, see Section 3.2.31 on page 366.
- 14 • `omp_set_default_device` routine, see Section 3.2.34 on page 369.
- 15 • `omp_get_default_device` routine, see Section 3.2.35 on page 370.
- 16 • `omp_get_max_task_priority` routine, see Section 3.2.42 on page 377.
- 17 • `omp_set_default_allocator` routine, see Section 3.7.4 on page 411.
- 18 • `omp_get_default_allocator` routine, see Section 3.7.5 on page 412.

19 2.5.4 How ICVs are Scoped

20 Table 2.3 shows the ICVs and their scope.

TABLE 2.3: Scopes of ICVs

ICV	Scope
<i>dyn-var</i>	data environment
<i>nthreads-var</i>	data environment

table continued on next page

table continued from previous page

ICV	Scope
<i>run-sched-var</i>	data environment
<i>def-sched-var</i>	device
<i>bind-var</i>	data environment
<i>stacksize-var</i>	device
<i>wait-policy-var</i>	device
<i>thread-limit-var</i>	data environment
<i>max-active-levels-var</i>	device
<i>active-levels-var</i>	data environment
<i>levels-var</i>	data environment
<i>place-partition-var</i>	implicit task
<i>cancel-var</i>	global
<i>display-affinity-var</i>	global
<i>affinity-format-var</i>	device
<i>default-device-var</i>	data environment
<i>target-offload-var</i>	global
<i>max-task-priority-var</i>	global
<i>tool-var</i>	global
<i>tool-libraries-var</i>	global
<i>debug-var</i>	global
<i>def-allocator-var</i>	implicit task

Description

- There is one copy per device of each ICV with device scope.
- Each data environment has its own copies of ICVs with data environment scope.
- Each implicit task has its own copy of ICVs with implicit task scope.

Calls to OpenMP API routines retrieve or modify data environment scoped ICVs in the data environment of their binding tasks.

1 **2.5.4.1 How the Per-Data Environment ICVs Work**

2 When a **task** construct or **parallel** construct is encountered, the generated task(s) inherit the
3 values of the data environment scoped ICVs from the generating task’s ICV values.

4 When a **parallel** construct is encountered, the value of each ICV with implicit task scope is
5 inherited, unless otherwise specified, from the implicit binding task of the generating task unless
6 otherwise specified.

7 When a **task** construct is encountered, the generated task inherits the value of *nthreads-var* from
8 the generating task’s *nthreads-var* value. When a **parallel** construct is encountered, and the
9 generating task’s *nthreads-var* list contains a single element, the generated task(s) inherit that list as
10 the value of *nthreads-var*. When a **parallel** construct is encountered, and the generating task’s
11 *nthreads-var* list contains multiple elements, the generated task(s) inherit the value of *nthreads-var*
12 as the list obtained by deletion of the first element from the generating task’s *nthreads-var* value.
13 The *bind-var* ICV is handled in the same way as the *nthreads-var* ICV.

14 When a *target task* executes a **target** region, the generated initial task uses the values of the data
15 environment scoped ICVs from the device data environment ICV values of the device that will
16 execute the region.

17 If a **teams** construct with a **thread_limit** clause is encountered, the *thread-limit-var* ICV
18 from the data environment of the initial task for each team is instead set to a value that is less than
19 or equal to the value specified in the clause.

20 When encountering a worksharing-loop region for which the **runtime** schedule kind is specified,
21 all implicit task regions that constitute the binding parallel region must have the same value for
22 *run-sched-var* in their data environments. Otherwise, the behavior is unspecified.

23 **2.5.5 ICV Override Relationships**

24 Table 2.4 shows the override relationships among construct clauses and ICVs.

TABLE 2.4: ICV Override Relationships

ICV	construct clause, if used
<i>dyn-var</i>	(none)
<i>nthreads-var</i>	num_threads
<i>run-sched-var</i>	schedule

table continued on next page

table continued from previous page

ICV	construct clause, if used
<i>def-sched-var</i>	schedule
<i>bind-var</i>	proc_bind
<i>stacksize-var</i>	(none)
<i>wait-policy-var</i>	(none)
<i>thread-limit-var</i>	(none)
<i>max-active-levels-var</i>	(none)
<i>active-levels-var</i>	(none)
<i>levels-var</i>	(none)
<i>place-partition-var</i>	(none)
<i>cancel-var</i>	(none)
<i>display-affinity-var</i>	(none)
<i>affinity-format-var</i>	(none)
<i>default-device-var</i>	(none)
<i>target-offload-var</i>	(none)
<i>max-task-priority-var</i>	(none)
<i>tool-var</i>	(none)
<i>tool-libraries-var</i>	(none)
<i>debug-var</i>	(none)
<i>def-allocator-var</i>	allocator

Description

- The **num_threads** clause overrides the value of the first element of the *nthreads-var* ICV.
- If a **schedule** clause specifies a modifier then that modifier overrides any modifier that is specified in the *run-sched-var* ICV.
- If *bind-var* is not set to *false* then the **proc_bind** clause overrides the value of the first element of the *bind-var* ICV; otherwise, the **proc_bind** clause has no effect.

Cross References

- **parallel** construct, see Section 2.6 on page 74.
- **proc_bind** clause, Section 2.6 on page 74.
- **num_threads** clause, see Section 2.6.1 on page 78.

- Worksharing-Loop construct, see Section 2.9.2 on page 101.
- **schedule** clause, see Section 2.9.2.1 on page 109.

2.6 parallel Construct

Summary

The parallel construct creates a team of OpenMP threads that execute the region.

Syntax

C / C++

The syntax of the **parallel** construct is as follows:

```
#pragma omp parallel [clause[ [, ] clause] ... ] new-line  
    structured-block
```

where *clause* is one of the following:

```
if ([parallel :] scalar-expression)  
num_threads (integer-expression)  
default (shared | none)  
private (list)  
firstprivate (list)  
shared (list)  
copyin (list)  
reduction ([reduction-modifier , ] reduction-identifier : list)  
proc_bind (master | close | spread)  
allocate ([allocator :] list)
```

C / C++

The syntax of the **parallel** construct is as follows:

```
!$omp parallel [clause [ , ] clause] ... ]
    structured-block
!$omp end parallel
```

where *clause* is one of the following:

```
if ([parallel :] scalar-logical-expression)
num_threads (scalar-integer-expression)
default (private | firstprivate | shared | none)
private (list)
firstprivate (list)
shared (list)
copyin (list)
reduction ([reduction-modifier , ] reduction-identifier : list)
proc_bind (master | close | spread)
allocate ([allocator :] list)
```

Binding

The binding thread set for a **parallel** region is the encountering thread. The encountering thread becomes the master thread of the new team.

Description

When a thread encounters a **parallel** construct, a team of threads is created to execute the **parallel** region (see Section 2.6.1 on page 78 for more information about how the number of threads in the team is determined, including the evaluation of the **if** and **num_threads** clauses). The thread that encountered the **parallel** construct becomes the master thread of the new team, with a thread number of zero for the duration of the new **parallel** region. All threads in the new team, including the master thread, execute the region. Once the team is created, the number of threads in the team remains constant for the duration of that **parallel** region.

The optional **proc_bind** clause, described in Section 2.6.2 on page 80, specifies the mapping of OpenMP threads to places within the current place partition, that is, within the places listed in the *place-partition-var* ICV for the implicit task of the encountering thread.

Within a **parallel** region, thread numbers uniquely identify each thread. Thread numbers are consecutive whole numbers ranging from zero for the master thread up to one less than the number

of threads in the team. A thread may obtain its own thread number by a call to the `omp_get_thread_num` library routine.

A set of implicit tasks, equal in number to the number of threads in the team, is generated by the encountering thread. The structured block of the **parallel** construct determines the code that will be executed in each implicit task. Each task is assigned to a different thread in the team and becomes tied. The task region of the task being executed by the encountering thread is suspended and each thread in the team executes its implicit task. Each thread can execute a path of statements that is different from that of the other threads.

The implementation may cause any thread to suspend execution of its implicit task at a task scheduling point, and to switch to execution of any explicit task generated by any of the threads in the team, before eventually resuming execution of the implicit task (for more details see Section 2.10 on page 135).

There is an implied barrier at the end of a **parallel** region. After the end of a **parallel** region, only the master thread of the team resumes execution of the enclosing task region.

If a thread in a team executing a **parallel** region encounters another **parallel** directive, it creates a new team, according to the rules in Section 2.6.1 on page 78, and it becomes the master of that new team.

If execution of a thread terminates while inside a **parallel** region, execution of all threads in all teams terminates. The order of termination of threads is unspecified. All work done by a team prior to any barrier that the team has passed in the program is guaranteed to be complete. The amount of work done by each thread after the last barrier that it passed and before it terminates is unspecified.

Execution Model Events

The *parallel-begin* event occurs in a thread that encounters a **parallel** construct before any implicit task is created for the corresponding **parallel** region.

Upon creation of each implicit task, an *implicit-task-begin* event occurs in the thread that executes the implicit task after the implicit task is fully initialized but before the thread begins to execute the structured block of the **parallel** construct.

If the **parallel** region creates a native thread, a *native-thread-begin* event occurs as the first event in the context of the new thread prior to the *implicit-task-begin* event.

Events associated with implicit barriers occur at the end of a **parallel** region. Section 2.17.3 describes events associated with implicit barriers.

When a thread finishes an implicit task, an *implicit-task-end* event occurs in the thread after events associated with implicit barrier synchronization in the implicit task.

The *parallel-end* event occurs in the thread that encounters the **parallel** construct after the thread executes its *implicit-task-end* event but before the thread resumes execution of the encountering task.

If a native thread is destroyed at the end of a **parallel** region, a *native thread-end* event occurs in the thread as the last event prior to destruction of the thread.

Tool Callbacks

A thread dispatches a registered **ompt_callback_parallel_begin** callback for each occurrence of a *parallel-begin* event in that thread. The callback occurs in the task that encounters the **parallel** construct. This callback has the type signature **ompt_callback_parallel_begin_t**. In the dispatched callback, *(flags & ompt_parallel_team)* evaluates to *true*.

A thread dispatches a registered **ompt_callback_implicit_task** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of an *implicit-task-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_implicit_task** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of an *implicit-task-end* event in that thread. The callbacks occur in the context of the implicit task and have type signature **ompt_callback_implicit_task_t**. In the dispatched callback, *(flags & ompt_task_implicit)* evaluates to *true*.

A thread dispatches a registered **ompt_callback_parallel_end** callback for each occurrence of a *parallel-end* event in that thread. The callback occurs in the task that encounters the **parallel** construct. This callback has the type signature **ompt_callback_parallel_end_t**.

A thread dispatches a registered **ompt_callback_thread_begin** callback for the *native-thread-begin* event in that thread. The callback occurs in the context of the thread. The callback has type signature **ompt_callback_thread_begin_t**.

A thread dispatches a registered **ompt_callback_thread_end** callback for the *native-thread-end* event in that thread. The callback occurs in the context of the thread. The callback has type signature **ompt_callback_thread_end_t**.

Restrictions

Restrictions to the **parallel** construct are as follows:

- A program that branches into or out of a **parallel** region is non-conforming.
- A program must not depend on any ordering of the evaluations of the clauses of the **parallel** directive, or on any side effects of the evaluations of the clauses.
- At most one **if** clause can appear on the directive.
- At most one **proc_bind** clause can appear on the directive.
- At most one **num_threads** clause can appear on the directive. The **num_threads** expression must evaluate to a positive integer value.

- A **throw** executed inside a **parallel** region must cause execution to resume within the same **parallel** region, and the same thread that threw the exception must catch it.

Cross References

- OpenMP execution model, see Section 1.3 on page 20.
- **num_threads** clause, see Section 2.6 on page 74.
- **proc_bind** clause, see Section 2.6.2 on page 80.
- **allocate** clause, see Section 2.11.4 on page 158.
- **if** clause, see Section 2.15 on page 220.
- **default**, **shared**, **private**, **firstprivate**, and **reduction** clauses, see Section 2.19.4 on page 282.
- **copyin** clause, see Section 2.19.6 on page 309.
- **omp_get_thread_num** routine, see Section 3.2.4 on page 337.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.4.4.11 on page 443.
- **ompt_callback_thread_begin_t**, see Section 4.5.2.1 on page 459.
- **ompt_callback_thread_end_t**, see Section 4.5.2.2 on page 460.
- **ompt_callback_parallel_begin_t**, see Section 4.5.2.3 on page 461.
- **ompt_callback_parallel_end_t**, see Section 4.5.2.4 on page 463.
- **ompt_callback_implicit_task_t**, see Section 4.5.2.11 on page 471.

2.6.1 Determining the Number of Threads for a **parallel** Region

When execution encounters a **parallel** directive, the value of the **if** clause or **num_threads** clause (if any) on the directive, the current parallel context, and the values of the *nthreads-var*, *dyn-var*, *thread-limit-var*, and *max-active-levels-var* ICVs are used to determine the number of threads to use in the region.

Using a variable in an **if** or **num_threads** clause expression of a **parallel** construct causes an implicit reference to the variable in all enclosing constructs. The **if** clause expression and the **num_threads** clause expression are evaluated in the context outside of the **parallel** construct,

and no ordering of those evaluations is specified. In what order or how many times any side effects of the evaluation of the **num_threads** or **if** clause expressions occur is also unspecified.

When a thread encounters a **parallel** construct, the number of threads is determined according to Algorithm 2.1.

Algorithm 2.1

```
let ThreadsBusy be the number of OpenMP threads currently executing in this contention group;
let ActiveParRegions be the number of enclosing active parallel regions;
if an if clause exists
then let IfClauseValue be the value of the if clause expression;
else let IfClauseValue = true;
if a num_threads clause exists
then let ThreadsRequested be the value of the num_threads clause expression;
else let ThreadsRequested = value of the first element of nthreads-var;
let ThreadsAvailable = (thread-limit-var - ThreadsBusy + 1);
if (IfClauseValue = false)
then number of threads = 1;
else if (ActiveParRegions = max-active-levels-var)
then number of threads = 1;
else if (dyn-var = true) and (ThreadsRequested ≤ ThreadsAvailable)
then 1 ≤ number of threads ≤ ThreadsRequested;
else if (dyn-var = true) and (ThreadsRequested > ThreadsAvailable)
then 1 ≤ number of threads ≤ ThreadsAvailable;
else if (dyn-var = false) and (ThreadsRequested ≤ ThreadsAvailable)
then number of threads = ThreadsRequested;
else if (dyn-var = false) and (ThreadsRequested > ThreadsAvailable)
then behavior is implementation defined;
```

Note – Since the initial value of the *dyn-var* ICV is implementation defined, programs that depend on a specific number of threads for correct execution should explicitly disable dynamic adjustment of the number of threads.

Cross References

- *nthreads-var*, *dyn-var*, *thread-limit-var*, and *max-active-levels-var* ICVs, see Section 2.5 on page 63.
- **parallel** construct, see Section 2.6 on page 74.
- **num_threads** clause, see Section 2.6 on page 74.
- **if** clause, see Section 2.15 on page 220.

2.6.2 Controlling OpenMP Thread Affinity

When a thread encounters a **parallel** directive without a **proc_bind** clause, the *bind-var* ICV is used to determine the policy for assigning OpenMP threads to places within the current place partition, that is, within the places listed in the *place-partition-var* ICV for the implicit task of the encountering thread. If the **parallel** directive has a **proc_bind** clause then the binding policy specified by the **proc_bind** clause overrides the policy specified by the first element of the *bind-var* ICV. Once a thread in the team is assigned to a place, the OpenMP implementation should not move it to another place.

The **master** thread affinity policy instructs the execution environment to assign every thread in the team to the same place as the master thread. The place partition is not changed by this policy, and each implicit task inherits the *place-partition-var* ICV of the parent implicit task.

The **close** thread affinity policy instructs the execution environment to assign the threads in the team to places close to the place of the parent thread. The place partition is not changed by this policy, and each implicit task inherits the *place-partition-var* ICV of the parent implicit task. If T is the number of threads in the team, and P is the number of places in the parent's place partition, then the assignment of threads in the team to places is as follows:

- $T \leq P$: The master thread executes on the place of the parent thread. The thread with the next smallest thread number executes on the next place in the place partition, and so on, with wrap around with respect to the place partition of the master thread.

- $T > P$: Each place p will contain S_p threads with consecutive thread numbers where $\lfloor T/P \rfloor \leq S_p \leq \lceil T/P \rceil$. The first S_0 threads (including the master thread) are assigned to the place of the parent thread. The next S_1 threads are assigned to the next place in the place partition, and so on, with wrap around with respect to the place partition of the master thread. When P does not divide T evenly, the exact number of threads in a particular place is implementation defined.

The purpose of the **spread** thread affinity policy is to create a sparse distribution for a team of T threads among the P places of the parent's place partition. A sparse distribution is achieved by first subdividing the parent partition into T subpartitions if $T \leq P$, or P subpartitions if $T > P$. Then one thread ($T \leq P$) or a set of threads ($T > P$) is assigned to each subpartition. The *place-partition-var* ICV of each implicit task is set to its subpartition. The subpartitioning is not only a mechanism for achieving a sparse distribution, it also defines a subset of places for a thread to use when creating a nested **parallel** region. The assignment of threads to places is as follows:

- $T \leq P$: The parent thread's place partition is split into T subpartitions, where each subpartition contains $\lfloor P/T \rfloor$ or $\lceil P/T \rceil$ consecutive places. A single thread is assigned to each subpartition. The master thread executes on the place of the parent thread and is assigned to the subpartition that includes that place. The thread with the next smallest thread number is assigned to the first place in the next subpartition, and so on, with wrap around with respect to the original place partition of the master thread.
- $T > P$: The parent thread's place partition is split into P subpartitions, each consisting of a single place. Each subpartition is assigned S_p threads with consecutive thread numbers, where $\lfloor T/P \rfloor \leq S_p \leq \lceil T/P \rceil$. The first S_0 threads (including the master thread) are assigned to the subpartition containing the place of the parent thread. The next S_1 threads are assigned to the next subpartition, and so on, with wrap around with respect to the original place partition of the master thread. When P does not divide T evenly, the exact number of threads in a particular subpartition is implementation defined.

The determination of whether the affinity request can be fulfilled is implementation defined. If the affinity request cannot be fulfilled, then the affinity of threads in the team is implementation defined.

Note – Wrap around is needed if the end of a place partition is reached before all thread assignments are done. For example, wrap around may be needed in the case of **close** and $T \leq P$, if the master thread is assigned to a place other than the first place in the place partition. In this case, thread 1 is assigned to the place after the place of the master place, thread 2 is assigned to the place after that, and so on. The end of the place partition may be reached before all threads are assigned. In this case, assignment of threads is resumed with the first place in the place partition.

1 2.7 teams Construct

2 Summary

3 The **teams** construct creates a league of initial teams and the initial thread in each team executes
4 the region.

5 Syntax

C / C++

6 The syntax of the **teams** construct is as follows:

```
7 #pragma omp teams [clause [ , ] clause] ... ] new-line  
8 structured-block
```

9 where *clause* is one of the following:

```
10 num_teams (integer-expression)  
11 thread_limit (integer-expression)  
12 default (shared | none)  
13 private (list)  
14 firstprivate (list)  
15 shared (list)  
16 reduction ([default , ] reduction-identifier : list)  
17 allocate ([allocator : ] list)
```

C / C++

Fortran

18 The syntax of the **teams** construct is as follows:

```
19 !$omp teams [clause [ , ] clause] ... ]  
20 structured-block  
21 !$omp end teams
```

where *clause* is one of the following:

```
num_teams (scalar-integer-expression)  
thread_limit (scalar-integer-expression)  
default (shared | firstprivate | private | none)  
private (list)  
firstprivate (list)  
shared (list)  
reduction ([default ,] reduction-identifier : list)  
allocate ([allocator :] list)
```

Fortran

Binding

The binding thread set for a **teams** region is the encountering thread.

Description

When a thread encounters a **teams** construct, a league of teams is created. Each team is an initial team, and the initial thread in each team executes the **teams** region.

The number of teams created is implementation defined, but is less than or equal to the value specified in the **num_teams** clause. A thread may obtain the number of initial teams created by the construct by a call to the **omp_get_num_teams** routine.

The maximum number of threads participating in the contention group that each team initiates is implementation defined, but is less than or equal to the value specified in the **thread_limit** clause.

On a combined or composite construct that includes **target** and **teams** constructs, the expressions in **num_teams** and **thread_limit** clauses are evaluated on the host device on entry to the **target** construct.

Once the teams are created, the number of initial teams remains constant for the duration of the **teams** region.

Within a **teams** region, initial team numbers uniquely identify each initial team. Initial team numbers are consecutive whole numbers ranging from zero to one less than the number of initial teams. A thread may obtain its own initial team number by a call to the **omp_get_team_num** library routine. The policy for assigning the initial threads to places is implementation defined. The **teams** construct sets the *place-partition-var* and *default-device-var* ICVs for each initial thread to an implementation-defined value.

After the teams have completed execution of the **teams** region, the encountering task resumes execution of the enclosing task region.

Execution Model Events

The *teams-begin* event occurs in a thread that encounters a **teams** construct before any initial task is created for the corresponding **teams** region.

Upon creation of each initial task, an *initial-task-begin* event occurs in the thread that executes the initial task after the initial task is fully initialized but before the thread begins to execute the structured block of the **teams** construct.

If the **teams** region creates a native thread, a *native-thread-begin* event occurs as the first event in the context of the new thread prior to the *initial-task-begin* event.

When a thread finishes an initial task, an *initial-task-end* event occurs in the thread.

The *teams-end* event occurs in the thread that encounters the **teams** construct after the thread executes its *initial-task-end* event but before it resumes execution of the encountering task.

If a native thread is destroyed at the end of a **teams** region, a *native-thread-end* event occurs in the thread as the last event prior to destruction of the thread.

Tool Callbacks

A thread dispatches a registered **ompt_callback_parallel_begin** callback for each occurrence of a *teams-begin* event in that thread. The callback occurs in the task that encounters the **teams** construct. This callback has the type signature **ompt_callback_parallel_begin_t**. In the dispatched callback, *(flags & ompt_parallel_league)* evaluates to *true*.

A thread dispatches a registered **ompt_callback_implicit_task** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of an *initial-task-begin* in that thread. Similarly, a thread dispatches a registered **ompt_callback_implicit_task** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of an *initial-task-end* event in that thread. The callbacks occur in the context of the initial task and have type signature **ompt_callback_implicit_task_t**. In the dispatched callback, *(flags & ompt_task_initial)* evaluates to *true*.

A thread dispatches a registered **ompt_callback_parallel_end** callback for each occurrence of a *teams-end* event in that thread. The callback occurs in the task that encounters the **teams** construct. This callback has the type signature **ompt_callback_parallel_end_t**.

A thread dispatches a registered **ompt_callback_thread_begin** callback for the *native-thread-begin* event in that thread. The callback occurs in the context of the thread. The callback has type signature **ompt_callback_thread_begin_t**.

A thread dispatches a registered **ompt_callback_thread_end** callback for the *native-thread-end* event in that thread. The callback occurs in the context of the thread. The callback has type signature **ompt_callback_thread_end_t**.

Restrictions

Restrictions to the **teams** construct are as follows:

- A program that branches into or out of a **teams** region is non-conforming.
- A program must not depend on any ordering of the evaluations of the clauses of the **teams** directive, or on any side effects of the evaluation of the clauses.
- At most one **thread_limit** clause can appear on the directive. The **thread_limit** expression must evaluate to a positive integer value.
- At most one **num_teams** clause can appear on the directive. The **num_teams** expression must evaluate to a positive integer value.
- A **teams** region can only be strictly nested within the implicit parallel region or a **target** region. If a **teams** construct is nested within a **target** construct, that **target** construct must contain no statements, declarations or directives outside of the **teams** construct.
- **distribute**, **distribute simd**, distribute parallel worksharing-loop, distribute parallel worksharing-loop SIMD, **parallel** regions, including any **parallel** regions arising from combined constructs, **omp_get_num_teams()** regions, and **omp_get_team_num()** regions are the only OpenMP regions that may be strictly nested inside the **teams** region.

Cross References

- **parallel** construct, see Section 2.6 on page 74.
- **distribute** construct, see Section 2.9.4.1 on page 120.
- **distribute simd** construct, see Section 2.9.4.2 on page 123.
- **allocate** clause, see Section 2.11.4 on page 158.
- **target** construct, see Section 2.12.5 on page 170.
- **default**, **shared**, **private**, **firstprivate**, and **reduction** clauses, see Section 2.19.4 on page 282.
- **omp_get_num_teams** routine, see Section 3.2.38 on page 373.
- **omp_get_team_num** routine, see Section 3.2.39 on page 374.
- **ompt_callback_thread_begin_t**, see Section 4.5.2.1 on page 459.
- **ompt_callback_thread_end_t**, see Section 4.5.2.2 on page 460.
- **ompt_callback_parallel_begin_t**, see Section 4.5.2.3 on page 461.
- **ompt_callback_parallel_end_t**, see Section 4.5.2.4 on page 463.
- **ompt_callback_implicit_task_t**, see Section 4.5.2.11 on page 471.

1 2.8 Worksharing Constructs

2 A worksharing construct distributes the execution of the corresponding region among the members
3 of the team that encounters it. Threads execute portions of the region in the context of the implicit
4 tasks that each one is executing. If the team consists of only one thread then the worksharing region
5 is not executed in parallel.

6 A worksharing region has no barrier on entry; however, an implied barrier exists at the end of the
7 worksharing region, unless a **nowait** clause is specified. If a **nowait** clause is present, an
8 implementation may omit the barrier at the end of the worksharing region. In this case, threads that
9 finish early may proceed straight to the instructions that follow the worksharing region without
10 waiting for the other members of the team to finish the worksharing region, and without performing
11 a flush operation.

12 The OpenMP API defines the worksharing constructs that are described in this section as well as
13 the worksharing-loop construct, which is described in Section 2.9.2 on page 101.

14 Restrictions

15 The following restrictions apply to worksharing constructs:

- 16 • Each worksharing region must be encountered by all threads in a team or by none at all, unless
17 cancellation has been requested for the innermost enclosing parallel region.
- 18 • The sequence of worksharing regions and **barrier** regions encountered must be the same for
19 every thread in a team.

20 2.8.1 sections Construct

21 Summary

22 The **sections** construct is a non-iterative worksharing construct that contains a set of structured
23 blocks that are to be distributed among and executed by the threads in a team. Each structured
24 block is executed once by one of the threads in the team in the context of its implicit task.

Syntax

C / C++

The syntax of the **sections** construct is as follows:

```
#pragma omp sections [clause [ , ] clause] ... ] new-line
{
  [#pragma omp section new-line]
    structured-block
  [#pragma omp section new-line]
    structured-block]
  ...
}
```

where *clause* is one of the following:

```
private (list)
firstprivate (list)
lastprivate ([ lastprivate-modifier : ] list)
reduction ([reduction-modifier , ] reduction-identifier : list)
allocate ([allocator : ] list)
nowait
```

C / C++

Fortran

The syntax of the **sections** construct is as follows:

```
!$omp sections [clause [ , ] clause] ... ]
  [!$omp section]
    structured-block
  [!$omp section]
    structured-block]
  ...
!$omp end sections [nowait]
```

where *clause* is one of the following:

```
private (list)
firstprivate (list)
lastprivate ([ lastprivate-modifier : ] list)
reduction ([reduction-modifier , ] reduction-identifier : list)
allocate ([allocator : ] list)
```

Fortran

Binding

The binding thread set for a **sections** region is the current team. A **sections** region binds to the innermost enclosing **parallel** region. Only the threads of the team that executes the binding **parallel** region participate in the execution of the structured blocks and the implied barrier of the **sections** region if the barrier is not eliminated by a **nowait** clause.

Description

Each structured block in the **sections** construct is preceded by a **section** directive except possibly the first block, for which a preceding **section** directive is optional.

The method of scheduling the structured blocks among the threads in the team is implementation defined.

There is an implicit barrier at the end of a **sections** construct unless a **nowait** clause is specified.

Execution Model Events

The *section-begin* event occurs after an implicit task encounters a **sections** construct but before the task executes any structured block of the **sections** region.

The *sections-end* event occurs after an implicit task finishes execution of a **sections** region but before it resumes execution of the enclosing context.

The *section-begin* event occurs before an implicit task starts to execute a structured block in the **sections** construct for each of those structured blocks that the task executes.

Tool Callbacks

A thread dispatches a registered **ompt_callback_work** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_work_sections** as its *wstype* argument for each occurrence of a *section-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_work** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_work_sections** as its *wstype* argument for each occurrence of a *sections-end* event in that thread. The callbacks occur in the context of the implicit task. The callbacks have type signature **ompt_callback_work_t**.

A thread dispatches a registered **ompt_callback_dispatch** callback for each occurrence of a *section-begin* event in that thread. The callback occurs in the context of the implicit task. The callback has type signature **ompt_callback_dispatch_t**.

Restrictions

Restrictions to the **sections** construct are as follows:

- Orphaned **section** directives are prohibited. That is, the **section** directives must appear within the **sections** construct and must not be encountered elsewhere in the **sections** region.
- The code enclosed in a **sections** construct must be a structured block.
- Only a single **nowait** clause can appear on a **sections** directive.

C++

- A throw executed inside a **sections** region must cause execution to resume within the same section of the **sections** region, and the same thread that threw the exception must catch it.

C++

Cross References

- **allocate** clause, see Section 2.11.4 on page 158.
- **private**, **firstprivate**, **lastprivate**, and **reduction** clauses, see Section 2.19.4 on page 282.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.4.4.11 on page 443.
- **ompt_work_sections**, see Section 4.4.4.15 on page 445.
- **ompt_callback_work_t**, see Section 4.5.2.5 on page 464.
- **ompt_callback_dispatch_t**, see Section 4.5.2.6 on page 465.

2.8.2 single Construct

Summary

The **single** construct specifies that the associated structured block is executed by only one of the threads in the team (not necessarily the master thread), in the context of its implicit task. The other threads in the team, which do not execute the block, wait at an implicit barrier at the end of the **single** construct unless a **nowait** clause is specified.

Syntax

C / C++

The syntax of the single construct is as follows:

```
#pragma omp single [clause[ [, ] clause] ... ] new-line  
    structured-block
```

where *clause* is one of the following:

```
private(list)  
firstprivate(list)  
copyprivate(list)  
allocate([allocator :] list)  
nowait
```

C / C++

Fortran

The syntax of the **single** construct is as follows:

```
!$omp single [clause[ [, ] clause] ... ]  
    structured-block  
!$omp end single [end_clause[ [, ] end_clause] ... ]
```

where *clause* is one of the following:

```
private(list)  
firstprivate(list)  
allocate([allocator :] list)
```

and *end_clause* is one of the following:

```
copyprivate(list)  
nowait
```

Fortran

Binding

The binding thread set for a **single** region is the current team. A **single** region binds to the innermost enclosing **parallel** region. Only the threads of the team that executes the binding **parallel** region participate in the execution of the structured block and the implied barrier of the **single** region if the barrier is not eliminated by a **nowait** clause.

1 **Description**

2 Only one of the encountering threads will execute the structured block associated with the **single**
3 construct. The method of choosing a thread to execute the structured block each time the team
4 encounters the construct is implementation defined. There is an implicit barrier at the end of the
5 **single** construct unless a **nowait** clause is specified.

6 **Execution Model Events**

7 The *single-begin* event occurs after an **implicit task** encounters a **single** construct but
8 before the task starts to execute the structured block of the **single** region.

9 The *single-end* event occurs after an implicit task finishes execution of a **single** region but before
10 it resumes execution of the enclosing region.

11 **Tool Callbacks**

12 A thread dispatches a registered **ompt_callback_work** callback with **ompt_scope_begin**
13 as its *endpoint* argument for each occurrence of a *single-begin* event in that thread. Similarly, a
14 thread dispatches a registered **ompt_callback_work** callback with **ompt_scope_begin** as
15 its *endpoint* argument for each occurrence of a *single-end* event in that thread. For each of these
16 callbacks, the *wstype* argument is **ompt_work_single_executor** if the thread executes the
17 structured block associated with the **single** region; otherwise, the *wstype* argument is
18 **ompt_work_single_other**. The callback has type signature **ompt_callback_work_t**.

19 **Restrictions**

20 Restrictions to the **single** construct are as follows:

- 21
 - The **copyprivate** clause must not be used with the **nowait** clause.
 - At most one **nowait** clause can appear on a **single** construct.

23

- A throw executed inside a **single** region must cause execution to resume within the same
24 **single** region, and the same thread that threw the exception must catch it.

C++

Cross References

- **allocate** clause, see Section 2.11.4 on page 158.
- **private** and **firstprivate** clauses, see Section 2.19.4 on page 282.
- **copyprivate** clause, see Section 2.19.6.2 on page 312.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.4.4.11 on page 443.
- **ompt_work_single_executor** and **ompt_work_single_other**, see Section 4.4.4.15 on page 445.
- **ompt_callback_work_t**, Section 4.5.2.5 on page 464.

Fortran

2.8.3 workshare Construct

Summary

The **workshare** construct divides the execution of the enclosed structured block into separate units of work, and causes the threads of the team to share the work such that each unit is executed only once by one thread, in the context of its implicit task.

Syntax

The syntax of the **workshare** construct is as follows:

```
!$omp workshare
    structured-block
!$omp end workshare [nowait/]
```

Binding

The binding thread set for a **workshare** region is the current team. A **workshare** region binds to the innermost enclosing **parallel** region. Only the threads of the team that executes the binding **parallel** region participate in the execution of the units of work and the implied barrier of the **workshare** region if the barrier is not eliminated by a **nowait** clause.

Description

There is an implicit barrier at the end of a **workshare** construct unless a **nowait** clause is specified.

An implementation of the **workshare** construct must insert any synchronization that is required to maintain standard Fortran semantics. For example, the effects of one statement within the structured block must appear to occur before the execution of succeeding statements, and the evaluation of the right hand side of an assignment must appear to complete prior to the effects of assigning to the left hand side.

The statements in the **workshare** construct are divided into units of work as follows:

- For array expressions within each statement, including transformational array intrinsic functions that compute scalar values from arrays:
 - Evaluation of each element of the array expression, including any references to **ELEMENTAL** functions, is a unit of work.
 - Evaluation of transformational array intrinsic functions may be freely subdivided into any number of units of work.
- For an array assignment statement, the assignment of each element is a unit of work.
- For a scalar assignment statement, the assignment operation is a unit of work.
- For a **WHERE** statement or construct, the evaluation of the mask expression and the masked assignments are each a unit of work.
- For a **FORALL** statement or construct, the evaluation of the mask expression, expressions occurring in the specification of the iteration space, and the masked assignments are each a unit of work.
- For an **atomic** construct, the atomic operation on the storage location designated as x is a unit of work.
- For a **critical** construct, the construct is a single unit of work.
- For a **parallel** construct, the construct is a unit of work with respect to the **workshare** construct. The statements contained in the **parallel** construct are executed by a new thread team.
- If none of the rules above apply to a portion of a statement in the structured block, then that portion is a unit of work.

The transformational array intrinsic functions are **MATMUL**, **DOT_PRODUCT**, **SUM**, **PRODUCT**, **MAXVAL**, **MINVAL**, **COUNT**, **ANY**, **ALL**, **SPREAD**, **PACK**, **UNPACK**, **RESHAPE**, **TRANSPOSE**, **EOSHIFT**, **CSHIFT**, **MINLOC**, and **MAXLOC**.

It is unspecified how the units of work are assigned to the threads executing a **workshare** region.

If an array expression in the block references the value, association status, or allocation status of private variables, the value of the expression is undefined, unless the same value would be computed by every thread.

If an array assignment, a scalar assignment, a masked array assignment, or a **FORALL** assignment assigns to a private variable in the block, the result is unspecified.

The **workshare** directive causes the sharing of work to occur only in the **workshare** construct, and not in the remainder of the **workshare** region.

Execution Model Events

The *workshare-begin* event occurs after an implicit task encounters a **workshare** construct but before the task starts to execute the structured block of the **workshare** region.

The *workshare-end* event occurs after an implicit task finishes execution of a **workshare** region but before it resumes execution of the enclosing context.

Tool Callbacks

A thread dispatches a registered **ompt_callback_work** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_work_workshare** as its *wstype* argument for each occurrence of a *workshare-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_work** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_work_workshare** as its *wstype* argument for each occurrence of a *workshare-end* event in that thread. The callbacks occur in the context of the implicit task. The callbacks have type signature **ompt_callback_work_t**.

Restrictions

The following restrictions apply to the **workshare** construct:

- The only OpenMP constructs that may be closely nested inside a **workshare** construct are the **atomic**, **critical**, and **parallel** constructs.
- Base language statements that are encountered inside a **workshare** construct but that are not enclosed within a **parallel** construct that is nested inside the **workshare** construct must consist of only the following:
 - array assignments
 - scalar assignments
 - **FORALL** statements
 - **FORALL** constructs
 - **WHERE** statements

- 1 – **WHERE** constructs
- 2 • All array assignments, scalar assignments, and masked array assignments that are encountered
- 3 inside a **workshare** construct but are not nested inside a **parallel** construct that is nested
- 4 inside the **workshare** construct must be intrinsic assignments.
- 5 • The construct must not contain any user defined function calls unless the function is
- 6 **ELEMENTAL** or the function call is contained inside a **parallel** construct that is nested inside
- 7 the **workshare** construct.

8 Cross References

- 9 • **parallel** construct, see Section 2.6 on page 74.
- 10 • **critical** construct, see Section 2.17.1 on page 223.
- 11 • **atomic** construct, see Section 2.17.7 on page 234.
- 12 • **ompt_scope_begin** and **ompt_scope_end**, see Section 4.4.4.11 on page 443.
- 13 • **ompt_work_workshare**, see Section 4.4.4.15 on page 445.
- 14 • **ompt_callback_work_t**, see Section 4.5.2.5 on page 464.

Fortran

15 2.9 Loop-Related Directives

16 2.9.1 Canonical Loop Form

C / C++

17 The loops associated with a loop-associated directive have *canonical loop form* if they conform to

18 the following:

for (*init-expr*; *test-expr*; *incr-expr*) *structured-block*

<i>init-expr</i>	One of the following:
	<i>var</i> = <i>lb</i>
	<i>integer-type var</i> = <i>lb</i>
	<i>random-access-iterator-type var</i> = <i>lb</i>
	<i>pointer-type var</i> = <i>lb</i>

continued on next page

continued from previous page

<i>test-expr</i>	One of the following: <i>var relational-op b</i> <i>b relational-op var</i>
<i>incr-expr</i>	One of the following: $++var$ $var++$ $--var$ $var--$ $var += incr$ $var -= incr$ $var = var + incr$ $var = incr + var$ $var = var - incr$
<i>var</i>	One of the following: A variable of a signed or unsigned integer type. For C++, a variable of a random access iterator type. For C, a variable of a pointer type. This variable must not be modified during the execution of the <i>for-loop</i> other than in <i>incr-expr</i> .
<i>relational-op</i>	One of the following: $<$ $<=$ $>$ $>=$ $!=$
<i>lb</i> and <i>b</i>	Expressions of a type compatible with the type of <i>var</i> that are loop invariant with respect to the outermost associated loop or are one of the following (where <i>var-outer</i> , <i>a1</i> , and <i>a2</i> have a type compatible with the type of <i>var</i> , <i>var-outer</i> is <i>var</i> from an outer associated loop, and <i>a1</i> and <i>a2</i> are loop invariant integer expressions with respect to the outermost loop):

continued on next page

continued from previous page

var-outer
var-outer + *a2*
a2 + *var-outer*
var-outer - *a2*
a2 - *var-outer*
a1 * *var-outer*
a1 * *var-outer* + *a2*
a2 + *a1* * *var-outer*
a1 * *var-outer* - *a2*
a2 - *a1* * *var-outer*
var-outer * *a1*
var-outer * *a1* + *a2*
a2 + *var-outer* * *a1*
var-outer * *a1* - *a2*
a2 - *var-outer* * *a1*

incr An integer expression that is loop invariant with respect to the outermost associated loop.

▲  C / C++  ▲

▼  Fortran  ▼

- 1 The loops associated with a loop-associated directive have *canonical loop form* if each of them is a
2 *do-loop* that is a *do-construct* or an *inner-shared-do-construct* as defined by the Fortran standard. If
3 an **end do** directive follows a *do-construct* in which several loop statements share a **DO** termination
4 statement, then the directive can only be specified for the outermost of these **DO** statements.

The *do-stmt* for any *do-loop* must conform to the following:

DO [label] var = lb , b [, incr]	
<i>var</i>	A variable of integer type.
<i>lb</i> and <i>b</i>	Expressions of a type compatible with the type of <i>var</i> that are loop invariant with respect to the outermost associated loop or are one of the following (where <i>var-outer</i> , <i>a1</i> , and <i>a2</i> have a type compatible with the type of <i>var</i> , <i>var-outer</i> is <i>var</i> from an outer associated loop, and <i>a1</i> and <i>a2</i> are loop invariant integer expressions with respect to the outermost loop): <i>var-outer</i> <i>var-outer</i> + <i>a2</i> <i>a2</i> + <i>var-outer</i> <i>var-outer</i> - <i>a2</i> <i>a2</i> - <i>var-outer</i> <i>a1</i> * <i>var-outer</i> <i>a1</i> * <i>var-outer</i> + <i>a2</i> <i>a2</i> + <i>a1</i> * <i>var-outer</i> <i>a1</i> * <i>var-outer</i> - <i>a2</i> <i>a2</i> - <i>a1</i> * <i>var-outer</i> <i>var-outer</i> * <i>a1</i> <i>var-outer</i> * <i>a1</i> + <i>a2</i> <i>a2</i> + <i>var-outer</i> * <i>a1</i> <i>var-outer</i> * <i>a1</i> - <i>a2</i> <i>a2</i> - <i>var-outer</i> * <i>a1</i>
<i>incr</i>	An integer expression that is loop invariant with respect to the outermost associated loop. If it is not explicitly specified, its value is assumed to be 1.

▲ Fortran ▲

- 2 The canonical form allows the iteration count of all associated loops to be computed before
3 executing the outermost loop. The *incr* and *range-expr* are evaluated before executing the
4 loop-associated construct. If *b* or *lb* is loop invariant with respect to the outermost associated loop,
5 it is evaluated before executing the loop-associated construct. If *b* or *lb* is not loop invariant with
6 respect to the outermost associated loop, *a1* and/or *a2* are evaluated before executing the
7 loop-associated construct. The computation is performed for each loop in an integer type. This type
8 is derived from the type of *var* as follows:
- If *var* is of an integer type, then the type is the type of *var*.
- 9

C++

- If *var* is of a random access iterator type, then the type is the type that would be used by *std::distance* applied to variables of the type of *var*.

C++

C

- If *var* is of a pointer type, then the type is **ptrdiff_t**.

C

The behavior is unspecified if any intermediate result required to compute the iteration count cannot be represented in the type determined above.

There is no implied synchronization during the evaluation of the *lb*, *b*, or *incr* expressions. It is unspecified whether, in what order, or how many times any side effects within the *lb*, *b*, or *incr* expressions occur.

Note – Random access iterators are required to support random access to elements in constant time. Other iterators are precluded by the restrictions since they can take linear time or offer limited functionality. The use of tasks to parallelize those cases is therefore advisable.

C++

A range-based for loop that is valid in the base language and has a begin value that satisfies the random access iterator requirement has *canonical loop form*. Range-based for loops are of the following form:

for (*range-decl*: *range-expr*) *structured-block*

The *begin-expr* and *end-expr* expressions are derived from *range-expr* by the base language and assigned to variables to which this specification refers as **__begin** and **__end** respectively. Both **__begin** and **__end** are privatized. For the purpose of the rest of the standard **__begin** is the iteration variable of the range-for loop.

C++

Restrictions

The following restrictions also apply:

C / C++

- If *test-expr* is of the form *var relational-op b* and *relational-op* is < or <= then *incr-expr* must cause *var* to increase on each iteration of the loop. If *test-expr* is of the form *var relational-op b* and *relational-op* is > or >= then *incr-expr* must cause *var* to decrease on each iteration of the loop.
- If *test-expr* is of the form *b relational-op var* and *relational-op* is < or <= then *incr-expr* must cause *var* to decrease on each iteration of the loop. If *test-expr* is of the form *b relational-op var* and *relational-op* is > or >= then *incr-expr* must cause *var* to increase on each iteration of the loop.
- If *test-expr* is of the form *b != var* or *var != b* then *incr-expr* must cause *var* either to increase on each iteration of the loop or to decrease on each iteration of the loop.
- If *relational-op* is != and *incr-expr* is of the form that has *incr* then *incr* must be a constant expression and evaluate to -1 or 1.

C / C++

C++

- In the **simd** construct the only random access iterator types that are allowed for *var* are pointer types.
 - The *range-expr* of a range-for loop must be loop invariant with respect to the outermost associated loop, and must not reference iteration variables of any associated loops.
 - The loops associated with an ordered clause with a parameter may not include range-for loops.
- C++
- The *b*, *lb*, *incr*, and *range-expr* expressions may not reference any *var* or member of the *range-decl* of any enclosed associated loop.
 - For any associated loop where the *b* or *lb* expression is not loop invariant with respect to the outermost loop, the *var-outer* that appears in the expression may not have a random access iterator type.
 - For any associated loop where *b* or *lb* is not loop invariant with respect to the outermost loop, the expression $b - lb$ will have the form $c * var-outer + d$, where *c* and *d* are loop invariant integer expressions. Let *incr-outer* be the *incr* expression of the outer loop referred to by *var-outer*. The value of $c * incr-outer \bmod incr$ must be 0.

Cross References

- **simd** construct, see Section 2.9.3.1 on page 110.
- **lastprivate** clause, see Section 2.19.4.5 on page 288.
- **linear** clause, see Section 2.19.4.6 on page 290.

2.9.2 Worksharing-Loop Construct

Summary

The worksharing-loop construct specifies that the iterations of one or more associated loops will be executed in parallel by threads in the team in the context of their implicit tasks. The iterations are distributed across threads that already exist in the team that is executing the **parallel** region to which the worksharing-loop region binds.

Syntax

C / C++

The syntax of the worksharing-loop construct is as follows:

```
#pragma omp for [clause[ [, ] clause] ... ] new-line  
for-loops
```

where clause is one of the following:

```
private(list)  
firstprivate(list)  
lastprivate([ lastprivate-modifier : ] list)  
linear(list[ : linear-step])  
reduction([ reduction-modifier, ]reduction-identifier : list)  
schedule([modifier [, modifier]:]kind[, chunk_size])  
collapse(n)  
ordered[ (n) ]  
nowait  
allocate([allocator :]list)  
order(concurrent)
```

The **for** directive places restrictions on the structure of all associated *for-loops*. Specifically, all associated *for-loops* must have *canonical loop form* (see Section 2.9.1 on page 95).

C / C++

The syntax of the worksharing-loop construct is as follows:

```

1  !$omp do [clause[ [, ] clause] ... ]
2      do-loops
3
4  [!$omp end do [nowait]]

```

where *clause* is one of the following:

```

6      private (list)
7      firstprivate (list)
8      lastprivate ([ lastprivate-modifier : ] list)
9      linear (list[ : linear-step ])
10     reduction ([ reduction-modifier, ] reduction-identifier : list)
11     schedule ([ modifier [, modifier] : ] kind [, chunk_size ])
12     collapse (n)
13     ordered/ (n) ]
14     allocate ([ allocator : ] list)
15     order (concurrent)

```

If an **end do** directive is not specified, an **end do** directive is assumed at the end of the *do-loops*.

The **do** directive places restrictions on the structure of all associated *do-loops*. Specifically, all associated *do-loops* must have *canonical loop form* (see Section 2.9.1 on page 95).

Binding

The binding thread set for a worksharing-loop region is the current team. A worksharing-loop region binds to the innermost enclosing **parallel** region. Only the threads of the team executing the binding **parallel** region participate in the execution of the loop iterations and the implied barrier of the worksharing-loop region if the barrier is not eliminated by a **nowait** clause.

Description

The worksharing-loop construct is associated with a loop nest that consists of one or more loops that follow the directive.

There is an implicit barrier at the end of a worksharing-loop construct unless a **nowait** clause is specified.

The **collapse** clause may be used to specify how many loops are associated with the worksharing-loop construct. The parameter of the **collapse** clause must be a constant positive

integer expression. If a **collapse** clause is specified with a parameter value greater than 1, then the iterations of the associated loops to which the clause applies are collapsed into one larger iteration space that is then divided according to the **schedule** clause. The sequential execution of the iterations in these associated loops determines the order of the iterations in the collapsed iteration space. If no **collapse** clause is present or its parameter is 1, the only loop that is associated with the worksharing-loop construct for the purposes of determining how the iteration space is divided according to the **schedule** clause is the one that immediately follows the worksharing-loop directive.

If more than one loop is associated with the worksharing-loop construct then the number of times that any intervening code between any two associated loops will be executed is unspecified but will be at least once per iteration of the loop enclosing the intervening code and at most once per iteration of the innermost loop associated with the construct. If the iteration count of any loop that is associated with the worksharing-loop construct is zero and that loop does not enclose the intervening code, the behavior is unspecified.

The integer type (or kind, for Fortran) used to compute the iteration count for the collapsed loop is implementation defined.

A worksharing-loop has logical iterations numbered 0,1,...,N-1 where N is the number of loop iterations, and the logical numbering denotes the sequence in which the iterations would be executed if a set of associated loop(s) were executed sequentially. At the beginning of each logical iteration, the loop iteration variable of each associated loop has the value that it would have if the set of the associated loop(s) were executed sequentially. The **schedule** clause specifies how iterations of these associated loops are divided into contiguous non-empty subsets, called chunks, and how these chunks are distributed among threads of the team. Each thread executes its assigned chunk(s) in the context of its implicit task. The iterations of a given chunk are executed in sequential order by the assigned thread. The *chunk_size* expression is evaluated using the original list items of any variables that are made private in the worksharing-loop construct. It is unspecified whether, in what order, or how many times, any side effects of the evaluation of this expression occur. The use of a variable in a **schedule** clause expression of a worksharing-loop construct causes an implicit reference to the variable in all enclosing constructs.

Different worksharing-loop regions with the same schedule and iteration count, even if they occur in the same parallel region, can distribute iterations among threads differently. The only exception is for the **static** schedule as specified in Table 2.5. Programs that depend on which thread executes a particular iteration under any other circumstances are non-conforming.

See Section 2.9.2.1 on page 109 for details of how the schedule for a worksharing-loop region is determined.

The schedule *kind* can be one of those specified in Table 2.5.

The schedule *modifier* can be one of those specified in Table 2.6. If the **static** schedule kind is specified or if the **ordered** clause is specified, and if the **nonmonotonic** modifier is not specified, the effect is as if the **monotonic** modifier is specified. Otherwise, unless the **monotonic** modifier is specified, the effect is as if the **nonmonotonic** modifier is specified. If

a **schedule** clause specifies a modifier then that modifier overrides any modifier that is specified in the *run-sched-var* ICV.

The **ordered** clause with the parameter may also be used to specify how many loops are associated with the worksharing-loop construct. The parameter of the **ordered** clause must be a constant positive integer expression if specified. The parameter of the **ordered** clause does not affect how the logical iteration space is then divided. If an **ordered** clause with the parameter is specified for the worksharing-loop construct, then those associated loops form a *doacross loop nest*.

If the value of the parameter in the **collapse** or **ordered** clause is larger than the number of nested loops following the construct, the behavior is unspecified.

If an **order(concurrent)** clause is present, then after assigning the iterations of the associated loops to their respective threads, as specified in Table 2.5, the iterations may be executed in any order, including concurrently.

TABLE 2.5: schedule Clause kind Values

static	<p>When <i>kind</i> is static, iterations are divided into chunks of size <i>chunk_size</i>, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number. Each chunk contains <i>chunk_size</i> iterations, except for the chunk that contains the sequentially last iteration, which may have fewer iterations.</p> <p>When no <i>chunk_size</i> is specified, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread. The size of the chunks is unspecified in this case.</p> <p>A compliant implementation of the static schedule must ensure that the same assignment of logical iteration numbers to threads will be used in two worksharing-loop regions if the following conditions are satisfied: 1) both worksharing-loop regions have the same number of loop iterations, 2) both worksharing-loop regions have the same value of <i>chunk_size</i> specified, or both worksharing-loop regions have no <i>chunk_size</i> specified, 3) both worksharing-loop regions bind to the same parallel region, and 4) neither loop is associated with a SIMD construct. A data dependence between the same logical iterations in two such loops is guaranteed to be satisfied allowing safe use of the nowait clause.</p>
---------------	---

table continued on next page

table continued from previous page

dynamic	When <i>kind</i> is dynamic , the iterations are distributed to threads in the team in chunks. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed. Each chunk contains <i>chunk_size</i> iterations, except for the chunk that contains the sequentially last iteration, which may have fewer iterations. When no <i>chunk_size</i> is specified, it defaults to 1.
guided	When <i>kind</i> is guided , the iterations are assigned to threads in the team in chunks. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned. For a <i>chunk_size</i> of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team, decreasing to 1. For a <i>chunk_size</i> with value <i>k</i> (greater than 1), the size of each chunk is determined in the same way, with the restriction that the chunks do not contain fewer than <i>k</i> iterations (except for the chunk that contains the sequentially last iteration, which may have fewer than <i>k</i> iterations). When no <i>chunk_size</i> is specified, it defaults to 1.
auto	When <i>kind</i> is auto , the decision regarding scheduling is delegated to the compiler and/or runtime system. The programmer gives the implementation the freedom to choose any possible mapping of iterations to threads in the team.
runtime	When <i>kind</i> is runtime , the decision regarding scheduling is deferred until run time, and the schedule and chunk size are taken from the <i>run-sched-var</i> ICV. If the ICV is set to auto , the schedule is implementation defined.

Note – For a team of p threads and a loop of n iterations, let $\lceil n/p \rceil$ be the integer q that satisfies $n = p * q - r$, with $0 \leq r < p$. One compliant implementation of the **static** schedule (with no specified *chunk_size*) would behave as though *chunk_size* had been specified with value q . Another compliant implementation would assign q iterations to the first $p - r$ threads, and $q - 1$ iterations to the remaining r threads. This illustrates why a conforming program must not rely on the details of a particular implementation.

A compliant implementation of the **guided** schedule with a *chunk_size* value of k would assign $q = \lceil n/p \rceil$ iterations to the first available thread and set n to the larger of $n - q$ and $p * k$. It would then repeat this process until q is greater than or equal to the number of remaining iterations, at which time the remaining iterations form the final chunk. Another compliant implementation could use the same method, except with $q = \lceil n/(2p) \rceil$, and set n to the larger of $n - q$ and $2 * p * k$.

TABLE 2.6: `schedule` Clause *modifier* Values

monotonic	When the monotonic modifier is specified then each thread executes the chunks that it is assigned in increasing logical iteration order.
nonmonotonic	When the nonmonotonic modifier is specified then chunks are assigned to threads in any order and the behavior of an application that depends on any execution order of the chunks is unspecified.
simd	When the simd modifier is specified and the loop is associated with a SIMD construct, the <i>chunk_size</i> for all chunks except the first and last chunks is $new_chunk_size = \lceil chunk_size / simd_width \rceil * simd_width$ where <i>simd_width</i> is an implementation-defined value. The first chunk will have at least <i>new_chunk_size</i> iterations except if it is also the last chunk. The last chunk may have fewer iterations than <i>new_chunk_size</i> . If the simd modifier is specified and the loop is not associated with a SIMD construct, the modifier is ignored.

Execution Model Events

The *ws-loop-begin* event occurs after an implicit task encounters a worksharing-loop construct but before the task starts execution of the structured block of the worksharing-loop region.

The *ws-loop-end* event occurs after a worksharing-loop region finishes execution but before resuming execution of the encountering task.

The *ws-loop-iteration-begin* event occurs once for each iteration of a worksharing-loop before the iteration is executed by an implicit task.

Tool Callbacks

A thread dispatches a registered **ompt_callback_work** callback with **ompt_scope_begin** as its *endpoint* argument and **work_loop** as its *wstype* argument for each occurrence of a *ws-loop-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_work** callback with **ompt_scope_end** as its *endpoint* argument and **work_loop** as its *wstype* argument for each occurrence of a *ws-loop-end* event in that thread. The callbacks occur in the context of the implicit task. The callbacks have type signature **ompt_callback_work_t**.

A thread dispatches a registered **ompt_callback_dispatch** callback for each occurrence of a *ws-loop-iteration-begin* event in that thread. The callback occurs in the context of the implicit task. The callback has type signature **ompt_callback_dispatch_t**.

Restrictions

Restrictions to the worksharing-loop construct are as follows:

- No OpenMP directive may appear in the region between any associated loops.
- If a **collapse** clause is specified, exactly one loop must occur in the region at each nesting level up to the number of loops specified by the parameter of the **collapse** clause.
- If the **ordered** clause is present, all loops associated with the construct must be perfectly nested; that is there must be no intervening code between any two loops.
- If a **reduction** clause with the **inscan** modifier is specified, neither the **ordered** nor **schedule** clause may appear on the worksharing-loop directive.
- The values of the loop control expressions of the loops associated with the worksharing-loop construct must be the same for all threads in the team.
- Only one **schedule** clause can appear on a worksharing-loop directive.
- The **schedule** clause must not appear on the worksharing-loop directive if the associated loop(s) form a non-rectangular loop nest.
- The **ordered** clause must not appear on the worksharing-loop directive if the associated loop(s) form a non-rectangular loop nest.
- Only one **collapse** clause can appear on a worksharing-loop directive.
- *chunk_size* must be a loop invariant integer expression with a positive value.
- The value of the *chunk_size* expression must be the same for all threads in the team.
- The value of the *run-sched-var* ICV must be the same for all threads in the team.
- When **schedule(runtime)** or **schedule(auto)** is specified, *chunk_size* must not be specified.
- A *modifier* may not be specified on a **linear** clause.
- Only one **ordered** clause can appear on a worksharing-loop directive.
- The **ordered** clause must be present on the worksharing-loop construct if any **ordered** region ever binds to a worksharing-loop region arising from the worksharing-loop construct.
- The **nonmonotonic** modifier cannot be specified if an **ordered** clause is specified.
- Either the **monotonic** modifier or the **nonmonotonic** modifier can be specified but not both.
- The loop iteration variable may not appear in a **threadprivate** directive.
- If both the **collapse** and **ordered** clause with a parameter are specified, the parameter of the **ordered** clause must be greater than or equal to the parameter of the **collapse** clause.

- A **linear** clause or an **ordered** clause with a parameter can be specified on a worksharing-loop directive but not both.
- If an **order (concurrent)** clause is present, all restrictions from the **loop** construct with an **order (concurrent)** clause also apply.
- If an **order (concurrent)** clause is present, an **ordered** clause may not appear on the same directive.

C / C++

- The associated *for-loops* must be structured blocks.
- Only an iteration of the innermost associated loop may be curtailed by a **continue** statement.
- No statement can branch to any associated **for** statement.
- Only one **nowait** clause can appear on a **for** directive.
- A throw executed inside a worksharing-loop region must cause execution to resume within the same iteration of the worksharing-loop region, and the same thread that threw the exception must catch it.

C / C++

Fortran

- The associated *do-loops* must be structured blocks.
- Only an iteration of the innermost associated loop may be curtailed by a **CYCLE** statement.
- No statement in the associated loops other than the **DO** statements can cause a branch out of the loops.
- The *do-loop* iteration variable must be of type integer.
- The *do-loop* cannot be a **DO WHILE** or a **DO** loop without loop control.

Fortran

Cross References

- **order (concurrent)** clause, see Section 2.9.5 on page 128.
- **ordered** construct, see Section 2.17.9 on page 250.
- **depend** clause, see Section 2.17.11 on page 255.
- **private**, **firstprivate**, **lastprivate**, **linear**, and **reduction** clauses, see Section 2.19.4 on page 282.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.4.4.11 on page 443.
- **ompt_work_loop**, see Section 4.4.4.15 on page 445.
- **ompt_callback_work_t**, see Section 4.5.2.5 on page 464.
- **OMP_SCHEDULE** environment variable, see Section 6.1 on page 601.

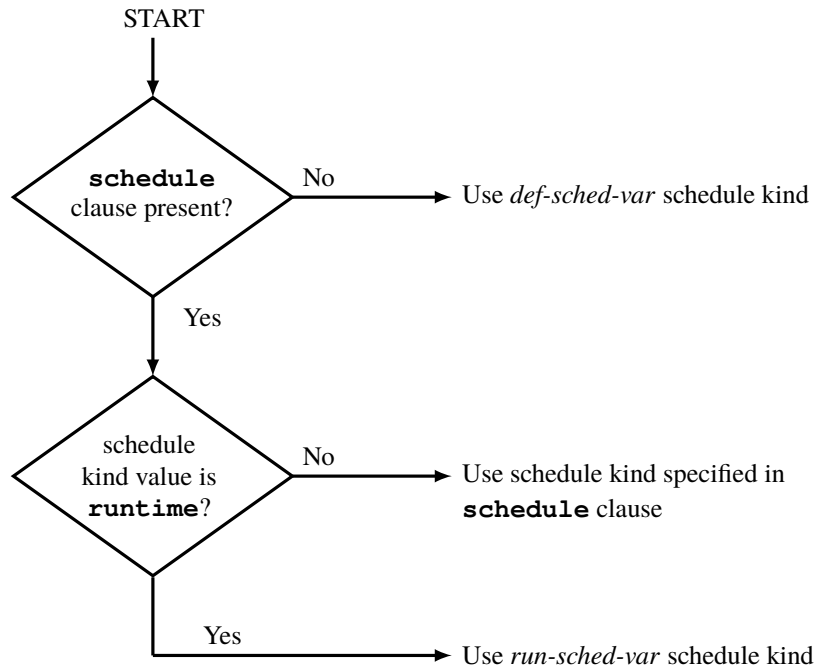


FIGURE 2.1: Determining the **schedule** for a Worksharing-Loop

1 2.9.2.1 Determining the Schedule of a Worksharing-Loop

2 When execution encounters a worksharing-loop directive, the **schedule** clause (if any) on the
 3 directive, and the *run-sched-var* and *def-sched-var* ICVs are used to determine how loop iterations
 4 are assigned to threads. See Section 2.5 on page 63 for details of how the values of the ICVs are
 5 determined. If the worksharing-loop directive does not have a **schedule** clause then the current
 6 value of the *def-sched-var* ICV determines the schedule. If the worksharing-loop directive has a
 7 **schedule** clause that specifies the **runtime** schedule kind then the current value of the
 8 *run-sched-var* ICV determines the schedule. Otherwise, the value of the **schedule** clause
 9 determines the schedule. Figure 2.1 describes how the schedule for a worksharing-loop is
 10 determined.

11 Cross References

- 12 • ICVs, see Section 2.5 on page 63.

1 2.9.3 SIMD Directives


2 2.9.3.1 `simd` Construct

3 Summary

4 The `simd` construct can be applied to a loop to indicate that the loop can be transformed into a
5 SIMD loop (that is, multiple iterations of the loop can be executed concurrently using SIMD
6 instructions).

7 Syntax

8 The syntax of the `simd` construct is as follows:

9 
10 `#pragma omp simd [clause[[,] clause] ...] new-line`
11 *for-loops*

11 where *clause* is one of the following:

12 `if([simd :] scalar-expression)`
13 `safelen(length)`
14 `simdlen(length)`
15 `linear(list [: linear-step])`
16 `aligned(list [: alignment])`
17 `nontemporal(list)`
18 `private(list)`
19 `lastprivate([lastprivate-modifier :] list)`
20 `reduction([reduction-modifier ,] reduction-identifier : list)`
21 `collapse(n)`
22 `order(concurrent)`

23 The `simd` directive places restrictions on the structure of the associated *for-loops*. Specifically, all
24 associated *for-loops* must have *canonical loop form* (Section 2.9.1 on page 95).



```

1  !$omp simd [clause[ [, ] clause ... ]
2      do-loops
3  [!$omp end simd]

```

where *clause* is one of the following:

```

5      if([simd :] scalar-logical-expression)
6      safelen(length)
7      simdlen(length)
8      linear(list[ : linear-step])
9      aligned(list[ : alignment])
10     nontemporal(list)
11     private(list)
12     lastprivate([ lastprivate-modifier: ] list)
13     reduction([ reduction-modifier, ]reduction-identifier : list)
14     collapse(n)
15     order(concurrent)

```

If an **end simd** directive is not specified, an **end simd** directive is assumed at the end of the *do-loops*.

The **simd** directive places restrictions on the structure of all associated *do-loops*. Specifically, all associated *do-loops* must have *canonical loop form* (see Section 2.9.1 on page 95).

Binding

A **simd** region binds to the current task region. The binding thread set of the **simd** region is the current team.

Description

The **simd** construct enables the execution of multiple iterations of the associated loops concurrently by means of SIMD instructions.

The **collapse** clause may be used to specify how many loops are associated with the construct. The parameter of the **collapse** clause must be a constant positive integer expression. If no **collapse** clause is present, the only loop that is associated with the **simd** construct is the one that immediately follows the directive.

If more than one loop is associated with the **simd** construct, then the iterations of all associated loops are collapsed into one larger iteration space that is then executed with SIMD instructions. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

If more than one loop is associated with the **simd** construct then the number of times that any intervening code between any two associated loops will be executed is unspecified but will be at least once per iteration of the loop enclosing the intervening code and at most once per iteration of the innermost loop associated with the construct. If the iteration count of any loop that is associated with the **simd** construct is zero and that loop does not enclose the intervening code, the behavior is unspecified.

The integer type (or kind, for Fortran) used to compute the iteration count for the collapsed loop is implementation defined.

A SIMD loop has logical iterations numbered 0,1,...,N-1 where N is the number of loop iterations, and the logical numbering denotes the sequence in which the iterations would be executed if the associated loop(s) were executed with no SIMD instructions. At the beginning of each logical iteration, the loop iteration variable of each associated loop has the value that it would have if the set of the associated loop(s) were executed sequentially. The number of iterations that are executed concurrently at any given time is implementation defined. Each concurrent iteration will be executed by a different SIMD lane. Each set of concurrent iterations is a SIMD chunk. Lexical forward dependencies in the iterations of the original loop must be preserved within each SIMD chunk.

The **safelen** clause specifies that no two concurrent iterations within a SIMD chunk can have a distance in the logical iteration space that is greater than or equal to the value given in the clause. The parameter of the **safelen** clause must be a constant positive integer expression. The **simdlen** clause specifies the preferred number of iterations to be executed concurrently unless an **if** clause is present and evaluates to *false*, in which case the preferred number of iterations to be executed concurrently is one. The parameter of the **simdlen** clause must be a constant positive integer expression.

▼ C / C++ ▼

The **aligned** clause declares that the object to which each list item points is aligned to the number of bytes expressed in the optional parameter of the **aligned** clause.

▲ C / C++ ▲

▼ Fortran ▼

The **aligned** clause declares that the location of each list item is aligned to the number of bytes expressed in the optional parameter of the **aligned** clause.

▲ Fortran ▲

The optional parameter of the **aligned** clause, *alignment*, must be a constant positive integer expression. If no optional parameter is specified, implementation-defined default alignments for SIMD instructions on the target platforms are assumed.

The **nontemporal** clause specifies that accesses to the storage locations to which the list items refer have low temporal locality across the iterations in which those storage locations are accessed.

Restrictions

- No OpenMP directive may appear in the region between any associated loops.
- If a **collapse** clause is specified, exactly one loop must occur in the region at each nesting level up to the number of loops specified by the parameter of the **collapse** clause.
- The associated loops must be structured blocks.
- A program that branches into or out of a **simd** region is non-conforming.
- Only one **collapse** clause can appear on a **simd** directive.
- A *list-item* cannot appear in more than one **aligned** clause.
- A *list-item* cannot appear in more than one **nontemporal** clause.
- Only one **safelen** clause can appear on a **simd** directive.
- Only one **simdlen** clause can appear on a **simd** directive.
- If both **simdlen** and **safelen** clauses are specified, the value of the **simdlen** parameter must be less than or equal to the value of the **safelen** parameter.
- A *modifier* may not be specified on a **linear** clause.
- The only OpenMP constructs that can be encountered during execution of a **simd** region are the **atomic** construct, the **loop** construct, the **simd** construct and the **ordered** construct with the **simd** clause.
- If an **order (concurrent)** clause is present, all restrictions from the **loop** construct with an **order (concurrent)** clause also apply.

- ▼ C / C++ ▼
- The **simd** region cannot contain calls to the **longjmp** or **setjmp** functions.
- ▲ C / C++ ▲
- ▼ C ▼
- The type of list items appearing in the **aligned** clause must be array or pointer.
- ▲ C ▲

C++

- The type of list items appearing in the **aligned** clause must be array, pointer, reference to array, or reference to pointer.
- No exception can be raised in the **simd** region.

C++

Fortran

- The *do-loop* iteration variable must be of type **integer**.
- The *do-loop* cannot be a **DO WHILE** or a **DO** loop without loop control.
- If a list item on the **aligned** clause has the **ALLOCATABLE** attribute, the allocation status must be allocated.
- If a list item on the **aligned** clause has the **POINTER** attribute, the association status must be associated.
- If the type of a list item on the **aligned** clause is either **C_PTR** or Cray pointer, the list item must be defined.

Fortran

Cross References

- **order (concurrent)** clause, see Section 2.9.5 on page 128.
- **if** Clause, see Section 2.15 on page 220.
- **private**, **lastprivate**, **linear** and **reduction** clauses, see Section 2.19.4 on page 282.

2.9.3.2 Worksharing-Loop SIMD Construct

Summary

The worksharing-loop SIMD construct specifies that the iterations of one or more associated loops will be distributed across threads that already exist in the team and that the iterations executed by each thread can also be executed concurrently using SIMD instructions. The worksharing-loop SIMD construct is a composite construct.

Syntax

C / C++

```
#pragma omp for simd [clause[ [, ] clause]... ] new-line  
    for-loops
```

where *clause* can be any of the clauses accepted by the **for** or **simd** directives with identical meanings and restrictions.

C / C++

Fortran

```
!$omp do simd [clause[ [, ] clause] ... ]  
    do-loops  
[!$omp end do simd [nowait] ]
```

where *clause* can be any of the clauses accepted by the **simd** or **do** directives, with identical meanings and restrictions.

If an **end do simd** directive is not specified, an **end do simd** directive is assumed at the end of the *do-loops*.

Fortran

Description

The worksharing-loop SIMD construct will first distribute the iterations of the associated loop(s) across the implicit tasks of the parallel region in a manner consistent with any clauses that apply to the worksharing-loop construct. The resulting chunks of iterations will then be converted to a SIMD loop in a manner consistent with any clauses that apply to the **simd** construct.

Execution Model Events

This composite construct generates the same events as the worksharing-loop construct.

Tool Callbacks

This composite construct dispatches the same callbacks as the worksharing-loop construct.

Restrictions

All restrictions to the worksharing-loop construct and the **simd** construct apply to the worksharing-loop SIMD construct. In addition, the following restrictions apply:

- No **ordered** clause with a parameter can be specified.
- A list item may appear in a **linear** or **firstprivate** clause but not both.

Cross References

- worksharing-loop construct, see Section 2.9.2 on page 101.
- **simd** construct, see Section 2.9.3.1 on page 110.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.9.3.3 declare simd Directive

Summary

The **declare simd** directive can be applied to a function (C, C++ and Fortran) or a subroutine (Fortran) to enable the creation of one or more versions that can process multiple arguments using SIMD instructions from a single invocation in a SIMD loop. The **declare simd** directive is a declarative directive. There may be multiple **declare simd** directives for a function (C, C++, Fortran) or subroutine (Fortran).

Syntax

The syntax of the **declare simd** directive is as follows:

C / C++

```
#pragma omp declare simd [clause[ [, ] clause] ... ] new-line
[#pragma omp declare simd [clause[ [, ] clause] ... ] new-line]
[ ... ]
    function definition or declaration
```

where *clause* is one of the following:

```
    simdlen (length)
    linear (linear-list[ : linear-step])
    aligned (argument-list[ : alignment])
    uniform (argument-list)
    inbranch
    notinbranch
```

C / C++

Fortran

```
1 !$omp declare simd [ (proc-name) ] [clause[ [, ] clause] ... ]
```

2 where *clause* is one of the following:

```
3 simdlen(length)  
4 linear(linear-list[ : linear-step])  
5 aligned(argument-list[ : alignment])  
6 uniform(argument-list)  
7 inbranch  
8 notinbranch
```

Fortran

Description

C / C++

10 The use of one or more **declare simd** directives immediately prior to a function declaration or
11 definition enables the creation of corresponding SIMD versions of the associated function that can
12 be used to process multiple arguments from a single invocation in a SIMD loop concurrently.

13 The expressions appearing in the clauses of each directive are evaluated in the scope of the
14 arguments of the function declaration or definition.

C / C++

Fortran

15 The use of one or more **declare simd** directives for a specified subroutine or function enables
16 the creation of corresponding SIMD versions of the subroutine or function that can be used to
17 process multiple arguments from a single invocation in a SIMD loop concurrently.

Fortran

18 If a SIMD version is created, the number of concurrent arguments for the function is determined by
19 the **simdlen** clause. If the **simdlen** clause is used its value corresponds to the number of
20 concurrent arguments of the function. The parameter of the **simdlen** clause must be a constant
21 positive integer expression. Otherwise, the number of concurrent arguments for the function is
22 implementation defined.

C++

23 The special *this* pointer can be used as if it was one of the arguments to the function in any of the
24 **linear**, **aligned**, or **uniform** clauses.

C++

25 The **uniform** clause declares one or more arguments to have an invariant value for all concurrent
26 invocations of the function in the execution of a single SIMD loop.

C / C++

The **aligned** clause declares that the object to which each list item points is aligned to the number of bytes expressed in the optional parameter of the **aligned** clause.

C / C++

Fortran

The **aligned** clause declares that the target of each list item is aligned to the number of bytes expressed in the optional parameter of the **aligned** clause.

Fortran

The optional parameter of the **aligned** clause, *alignment*, must be a constant positive integer expression. If no optional parameter is specified, implementation-defined default alignments for SIMD instructions on the target platforms are assumed.

The **inbranch** clause specifies that the SIMD version of the function will always be called from inside a conditional statement of a SIMD loop. The **notinbranch** clause specifies that the SIMD version of the function will never be called from inside a conditional statement of a SIMD loop. If neither clause is specified, then the SIMD version of the function may or may not be called from inside a conditional statement of a SIMD loop.

Restrictions

- Each argument can appear in at most one **uniform** or **linear** clause.
- At most one **simdlen** clause can appear in a **declare simd** directive.
- Either **inbranch** or **notinbranch** may be specified, but not both.
- When a *linear-step* expression is specified in a **linear** clause it must be either a constant integer expression or an integer-typed parameter that is specified in a **uniform** clause on the directive.
- The function or subroutine body must be a structured block.
- The execution of the function or subroutine, when called from a SIMD loop, cannot result in the execution of an OpenMP construct except for an **ordered** construct with the **simd** clause or an **atomic** construct.
- The execution of the function or subroutine cannot have any side effects that would alter its execution for concurrent iterations of a SIMD chunk.
- A program that branches into or out of the function is non-conforming.

C / C++

- If the function has any declarations, then the **declare simd** construct for any declaration that has one must be equivalent to the one specified for the definition. Otherwise, the result is unspecified.

- The function cannot contain calls to the **longjmp** or **setjmp** functions.

C / C++

C

- The type of list items appearing in the **aligned** clause must be array or pointer.

C

C++

- The function cannot contain any calls to **throw**.
- The type of list items appearing in the **aligned** clause must be array, pointer, reference to array, or reference to pointer.

C++

Fortran

- *proc-name* must not be a generic name, procedure pointer or entry name.
- If *proc-name* is omitted, the **declare simd** directive must appear in the specification part of a subroutine subprogram or a function subprogram for which creation of the SIMD versions is enabled.
- Any **declare simd** directive must appear in the specification part of a subroutine subprogram, function subprogram or interface body to which it applies.
- If a **declare simd** directive is specified in an interface block for a procedure, it must match a **declare simd** directive in the definition of the procedure.
- If a procedure is declared via a procedure declaration statement, the procedure *proc-name* should appear in the same specification.
- If a **declare simd** directive is specified for a procedure name with explicit interface and a **declare simd** directive is also specified for the definition of the procedure then the two **declare simd** directives must match. Otherwise the result is unspecified.
- Procedure pointers may not be used to access versions created by the **declare simd** directive.
- The type of list items appearing in the **aligned** clause must be **C_PTR** or Cray pointer, or the list item must have the **POINTER** or **ALLOCATABLE** attribute.

Fortran

Cross References

- **linear** clause, see Section [2.19.4.6](#) on page [290](#).
- **reduction** clause, see Section [2.19.5.4](#) on page [300](#).

1 2.9.4 distribute Loop Constructs

2 2.9.4.1 distribute Construct

3 Summary

4 The **distribute** construct specifies that the iterations of one or more loops will be executed by
5 the initial teams in the context of their implicit tasks. The iterations are distributed across the initial
6 threads of all initial teams that execute the **teams** region to which the **distribute** region binds.

7 Syntax

8  C / C++ 

8 The syntax of the **distribute** construct is as follows:

```
9 #pragma omp distribute [clause[ [, ] clause] ... ] new-line  
10 for-loops
```

11 Where *clause* is one of the following:

```
12 private (list)  
13 firstprivate (list)  
14 lastprivate (list)  
15 collapse (n)  
16 dist_schedule (kind[, chunk_size])  
17 allocate ([allocator :]list)
```

18 The **distribute** directive places restrictions on the structure of all associated *for-loops*.
19 Specifically, all associated *for-loops* must have *canonical loop form* (see Section 2.9.1 on page 95).

20  C / C++ 

21  Fortran 

20 The syntax of the **distribute** construct is as follows:

```
21 !$omp distribute [clause[ [, ] clause] ... ]  
22 do-loops  
23 [!$omp end distribute]
```

Where *clause* is one of the following:

```
private(list)
firstprivate(list)
lastprivate(list)
collapse(n)
dist_schedule(kind[, chunk_size])
allocate([allocator:]list)
```

If an **end distribute** directive is not specified, an **end distribute** directive is assumed at the end of the *do-loops*.

The **distribute** directive places restrictions on the structure of all associated *do-loops*. Specifically, all associated *do-loops* must have *canonical loop form* (see Section 2.9.1 on page 95).

Fortran

Binding

The binding thread set for a **distribute** region is the set of initial threads executing an enclosing **teams** region. A **distribute** region binds to this **teams** region.

Description

The **distribute** construct is associated with a loop nest consisting of one or more loops that follow the directive.

There is no implicit barrier at the end of a **distribute** construct. To avoid data races the original list items modified due to **lastprivate** or **linear** clauses should not be accessed between the end of the **distribute** construct and the end of the **teams** region to which the **distribute** binds.

The **collapse** clause may be used to specify how many loops are associated with the **distribute** construct. The parameter of the **collapse** clause must be a constant positive integer expression. If no **collapse** clause is present or its parameter is 1, the only loop that is associated with the **distribute** construct is the one that immediately follows the **distribute** construct. If a **collapse** clause is specified with a parameter value greater than 1 and more than one loop is associated with the **distribute** construct, then the iteration of all associated loops are collapsed into one larger iteration space. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

A distribute loop has logical iterations numbered 0,1,...,N-1 where N is the number of loop iterations, and the logical numbering denotes the sequence in which the iterations would be executed if the set of associated loop(s) were executed sequentially. At the beginning of each

logical iteration, the loop iteration variable of each associated loop has the value that it would have if the set of the associated loop(s) were executed sequentially.

If more than one loop is associated with the **distribute** construct then the number of times that any intervening code between any two associated loops will be executed is unspecified but will be at least once per iteration of the loop enclosing the intervening code and at most once per iteration of the innermost loop associated with the construct. If the iteration count of any loop that is associated with the **distribute** construct is zero and that loop does not enclose the intervening code, the behavior is unspecified.

The integer type (or kind, for Fortran) used to compute the iteration count for the collapsed loop is implementation defined.

If **dist_schedule** is specified, *kind* must be **static**. If specified, iterations are divided into chunks of size *chunk_size*, chunks are assigned to the initial teams of the league in a round-robin fashion in the order of the initial team number. When no *chunk_size* is specified, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each initial team of the league. The size of the chunks is unspecified in this case.

When no **dist_schedule** clause is specified, the schedule is implementation defined.

Execution Model Events

The *distribute-begin* event occurs after an implicit task encounters a **distribute** construct but before the task starts to execute the structured block of the **distribute** region.

The *distribute-end* event occurs after an implicit task finishes execution of a **distribute** region but before it resumes execution of the enclosing context.

Tool Callbacks

A thread dispatches a registered **ompt_callback_work** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_work_distribute** as its *wstype* argument for each occurrence of a *distribute-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_work** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_work_distribute** as its *wstype* argument for each occurrence of a *distribute-end* event in that thread. The callbacks occur in the context of the implicit task. The callbacks have type signature **ompt_callback_work_t**.

Restrictions

Restrictions to the **distribute** construct are as follows:

- The **distribute** construct inherits the restrictions of the worksharing-loop construct.
- Each **distribute** region must be encountered by the initial threads of all initial teams in a league or by none at all.

- The sequence of the **distribute** regions encountered must be the same for every initial thread of every initial team in a league.
- The region corresponding to the **distribute** construct must be strictly nested inside a **teams** region.
- A list item may appear in a **firstprivate** or **lastprivate** clause but not both.
- The **dist_schedule** clause must not appear on the **distribute** directive if the associated loop(s) form a non-rectangular loop nest.

Cross References

- **teams** construct, see Section 2.7 on page 82
- worksharing-loop construct, see Section 2.9.2 on page 101.
- **ompt_work_distribute**, see Section 4.4.4.15 on page 445.
- **ompt_callback_work_t**, see Section 4.5.2.5 on page 464.

2.9.4.2 **distribute simd** Construct

Summary

The **distribute simd** construct specifies a loop that will be distributed across the master threads of the **teams** region and executed concurrently using SIMD instructions. The **distribute simd** construct is a composite construct.

Syntax

The syntax of the **distribute simd** construct is as follows:

C / C++

```
#pragma omp distribute simd [clause[ [, ] clause] ... ] newline
    for-loops
```

where *clause* can be any of the clauses accepted by the **distribute** or **simd** directives with identical meanings and restrictions.

C / C++

Fortran

```
1  !$omp distribute simd [clause[ [, ] clause] ... ]  
2      do-loops  
3  [!$omp end distribute simd]
```

where *clause* can be any of the clauses accepted by the **distribute** or **simd** directives with identical meanings and restrictions.

If an **end distribute simd** directive is not specified, an **end distribute simd** directive is assumed at the end of the *do-loops*.

Fortran

Description

The **distribute simd** construct will first distribute the iterations of the associated loop(s) according to the semantics of the **distribute** construct and any clauses that apply to the distribute construct. The resulting chunks of iterations will then be converted to a SIMD loop in a manner consistent with any clauses that apply to the **simd** construct.

Execution Model Events

This composite construct generates the same events as the **distribute** construct.

Tool Callbacks

This composite construct dispatches the same callbacks as the **distribute** construct.

Restrictions

- The restrictions for the **distribute** and **simd** constructs apply.
- A list item may not appear in a **linear** clause unless it is the loop iteration variable of a loop that is associated with the construct.
- The **conditional** modifier may not appear in a **lastprivate** clause.

Cross References

- **simd** construct, see Section [2.9.3.1](#) on page [110](#).
- **distribute** construct, see Section [2.9.4.1](#) on page [120](#).
- Data attribute clauses, see Section [2.19.4](#) on page [282](#).


1 2.9.4.3 Distribute Parallel Worksharing-Loop Construct

2 Summary


3 The distribute parallel worksharing-loop construct specifies a loop that can be executed in parallel
4 by multiple threads that are members of multiple teams. The distribute parallel worksharing-loop
5 construct is a composite construct.

6 Syntax

7 The syntax of the distribute parallel worksharing-loop construct is as follows:

8  **#pragma omp distribute parallel for** [*clause* [,] *clause*] ...] *newline*
9 *for-loops*

10 where *clause* can be any of the clauses accepted by the **distribute** or parallel worksharing-loop
11 directives with identical meanings and restrictions.

12  **!\$omp distribute parallel do** [*clause* [,] *clause*] ...]
13 *do-loops*
14 **[!\$omp end distribute parallel do]**

15 where *clause* can be any of the clauses accepted by the **distribute** or parallel worksharing-loop
16 directives with identical meanings and restrictions.

17 If an **end distribute parallel do** directive is not specified, an **end distribute**
18 **parallel do** directive is assumed at the end of the *do-loops*.

19  Fortran

19 Description

20 The distribute parallel worksharing-loop construct will first distribute the iterations of the
21 associated loop(s) into chunks according to the semantics of the **distribute** construct and any
22 clauses that apply to the **distribute** construct. Each of these chunks will form a loop. Each
23 resulting loop will then be distributed across the threads within the **teams** region to which the
24 **distribute** construct binds in a manner consistent with any clauses that apply to the parallel
25 worksharing-loop construct.

26 Execution Model Events

27 This composite construct generates the same events as the **distribute** and parallel
28 worksharing-loop constructs.

Tool Callbacks

This composite construct dispatches the same callbacks as the **distribute** and parallel worksharing-loop constructs.

Restrictions

- The restrictions for the **distribute** and parallel worksharing-loop constructs apply.
- No **ordered** clause can be specified.
- No **linear** clause can be specified.
- The **conditional** modifier may not appear in a **lastprivate** clause.

Cross References

- **distribute** construct, see Section 2.9.4.1 on page 120.
- Parallel worksharing-loop construct, see Section 2.13.1 on page 185.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.9.4.4 Distribute Parallel Worksharing-Loop SIMD Construct

Summary

The distribute parallel worksharing-loop SIMD construct specifies a loop that can be executed concurrently using SIMD instructions in parallel by multiple threads that are members of multiple teams. The distribute parallel worksharing-loop SIMD construct is a composite construct.

Syntax

C / C++

The syntax of the distribute parallel worksharing-loop SIMD construct is as follows:

```
#pragma omp distribute parallel for simd \  
    [clause[ [, ] clause] ... ] newline  
for-loops
```

where *clause* can be any of the clauses accepted by the **distribute** or parallel worksharing-loop SIMD directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the distribute parallel worksharing-loop SIMD construct is as follows:

```
!$omp distribute parallel do simd [clause[ [, ] clause] ... ]  
do-loops  
/!$omp end distribute parallel do simd/
```

where *clause* can be any of the clauses accepted by the **distribute** or parallel worksharing-loop SIMD directives with identical meanings and restrictions.

If an **end distribute parallel do simd** directive is not specified, an **end distribute parallel do simd** directive is assumed at the end of the *do-loops*.

Fortran

Description

The distribute parallel worksharing-loop SIMD construct will first distribute the iterations of the associated loop(s) according to the semantics of the **distribute** construct and any clauses that apply to the **distribute** construct. The resulting loops will then be distributed across the threads contained within the **teams** region to which the **distribute** construct binds in a manner consistent with any clauses that apply to the parallel worksharing-loop construct. The resulting chunks of iterations will then be converted to a SIMD loop in a manner consistent with any clauses that apply to the **simd** construct.

Execution Model Events

This composite construct generates the same events as the **distribute** and parallel worksharing-loop SIMD constructs.

Tool Callbacks

This composite construct dispatches the same callbacks as the **distribute** and parallel worksharing-loop SIMD constructs.

Restrictions

- The restrictions for the **distribute** and parallel worksharing-loop SIMD constructs apply.
- No **ordered** clause can be specified.
- A list item may not appear in a **linear** clause unless it is the loop iteration variable of a loop that is associated with the construct.
- The **conditional** modifier may not appear in a **lastprivate** clause.

Cross References

- **distribute** construct, see Section 2.9.4.1 on page 120.
- Parallel worksharing-loop SIMD construct, see Section 2.13.5 on page 190.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.9.5 loop Construct

Summary

A **loop** construct specifies that the iterations of the associated loops may execute concurrently and permits the encountering thread(s) to execute the loop accordingly.

Syntax

C / C++

The syntax of the **loop** construct is as follows:

```
#pragma omp loop [clause[ [, ] clause] ... ] new-line  
for-loops
```

where *clause* is one of the following:

```
bind(binding)  
collapse(n)  
order(concurrent)  
private(list)  
lastprivate(list)  
reduction([default ,]reduction-identifier : list)
```

where *binding* is one of the following:

```
teams  
parallel  
thread
```

The **loop** directive places restrictions on the structure of all associated *for-loops*. Specifically, all associated *for-loops* must have *canonical loop form* (see Section 2.9.1 on page 95).

C / C++

The syntax of the **loop** construct is as follows:

```
!$omp loop [clause [ , ] clause] ... ]
    do-loops
[!$omp end loop]
```

where *clause* is one of the following:

```
bind(binding)
collapse(n)
order(concurrent)
private(list)
lastprivate(list)
reduction([default , ]reduction-identifier : list)
```

where *binding* is one of the following:

```
teams
parallel
thread
```

If an **end loop** directive is not specified, an **end loop** directive is assumed at the end of the *do-loops*.

The **loop** directive places restrictions on the structure of all associated *do-loops*. Specifically, all associated *do-loops* must have *canonical loop form* (see Section 2.9.1 on page 95).

Binding

If the **bind** clause is present on the construct, the binding region is determined by *binding*. Specifically, if *binding* is **teams** and there exists an innermost enclosing **teams** region then the binding region is that **teams** region; if *binding* is **parallel** then the binding region is the innermost enclosing parallel region, which may be an implicit parallel region; and if *binding* is **thread** then the binding region is not defined. If the **bind** clause is not present on the construct and the **loop** construct is closely nested inside a **teams** or **parallel** construct, the binding region is the corresponding **teams** or **parallel** region. If none of those conditions hold, the binding region is not defined.

If the binding region is a **teams** region, then the binding thread set is the set of master threads that are executing that region. If the binding region is a parallel region, then the binding thread set is the team of threads that are executing that region. If the binding region is not defined, then the binding thread set is the encountering thread.

Description

The **loop** construct is associated with a loop nest that consists of one or more loops that follow the directive. The directive asserts that the iterations may execute in any order, including concurrently.

The **collapse** clause may be used to specify how many loops are associated with the **loop** construct. The parameter of the **collapse** clause must be a constant positive integer expression. If a **collapse** clause is specified with a parameter value greater than 1, then the iterations of the associated loops to which the clause applies are collapsed into one larger iteration space with unspecified ordering. If no **collapse** clause is present or its parameter is 1, the only loop that is associated with the **loop** construct is the one that immediately follows the **loop** directive.

If more than one loop is associated with the **loop** construct then the number of times that any intervening code between any two associated loops will be executed is unspecified but will be at least once per iteration of the loop enclosing the intervening code and at most once per iteration of the innermost loop associated with the construct. If the iteration count of any loop that is associated with the **loop** construct is zero and that loop does not enclose the intervening code, the behavior is unspecified.

The iteration space of the associated loops correspond to logical iterations numbered 0,1,...,N-1 where N is the number of loop iterations, and the logical numbering denotes the sequence in which the iterations would be executed if a set of associated loop(s) were executed sequentially. At the beginning of each logical iteration, the loop iteration variable of each associated loop has the value that it would have if the set of the associated loop(s) were executed sequentially.

Each logical iteration is executed once per instance of the **loop** region that is encountered by the binding thread set.

If the **order (concurrent)** clause appears on the **loop** construct, the iterations of the associated loops may execute in any order, including concurrently. If the **order** clause is not present, the behavior is as if the **order (concurrent)** clause appeared on the construct.

The set of threads that may execute the iterations of the **loop** region is the binding thread set. Each iteration is executed by one thread from this set.

If the **loop** region binds to a **teams** region, the threads in the binding thread set may continue execution after the **loop** region without waiting for all iterations of the associated loop(s) to complete. The iterations are guaranteed to complete before the end of the **teams** region.

If the **loop** region does not bind to a **teams** region, all iterations of the associated loop(s) must complete before the encountering thread(s) continue execution after the **loop** region.

Restrictions

Restrictions to the **loop** construct are as follows:

- If the **collapse** clause is specified then there may be no intervening OpenMP directives between the associated loops.

- At most one **collapse** clause can appear on a **loop** directive.
- A list item may not appear in a **lastprivate** clause unless it is the loop iteration variable of a loop that is associated with the construct.
- If a **loop** construct is not nested inside another OpenMP construct and it appears in a procedure, the **bind** clause must be present.
- If a **loop** region binds to a **teams** or parallel region, it must be encountered by all threads in the binding thread set or by none of them.
- If the **bind** clause is present and *binding* is **teams**, the **loop** region corresponding to the **loop** construct must be strictly nested inside a **teams** region.
- If the **bind** clause is present and *binding* is **parallel**, the behavior is unspecified if the **loop** region corresponding to a **loop** construct is closely nested inside a **simd** region.
- The only constructs that may be nested inside a **loop** region are the **loop** construct, the **parallel** construct, the **simd** construct, and combined constructs for which the first construct is a **parallel** construct.
- A **loop** region corresponding to a **loop** construct may not contain calls to procedures that contain OpenMP directives.
- A **loop** region corresponding to a **loop** construct may not contain calls to the OpenMP Runtime API.
- If a threadprivate variable is referenced inside a **loop** region, the behavior is unspecified.

C / C++

- The associated *for-loops* must be structured blocks.
- No statement can branch to any associated **for** statement.

C / C++

Fortran

- The associated *do-loops* must be structured blocks.
- No statement in the associated loops other than the DO statements can cause a branch out of the loops.

Fortran

Cross References

- The **single** construct, see Section 2.8.2 on page 89.
- The Worksharing-Loop construct, see Section 2.9.2 on page 101.
- SIMD directives, see Section 2.9.3 on page 110.
- **distribute** construct, see Section 2.9.4.1 on page 120.

1 2.9.6 scan Directive

2 Summary

3 The **scan** directive specifies that scan computations update the list items on each iteration.

4 Syntax

C / C++

5 The syntax of the **scan** directive is as follows:

```
6 loop-associated-directive  
7 for-loop-headers  
8 {  
9     structured-block  
10    #pragma omp scan clause new-line  
11    structured-block  
12 }
```

13 where *clause* is one of the following:

```
14    inclusive (list)  
15    exclusive (list)
```

16 and where *loop-associated-directive* is a **for**, **for simd**, or **simd** directive.

C / C++

Fortran

17 The syntax of the **scan** directive is as follows:

```
18 loop-associated-directive  
19 do-loop-headers  
20    structured-block  
21    !$omp scan clause  
22    structured-block  
23    do-termination-stmts(s)  
24    [end-loop-associated-directive]
```

25 where *clause* is one of the following:

```
26    inclusive (list)  
27    exclusive (list)
```

28 and where *loop-associated-directive* (*end-loop-associated-directive*) is a **do** (**end do**), **do simd**
29 (**end do simd**), or **simd** (**end simd**) directive.

Fortran

Description

The **scan** directive may appear in the body of a loop or loop nest associated with an enclosing worksharing-loop, worksharing-loop SIMD, or **simd** construct, to specify that a scan computation updates each list item on each loop iteration. The directive specifies that either an inclusive scan computation is to be performed for each list item that appears in an **inclusive** clause on the directive, or an exclusive scan computation is to be performed for each list item that appears in an **exclusive** clause on the directive. For each list item for which a scan computation is specified, statements that lexically precede or follow the directive constitute one of two phases for a given logical iteration of the loop – an *input phase* or a *scan phase*.

If the list item appears in an **inclusive** clause, all statements in the structured block that lexically precede the directive constitute the *input phase* and all statements in the structured block that lexically follow the directive constitute the *scan phase*. If the list item appears in an **exclusive** clause and the iteration is not the last iteration, all statements in the structured block that lexically precede the directive constitute the *scan phase* and all statements in the structured block that lexically follow the directive constitute the *input phase*. If the list item appears in an **exclusive** clause and the iteration is the last iteration, the iteration does not have an *input phase* and all statements that lexically precede or follow the directive constitute the *scan phase* for the iteration. The *input phase* contains all computations that update the list item in the iteration, and the *scan phase* ensures that any statement that reads the list item uses the result of the scan computation for that iteration.

The result of a scan computation for a given iteration is calculated according to the last *generalized prefix sum* ($\text{PRESUM}_{\text{last}}$) applied over the sequence of values given by the original value of the list item prior to the loop and all preceding updates to the list item in the logical iteration space of the loop. The operation $\text{PRESUM}_{\text{last}}(op, a_1, \dots, a_N)$ is defined for a given binary operator op and a sequence of N values a_1, \dots, a_N as follows:

- if $N = 1, a_1$
- if $N > 1, op(\text{PRESUM}_{\text{last}}(op, a_1, \dots, a_K), \text{PRESUM}_{\text{last}}(op, a_L, \dots, a_N))$, where $1 \leq K + 1 = L \leq N$.

At the beginning of the *input phase* of each iteration, the list item is initialized with the initializer value of the *reduction-identifier* specified by the **reduction** clause on the innermost enclosing construct. The *update value* of a list item is, for a given iteration, the value of the list item on completion of its *input phase*.

Let *orig-val* be the value of the original list item on entry to the enclosing worksharing-loop, worksharing-loop SIMD, or **simd** construct. Let *combiner* be the combiner for the *reduction-identifier* specified by the **reduction** clause on the construct. And let u_I be the update value of a list item for iteration I . For list items appearing in an **inclusive** clause on the **scan** directive, at the beginning of the *scan phase* for iteration I the list item is assigned the result of the operation $\text{PRESUM}_{\text{last}}(\text{combiner}, \text{orig-val}, u_0, \dots, u_I)$. For list items appearing in an **exclusive** clause on the **scan** directive, at the beginning of the *scan phase* for iteration $I = 0$

the list item is assigned the value *orig-val*, and at the beginning of the *scan phase* for iteration $I > 0$ the list item is assigned the result of the operation $\text{PRESUM}_{\text{last}}(\text{combiner}, \text{orig-val}, u_0, \dots, u_{I-1})$.

Restrictions

Restrictions to the **scan** directive are as follows:

- Exactly one **scan** directive must appear in the loop body of an enclosing worksharing-loop, worksharing-loop SIMD, or **simd** construct on which a **reduction** clause with the **inscan** modifier is present.
- A list item that appears in the **inclusive** or **exclusive** clause must appear in a **reduction** clause with the **inscan** modifier on the enclosing worksharing-loop, worksharing-loop SIMD, or **simd** construct.
- Cross-iteration dependences across different logical iterations must not exist, except for dependences for the list items specified in an **inclusive** or **exclusive** clause.
- Intra-iteration dependences from a statement in the structured block preceding a **scan** directive to a statement in the structured block following a **scan** directive must not exist, except for dependences for the list items specified in an **inclusive** or **exclusive** clause.

Cross References

- worksharing-loop construct, see Section [2.9.2](#) on page [101](#).
- **simd** construct, see Section [2.9.3.1](#) on page [110](#).
- worksharing-loop SIMD construct, see Section [2.9.3.2](#) on page [114](#).
- **reduction** clause, see Section [2.19.5.4](#) on page [300](#).

2.10 Tasking Constructs

2.10.1 task Construct

Summary

The **task** construct defines an explicit task.

Syntax

C / C++

The syntax of the **task** construct is as follows:

```
#pragma omp task [clause[ [, ] clause] ... ] new-line  
    structured-block
```

where *clause* is one of the following:

```
if([ task :] scalar-expression)  
final(scalar-expression)  
untied  
default(shared | none)  
mergeable  
private(list)  
firstprivate(list)  
shared(list)  
in_reduction(reduction-identifier : list)  
depend([depend-modifier, ] dependence-type : locator-list)  
priority(priority-value)  
allocate([allocator :] list)  
affinity([aff-modifier :] locator-list)  
detach(event-handle)
```

where *aff-modifier* is one of the following:

```
iterator(iterators-definition)
```

where *event-handle* is a variable of the **omp_event_handle_t** type.

C / C++

The syntax of the **task** construct is as follows:

```
!$omp task [clause[ , ] clause] ... ]
           structured-block
!$omp end task
```

where *clause* is one of the following:

```
if([ task :] scalar-logical-expression)
final(scalar-logical-expression)
untied
default(private | firstprivate | shared | none)
mergeable
private(list)
firstprivate(list)
shared(list)
in_reduction(reduction-identifier : list)
depend([depend-modifier,] dependence-type : locator-list)
priority(priority-value)
allocate([allocator :] list)
affinity([aff-modifier :] locator-list)
detach(event-handle)
```

where *aff-modifier* is one of the following:

```
iterator(iterators-definition)
```

where *event-handle* is an integer variable of **omp_event_handle_kind** kind.

Binding

The binding thread set of the **task** region is the current team. A **task** region binds to the innermost enclosing **parallel** region.

Description

The **task** construct is a *task generating construct*. When a thread encounters a **task** construct, an explicit task is generated from the code for the associated *structured-block*. The data environment of the task is created according to the data-sharing attribute clauses on the **task** construct, per-data environment ICVs, and any defaults that apply. The data environment of the task is destroyed when the execution code of the associated *structured-block* is completed.

The encountering thread may immediately execute the task, or defer its execution. In the latter case, any thread in the team may be assigned the task. Completion of the task can be guaranteed using task synchronization constructs. If a **task** construct is encountered during execution of an outer task, the generated **task** region corresponding to this construct is not a part of the outer task region unless the generated task is an included task.

If a **detach** clause is present on a **task** construct a new event *allow-completion-event* is created. The *allow-completion-event* is connected to the completion of the associated **task** region. The original *event-handle* will be updated to represent the *allow-completion-event* event before the task data environment is created. The *event-handle* will be considered as if it was specified on a **firstprivate** clause. The use of a variable in a **detach** clause expression of a **task** construct causes an implicit reference to the variable in all enclosing constructs.

If no **detach** clause is present on a **task** construct the generated **task** is completed when the execution of its associated *structured-block* is completed. If a **detach** clause is present on a **task** construct the task is completed when the execution of its associated *structured-block* is completed and the *allow-completion-event* is fulfilled.

When an **if** clause is present on a **task** construct, and the **if** clause expression evaluates to *false*, an undeferred task is generated, and the encountering thread must suspend the current task region, for which execution cannot be resumed until execution of the *structured block* that is associated with the generated task is completed. The use of a variable in an **if** clause expression of a **task** construct causes an implicit reference to the variable in all enclosing constructs.

When a **final** clause is present on a **task** construct and the **final** clause expression evaluates to *true*, the generated task will be a final task. All **task** constructs encountered during execution of a final task will generate final and included tasks. The use of a variable in a **final** clause expression of a **task** construct causes an implicit reference to the variable in all enclosing constructs. Encountering a **task** construct with the **detach** clause during the execution of a final task results in unspecified behavior.

The **if** clause expression and the **final** clause expression are evaluated in the context outside of the **task** construct, and no ordering of those evaluations is specified..

A thread that encounters a task scheduling point within the **task** region may temporarily suspend the **task** region. By default, a task is tied and its suspended **task** region can only be resumed by the thread that started its execution. If the **untied** clause is present on a **task** construct, any thread in the team can resume the **task** region after a suspension. The **untied** clause is ignored

if a **final** clause is present on the same **task** construct and the **final** clause expression evaluates to *true*, or if a task is an included task.

The **task** construct includes a task scheduling point in the task region of its generating task, immediately following the generation of the explicit task. Each explicit **task** region includes a task scheduling point at the end of its associated *structured-block*.

When the **mergeable** clause is present on a **task** construct, the generated task is a *mergeable task*.

The **priority** clause is a hint for the priority of the generated task. The *priority-value* is a non-negative integer expression that provides a hint for task execution order. Among all tasks ready to be executed, higher priority tasks (those with a higher numerical value in the **priority** clause expression) are recommended to execute before lower priority ones. The default *priority-value* when no **priority** clause is specified is zero (the lowest priority). If a value is specified in the **priority** clause that is higher than the *max-task-priority-var* ICV then the implementation will use the value of that ICV. A program that relies on task execution order being determined by this *priority-value* may have unspecified behavior.

The **affinity** clause is a hint to indicate data affinity of the generated task. The task is recommended to execute closely to the location of the list items. A program that relies on the task execution location being determined by this list may have unspecified behavior.

The list items that appear in the **affinity** clause may reference iterators defined by an *iterators-definition* appearing in the same clause. The list items that appear in the **affinity** clause may include array sections.

C / C++

The list items that appear in the **affinity** clause may use shape-operators.

C / C++

If a list item appears in an **affinity** clause then data affinity refers to the original list item.

Note – When storage is shared by an explicit **task** region, the programmer must ensure, by adding proper synchronization, that the storage does not reach the end of its lifetime before the explicit **task** region completes its execution.

Execution Model Events

The *task-create* event occurs when a thread encounters a construct that causes a new task to be created. The event occurs after the task is initialized but before it begins execution or is deferred.

Tool Callbacks

A thread dispatches a registered `ompt_callback_task_create` callback for each occurrence of a *task-create* event in the context of the encountering task. This callback has the type signature `ompt_callback_task_create_t` and the *flags* argument indicates the task types shown in Table 2.7.

TABLE 2.7: `ompt_callback_task_create` callback flags evaluation

Operation	Evaluates to true
$(flags \ \& \ ompt_task_explicit)$	Always in the dispatched callback
$(flags \ \& \ ompt_task_undelayed)$	If the task is an undelayed task
$(flags \ \& \ ompt_task_final)$	If the task is a final task
$(flags \ \& \ ompt_task_untied)$	If the task is an untied task
$(flags \ \& \ ompt_task_mergeable)$	If the task is a mergeable task
$(flags \ \& \ ompt_task_merged)$	If the task is a merged task

Restrictions

Restrictions to the `task` construct are as follows:

- A program that branches into or out of a `task` region is non-conforming.
- A program must not depend on any ordering of the evaluations of the clauses of the `task` directive, or on any side effects of the evaluations of the clauses.
- At most one `if` clause can appear on the directive.
- At most one `final` clause can appear on the directive.
- At most one `priority` clause can appear on the directive.
- At most one `detach` clause can appear on the directive.
- If a `detach` clause appears on the directive, then a `mergeable` clause cannot appear on the same directive.

C / C++

- A throw executed inside a `task` region must cause execution to resume within the same `task` region, and the same thread that threw the exception must catch it.

C / C++

Cross References

- Task scheduling constraints, see Section 2.10.6 on page 149.
- **allocate** clause, see Section 2.11.4 on page 158.
- **if** clause, see Section 2.15 on page 220.
- **depend** clause, see Section 2.17.11 on page 255.
- Data-sharing attribute clauses, Section 2.19.4 on page 282.
- **default** clause, see Section 2.19.4.1 on page 282.
- **in_reduction** clause, see Section 2.19.5.6 on page 303.
- **omp_fulfill_event**, see Section 3.5.1 on page 396.
- **ompt_callback_task_create_t**, see Section 4.5.2.7 on page 467.

2.10.2 taskloop Construct

Summary

The **taskloop** construct specifies that the iterations of one or more associated loops will be executed in parallel using explicit tasks. The iterations are distributed across tasks generated by the construct and scheduled to be executed.

Syntax

C / C++

The syntax of the **taskloop** construct is as follows:

```
#pragma omp taskloop [clause[[,] clause] ...] new-line  
    for-loops
```

where *clause* is one of the following:

```
if([ taskloop :] scalar-expression)  
shared(list)  
private(list)  
firstprivate(list)  
lastprivate(list)  
reduction([default ,]reduction-identifier : list)  
in_reduction(reduction-identifier : list)
```



```

1      default(shared | none)
2      grainsize(grain-size)
3      num_tasks(num-tasks)
4      collapse(n)
5      final(scalar-expr)
6      priority(priority-value)
7      untied
8      mergeable
9      nogroup
10     allocate([allocator :] list)

```

The **taskloop** directive places restrictions on the structure of all associated *for-loops*. Specifically, all associated *for-loops* must have *canonical loop form* (see Section 2.9.1 on page 95).



The syntax of the **taskloop** construct is as follows:

```

14     !$omp taskloop [clause[[,] clause] ...]
15         do-loops
16     [!$omp end taskloop]

```

where *clause* is one of the following:

```

18     if([ taskloop :] scalar-logical-expression)
19     shared(list)
20     private(list)
21     firstprivate(list)
22     lastprivate(list)
23     reduction([default ,] reduction-identifier : list)
24     in_reduction(reduction-identifier : list)
25     default(private | firstprivate | shared | none)
26     grainsize(grain-size)
27     num_tasks(num-tasks)
28     collapse(n)
29     final(scalar-logical-expr)
30     priority(priority-value)

```

```

1      untied
2      mergeable
3      nogroup
4      allocate ([allocator :] list)

```

If an **end taskloop** directive is not specified, an **end taskloop** directive is assumed at the end of the *do-loops*.

The **taskloop** directive places restrictions on the structure of all associated *do-loops*. Specifically, all associated *do-loops* must have *canonical loop form* (see Section 2.9.1 on page 95).

Fortran

Binding

The binding thread set of the **taskloop** region is the current team. A **taskloop** region binds to the innermost enclosing **parallel** region.

Description

The **taskloop** construct is a *task generating construct*. When a thread encounters a **taskloop** construct, the construct partitions the iterations of the associated loops into explicit tasks for parallel execution. The data environment of each generated task is created according to the data-sharing attribute clauses on the **taskloop** construct, per-data environment ICVs, and any defaults that apply. The order of the creation of the loop tasks is unspecified. Programs that rely on any execution order of the logical loop iterations are non-conforming.

By default, the **taskloop** construct executes as if it was enclosed in a **taskgroup** construct with no statements or directives outside of the **taskloop** construct. Thus, the **taskloop** construct creates an implicit **taskgroup** region. If the **nogroup** clause is present, no implicit **taskgroup** region is created.

If a **reduction** clause is present on the **taskloop** construct, the behavior is as if a **task_reduction** clause with the same reduction operator and list items was applied to the implicit **taskgroup** construct enclosing the **taskloop** construct. The **taskloop** construct executes as if each generated task was defined by a **task** construct on which an **in_reduction** clause with the same reduction operator and list items is present. Thus, the generated tasks are participants of the reduction defined by the **task_reduction** clause that was applied to the implicit **taskgroup** construct.

If an **in_reduction** clause is present on the **taskloop** construct, the behavior is as if each generated task was defined by a **task** construct on which an **in_reduction** clause with the same reduction operator and list items is present. Thus, the generated tasks are participants of a reduction previously defined by a reduction scoping clause.

If a **grainsize** clause is present on the **taskloop** construct, the number of logical loop iterations assigned to each generated task is greater than or equal to the minimum of the value of the *grain-size* expression and the number of logical loop iterations, but less than two times the value of the *grain-size* expression.

The parameter of the **grainsize** clause must be a positive integer expression. If **num_tasks** is specified, the **taskloop** construct creates as many tasks as the minimum of the *num-tasks* expression and the number of logical loop iterations. Each task must have at least one logical loop iteration. The parameter of the **num_tasks** clause must be a positive integer expression. If neither a **grainsize** nor **num_tasks** clause is present, the number of loop tasks generated and the number of logical loop iterations assigned to these tasks is implementation defined.

The **collapse** clause may be used to specify how many loops are associated with the **taskloop** construct. The parameter of the **collapse** clause must be a constant positive integer expression. If no **collapse** clause is present or its parameter is 1, the only loop that is associated with the **taskloop** construct is the one that immediately follows the **taskloop** directive. If a **collapse** clause is specified with a parameter value greater than 1 and more than one loop is associated with the **taskloop** construct, then the iterations of all associated loops are collapsed into one larger iteration space that is then divided according to the **grainsize** and **num_tasks** clauses. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

If more than one loop is associated with the **taskloop** construct then the number of times that any intervening code between any two associated loops will be executed is unspecified but will be at least once per iteration of the loop enclosing the intervening code and at most once per iteration of the innermost loop associated with the construct. If the iteration count of any loop that is associated with the **taskloop** construct is zero and that loop does not enclose intervening code, the behavior is unspecified.

A taskloop loop has logical iterations numbered 0,1,...,N-1 where N is the number of loop iterations, and the logical numbering denotes the sequence in which the iterations would be executed if the set of associated loop(s) were executed sequentially. At the beginning of each logical iteration, the loop iteration variable of each associated loop has the value that it would have if the set of the associated loop(s) were executed sequentially.

The iteration count for each associated loop is computed before entry to the outermost loop. If execution of any associated loop changes any of the values used to compute any of the iteration counts, then the behavior is unspecified.

The integer type (or kind, for Fortran) used to compute the iteration count for the collapsed loop is implementation defined.

When an **if** clause is present on a **taskloop** construct, and if the **if** clause expression evaluates to *false*, undeferred tasks are generated. The use of a variable in an **if** clause expression of a **taskloop** construct causes an implicit reference to the variable in all enclosing constructs.

When a **final** clause is present on a **taskloop** construct and the **final** clause expression evaluates to *true*, the generated tasks will be final tasks. The use of a variable in a **final** clause expression of a **taskloop** construct causes an implicit reference to the variable in all enclosing constructs.

When a **priority** clause is present on a **taskloop** construct, the generated tasks use the *priority-value* as if it was specified for each individual task. If the **priority** clause is not specified, tasks generated by the **taskloop** construct have the default task priority (zero).

If the **untied** clause is specified, all tasks generated by the **taskloop** construct are untied tasks.

When the **mergeable** clause is present on a **taskloop** construct, each generated task is a *mergeable task*.

C++

For **firstprivate** variables of class type, the number of invocations of copy constructors to perform the initialization is implementation-defined.

C++

Note – When storage is shared by a **taskloop** region, the programmer must ensure, by adding proper synchronization, that the storage does not reach the end of its lifetime before the **taskloop** region and its descendant tasks complete their execution.

Execution Model Events

The *taskloop-begin* event occurs after a task encounters a **taskloop** construct but before any other events that may trigger as a consequence of executing the **taskloop**. Specifically, a *taskloop-begin* event for a **taskloop** will precede the *taskgroup-begin* that occurs unless a **nogroup** clause is present. Regardless of whether an implicit taskgroup is present, a *taskloop-begin* will always precede any *task-create* events for generated tasks.

The *taskloop-end* event occurs after a **taskloop** region finishes execution but before resuming execution of the encountering task.

The *taskloop-iteration-begin* event occurs before an explicit task executes each iteration of a **taskloop**.

Tool Callbacks

A thread dispatches a registered `ompt_callback_work` callback for each occurrence of a *taskloop-begin* and *taskloop-end* event in that thread. The callback occurs in the context of the encountering task. The callback has type signature `ompt_callback_work_t`. The callback receives `ompt_scope_begin` or `ompt_scope_end` as its *endpoint* argument, as appropriate, and `ompt_work_taskloop` as its *wstype* argument.

A thread dispatches a registered `ompt_callback_dispatch` callback for each occurrence of a *taskloop-iteration-begin* event in that thread. The callback occurs in the context of the encountering task. The callback has type signature `ompt_callback_dispatch_t`.

Restrictions

The restrictions of the **taskloop** construct are as follows:

- A program that branches into or out of a **taskloop** region is non-conforming.
- No OpenMP directive may appear in the region between any associated loops.
- If a **collapse** clause is specified, exactly one loop must occur in the region at each nesting level up to the number of loops specified by the parameter of the **collapse** clause.
- If a **reduction** clause is present on the **taskloop** directive, the **nogroup** clause must not be specified.
- The same list item cannot appear in both a **reduction** and an **in_reduction** clause.
- At most one **grainsize** clause can appear on a **taskloop** directive.
- At most one **num_tasks** clause can appear on a **taskloop** directive.
- The **grainsize** clause and **num_tasks** clause are mutually exclusive and may not appear on the same **taskloop** directive.
- At most one **collapse** clause can appear on a **taskloop** directive.
- At most one **if** clause can appear on the directive.
- At most one **final** clause can appear on the directive.
- At most one **priority** clause can appear on the directive.

Cross References

- **task** construct, Section [2.10.1](#) on page [135](#).
- **if** clause, see Section [2.15](#) on page [220](#).
- **taskgroup** construct, Section [2.17.6](#) on page [232](#).
- Data-sharing attribute clauses, Section [2.19.4](#) on page [282](#).

- **default** clause, see Section 2.19.4.1 on page 282.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.4.4.11 on page 443.
- **ompt_work_taskloop**, see Section 4.4.4.15 on page 445.
- **ompt_callback_work_t**, see Section 4.5.2.5 on page 464.
- **ompt_callback_dispatch_t**, see Section 4.5.2.6 on page 465.

2.10.3 taskloop simd Construct

Summary

The **taskloop simd** construct specifies a loop that can be executed concurrently using SIMD instructions and that those iterations will also be executed in parallel using explicit tasks. The **taskloop simd** construct is a composite construct.

Syntax

C / C++

The syntax of the **taskloop simd** construct is as follows:

```
#pragma omp taskloop simd [clause[, ] clause] ...] new-line
    for-loops
```

where *clause* can be any of the clauses accepted by the **taskloop** or **simd** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **taskloop simd** construct is as follows:

```
!$omp taskloop simd [clause[, ] clause] ...]
    do-loops
[!$omp end taskloop simd]
```

where *clause* can be any of the clauses accepted by the **taskloop** or **simd** directives with identical meanings and restrictions.

If an **end taskloop simd** directive is not specified, an **end taskloop simd** directive is assumed at the end of the *do-loops*.

Fortran

Binding

The binding thread set of the **taskloop simd** region is the current team. A **taskloop simd** region binds to the innermost enclosing parallel region.

Description

The **taskloop simd** construct will first distribute the iterations of the associated loop(s) across tasks in a manner consistent with any clauses that apply to the **taskloop** construct. The resulting tasks will then be converted to a SIMD loop in a manner consistent with any clauses that apply to the **simd** construct, except for the **collapse** clause. For the purposes of each task's conversion to a SIMD loop, the **collapse** clause is ignored and the effect of any **in_reduction** clause is as if a **reduction** clause with the same reduction operator and list items is present on the construct.

Execution Model Events

This composite construct generates the same events as the **taskloop** construct.

Tool Callbacks

This composite construct dispatches the same callbacks as the **taskloop** construct.

Restrictions

- The restrictions for the **taskloop** and **simd** constructs apply.
- The **conditional** modifier may not appear in a **lastprivate** clause.

Cross References

- **simd** construct, see Section [2.9.3.1](#) on page [110](#).
- **taskloop** construct, see Section [2.10.2](#) on page [140](#).
- Data-sharing attribute clauses, see Section [2.19.4](#) on page [282](#).

2.10.4 taskyield Construct

Summary

The **taskyield** construct specifies that the current task can be suspended in favor of execution of a different task. The **taskyield** construct is a stand-alone directive.

Syntax

C / C++

The syntax of the **taskyield** construct is as follows:

```
#pragma omp taskyield new-line
```

C / C++

Fortran

The syntax of the **taskyield** construct is as follows:

```
!$omp taskyield
```

Fortran

Binding

A **taskyield** region binds to the current task region. The binding thread set of the **taskyield** region is the current team.

Description

The **taskyield** region includes an explicit task scheduling point in the current task region.

Cross References

- Task scheduling, see Section [2.10.6](#) on page [149](#).

2.10.5 Initial Task

Execution Model Events

No events are associated with the implicit parallel region in each initial thread.

The *initial-thread-begin* event occurs in an initial thread after the OpenMP runtime invokes the tool initializer but before the initial thread begins to execute the first OpenMP region in the initial task.

The *initial-task-begin* event occurs after an *initial-thread-begin* event but before the first OpenMP region in the initial task begins to execute.

The *initial-task-end* event occurs before an *initial-thread-end* event but after the last OpenMP region in the initial task finishes to execute.

The *initial-thread-end* event occurs as the final event in an initial thread at the end of an initial task immediately prior to invocation of the tool finalizer.

Tool Callbacks

A thread dispatches a registered `ompt_callback_thread_begin` callback for the *initial-thread-begin* event in an initial thread. The callback occurs in the context of the initial thread. The callback has type signature `ompt_callback_thread_begin_t`. The callback receives `ompt_thread_initial` as its *thread_type* argument.

A thread dispatches a registered `ompt_callback_implicit_task` callback with `ompt_scope_begin` as its *endpoint* argument for each occurrence of an *initial-task-begin* in that thread. Similarly, a thread dispatches a registered `ompt_callback_implicit_task` callback with `ompt_scope_end` as its *endpoint* argument for each occurrence of an *initial-task-end* event in that thread. The callbacks occur in the context of the initial task and have type signature `ompt_callback_implicit_task_t`. In the dispatched callback, `(flag & ompt_task_initial)` always evaluates to *true*.

A thread dispatches a registered `ompt_callback_thread_end` callback for the *initial-thread-end* event in that thread. The callback occurs in the context of the thread. The callback has type signature `ompt_callback_thread_end_t`. The implicit parallel region does not dispatch a `ompt_callback_parallel_end` callback; however, the implicit parallel region can be finalized within this `ompt_callback_thread_end` callback.

Cross References

- `ompt_thread_initial`, see Section 4.4.4.10 on page 443.
- `ompt_task_initial`, see Section 4.4.4.18 on page 446.
- `ompt_callback_thread_begin_t`, see Section 4.5.2.1 on page 459.
- `ompt_callback_thread_end_t`, see Section 4.5.2.2 on page 460.
- `ompt_callback_parallel_begin_t`, see Section 4.5.2.3 on page 461.
- `ompt_callback_parallel_end_t`, see Section 4.5.2.4 on page 463.
- `ompt_callback_implicit_task_t`, see Section 4.5.2.11 on page 471.

2.10.6 Task Scheduling

Whenever a thread reaches a task scheduling point, the implementation may cause it to perform a task switch, beginning or resuming execution of a different task bound to the current team. Task scheduling points are implied at the following locations:

- during the generation of an explicit task;
- the point immediately following the generation of an explicit task;

- after the point of completion of the structured block associated with a task;
- in a **taskyield** region;
- in a **taskwait** region;
- at the end of a **taskgroup** region;
- in an implicit barrier region;
- in an explicit **barrier** region;
- during the generation of a **target** region;
- the point immediately following the generation of a **target** region;
- at the beginning and end of a **target data** region;
- in a **target update** region;
- in a **target enter data** region;
- in a **target exit data** region;
- in the **omp_target_memcpy** routine;
- in the **omp_target_memcpy_rect** routine;

When a thread encounters a task scheduling point it may do one of the following, subject to the *Task Scheduling Constraints* (below):

- begin execution of a tied task bound to the current team;
- resume any suspended task region, bound to the current team, to which it is tied;
- begin execution of an untied task bound to the current team; or
- resume any suspended untied task region bound to the current team.

If more than one of the above choices is available, it is unspecified as to which will be chosen.

Task Scheduling Constraints are as follows:

1. Scheduling of new tied tasks is constrained by the set of task regions that are currently tied to the thread and that are not suspended in a barrier region. If this set is empty, any new tied task may be scheduled. Otherwise, a new tied task may be scheduled only if it is a descendent task of every task in the set.
2. A dependent task shall not start its execution until its task dependences are fulfilled.
3. A task shall not be scheduled while any task with which it is mutually exclusive has been scheduled, but has not yet completed.

4. When an explicit task is generated by a construct containing an **if** clause for which the expression evaluated to *false*, and the previous constraints are already met, the task is executed immediately after generation of the task.

A program relying on any other assumption about task scheduling is non-conforming.

Note – Task scheduling points dynamically divide task regions into parts. Each part is executed uninterrupted from start to end. Different parts of the same task region are executed in the order in which they are encountered. In the absence of task synchronization constructs, the order in which a thread executes parts of different schedulable tasks is unspecified.

A program must behave correctly and consistently with all conceivable scheduling sequences that are compatible with the rules above.

For example, if **threadprivate** storage is accessed (explicitly in the source code or implicitly in calls to library routines) in one part of a task region, its value cannot be assumed to be preserved into the next part of the same task region if another schedulable task exists that modifies it.

As another example, if a lock acquire and release happen in different parts of a task region, no attempt should be made to acquire the same lock in any part of another task that the executing thread may schedule. Otherwise, a deadlock is possible. A similar situation can occur when a **critical** region spans multiple parts of a task and another schedulable task contains a **critical** region with the same name.

The use of threadprivate variables and the use of locks or critical sections in an explicit task with an **if** clause must take into account that when the **if** clause evaluates to *false*, the task is executed immediately, without regard to *Task Scheduling Constraint 2*.

Execution Model Events

The *task-schedule* event occurs in a thread when the thread switches tasks at a task scheduling point; no event occurs when switching to or from a merged task.

Tool Callbacks

A thread dispatches a registered **ompt_callback_task_schedule** callback for each occurrence of a *task-schedule* event in the context of the task that begins or resumes. This callback has the type signature **ompt_callback_task_schedule_t**. The argument *prior_task_status* is used to indicate the cause for suspending the prior task. This cause may be the completion of the prior task region, the encountering of a **taskyield** construct, or the encountering of an active cancellation point.

Cross References

- **ompt_callback_task_schedule_t**, see Section 4.5.2.10 on page 470.

1 **2.11 Memory Management Directives**

2 **2.11.1 Memory Spaces**

3 OpenMP memory spaces represent storage resources where variables can be stored and retrieved.
4 Table 2.8 shows the list of predefined memory spaces. The selection of a given memory space
5 expresses an intent to use storage with certain traits for the allocations. The actual storage resources
6 that each memory space represents are implementation defined.

TABLE 2.8: Predefined Memory Spaces

Memory space name	Storage selection intent
<code>omp_default_mem_space</code>	Represents the system default storage.
<code>omp_large_cap_mem_space</code>	Represents storage with large capacity.
<code>omp_const_mem_space</code>	Represents storage optimized for variables with constant values. The result of writing to this storage is unspecified.
<code>omp_high_bw_mem_space</code>	Represents storage with high bandwidth.
<code>omp_low_lat_mem_space</code>	Represents storage with low latency.

7 ▼
8 Note – For variables allocated in the `omp_const_mem_space` memory space OpenMP
9 supports initializing constant memory either by means of the **firstprivate** clause or through
10 initialization with compile time constants for static and constant variables. Implementation-defined
11 mechanisms to provide the constant value of these variables may also be supported.
12 ▲

13 **Cross References**

- 14 • `omp_init_allocator` routine, see Section 3.7.2 on page 409.

15 **2.11.2 Memory Allocators**

16 OpenMP memory allocators can be used by a program to make allocation requests. When a
17 memory allocator receives a request to allocate storage of a certain size, an allocation of logically
18 consecutive *memory* in the resources of its associated memory space of at least the size that was
19 requested will be returned if possible. This allocation will not overlap with any other existing
20 allocation from an OpenMP memory allocator.

The behavior of the allocation process can be affected by the allocator traits that the user specifies. Table 2.9 shows the allowed allocators traits, their possible values and the default value of each trait.

TABLE 2.9: Allocator Traits

Allocator trait	Allowed values	Default value
<code>sync_hint</code>	<code>contended</code> , <code>uncontended</code> , <code>serialized</code> , <code>private</code>	<code>contended</code>
<code>alignment</code>	A positive integer value that is a power of 2	1 byte
<code>access</code>	<code>all</code> , <code>cgroup</code> , <code>pteam</code> , <code>thread</code>	<code>all</code>
<code>pool_size</code>	Positive integer value	Implementation defined
<code>fallback</code>	<code>default_mem_fb</code> , <code>null_fb</code> , <code>abort_fb</code> , <code>allocator_fb</code>	<code>default_mem_fb</code>
<code>fb_data</code>	an allocator handle	(none)
<code>pinned</code>	<code>true</code> , <code>false</code>	<code>false</code>
<code>partition</code>	<code>environment</code> , <code>nearest</code> , <code>blocked</code> , <code>interleaved</code>	<code>environment</code>

The `sync_hint` trait describes the expected manner in which multiple threads may use the allocator. The values and their description are:

- **contended**: high contention is expected on the allocator; that is, many threads are expected to request allocations simultaneously.
- **uncontended**: low contention is expected on the allocator; that is, few threads are expected to request allocations simultaneously.
- **serialized**: only one thread at a time will request allocations with the allocator. Requesting two allocations simultaneously when specifying **serialized** results in unspecified behavior.
- **private**: the same thread will request allocations with the allocator every time. Requesting an allocation from different threads, simultaneously or not, when specifying **private** results in unspecified behavior.

Allocated memory will be byte aligned to at least the value specified for the `alignment` trait of the allocator. Some directives and API routines can specify additional requirements on alignment beyond those described in this section.

Memory allocated by allocators with the `access` trait defined to be **all** must be accessible by all threads in the device where the allocation was requested. Memory allocated by allocators with the `access` trait defined to be **cgroup** will be memory accessible by all threads in the same

contention group as the thread that requested the allocation. Attempts to access the memory returned by an allocator with the **access** trait defined to be **cgroup** from a thread that is not part of the same contention group as the thread that allocated the memory result in unspecified behavior. Memory allocated by allocators with the **access** trait defined to be **pteam** will be memory accessible by all threads that bind to the same **parallel** region of the thread that requested the allocation. Attempts to access the memory returned by an allocator with the **access** trait defined to be **pteam** from a thread that does not bind to the same **parallel** region as the thread that allocated the memory result in unspecified behavior. Memory allocated by allocator with the **access** trait defined to be **thread** will be memory accessible by the *thread* that requested the allocation. Attempts to access the memory returned by an allocator with the **access** trait defined to be **thread** from a thread other than the one that allocated the memory result in unspecified behavior.

The total amount of storage in bytes that an allocator can use is limited by the **pool_size** trait. For allocators with the **access** trait defined to be **all**, this limit refers to allocations from all threads that access the allocator. For allocators with the **access** trait defined to be **cgroup**, this limit refers to allocations from threads that access the allocator from the same contention group. For allocators with the **access** trait defined to be **pteam**, this limit refers to allocations from threads that access the allocator from the same parallel team. For allocators with the **access** trait defined to be **thread**, this limit refers to allocations from each thread that access the allocator. Requests that would result in using more storage than **pool_size** will not be fulfilled by the allocator.

The **fallback** trait specifies how the allocator behaves when it cannot fulfill an allocation request. If the **fallback** trait is set to **null_fb**, the allocator returns the value zero if it fails to allocate the memory. If the **fallback** trait is set to **abort_fb**, program execution will be terminated if the allocation fails. If the **fallback** trait is set to **allocator_fb** then when an allocation fails the request will be delegated to the allocator specified in the **fb_data** trait. If the **fallback** trait is set to **default_mem_fb** then when an allocation fails another allocation will be tried in the **omp_default_mem_space** memory space, which assumes all allocator traits to be set to their default values except for **fallback** trait which will be set to **null_fb**.

Allocators with the **pinned** trait defined to be **true** ensure that their allocations remain in the same storage resource at the same location for their entire lifetime.

The **partition** trait describes the partitioning of allocated memory over the storage resources represented by the memory space associated with the allocator. The partitioning will be done in parts with a minimum size that is implementation defined. The values are:

- **environment**: the placement of allocated memory is determined by the execution environment.
- **nearest**: allocated memory is placed in the storage resource that is nearest to the thread that requests the allocation.
- **blocked**: allocated memory is partitioned into parts of approximately the same size with at most one part per storage resource.

- **interleaved:** allocated memory parts are distributed in a round-robin fashion across the storage resources.
- Table 2.10 shows the list of predefined memory allocators and their associated memory spaces. The predefined memory allocators have default values for their allocator traits unless otherwise specified.

TABLE 2.10: Predefined Allocators

Allocator name	Associated memory space	Non-default trait values
<code>omp_default_mem_alloc</code>	<code>omp_default_mem_space</code>	(none)
<code>omp_large_cap_mem_alloc</code>	<code>omp_large_cap_mem_space</code>	(none)
<code>omp_const_mem_alloc</code>	<code>omp_const_mem_space</code>	(none)
<code>omp_high_bw_mem_alloc</code>	<code>omp_high_bw_mem_space</code>	(none)
<code>omp_low_lat_mem_alloc</code>	<code>omp_low_lat_mem_space</code>	(none)
<code>omp_cgroup_mem_alloc</code>	Implementation defined	access:cgroup
<code>omp_pteam_mem_alloc</code>	Implementation defined	access:pteam
<code>omp_thread_mem_alloc</code>	Implementation defined	access:thread

Fortran

If any operation of the base language causes a reallocation of an array that is allocated with a memory allocator then that memory allocator will be used to release the current memory and to allocate the new memory.

Fortran

Cross References

- `omp_init_allocator` routine, see Section 3.7.2 on page 409.
- `omp_destroy_allocator` routine, see Section 3.7.3 on page 410.
- `omp_set_default_allocator` routine, see Section 3.7.4 on page 411.
- `omp_get_default_allocator` routine, see Section 3.7.5 on page 412.
- `OMP_ALLOCATOR` environment variable, see Section 6.21 on page 618.

2.11.3 allocate Directive

Summary

The **allocate** directive specifies how a set of variables are allocated. The **allocate** directive is a declarative directive if it is not associated with an allocation statement.

Syntax

C / C++

The syntax of the **allocate** directive is as follows:

```
#pragma omp allocate (list) [clause] new-line
```

where *clause* is one of the following:

```
allocator (allocator)
```

where *allocator* is an expression of **omp_allocator_handle_t** type.

C / C++

Fortran

The syntax of the **allocate** directive is as follows:

```
!$omp allocate (list) [clause]
```

or

```
!$omp allocate [ (list) ] clause  
[!$omp allocate (list) clause  
[...]]  
allocate statement
```

where *clause* is one of the following:

```
allocator (allocator)
```

where *allocator* is an integer expression of **omp_allocator_handle_kind** kind.

Fortran

1

2
3
4
5
6
7

89

Fortran

10
11
12
13

Fortran

14
15

16

- 17
18
19
20
21
22
23
24

C / C++

- 25
26
27
28

_____ C / C++ _____

Fortran

- List items specified in the **allocate** directive must not have the **ALLOCATABLE** attribute unless the directive is associated with an *allocate statement*.
- List items specified in an **allocate** directive that is associated with an *allocate statement* must be variables that are allocated by the *allocate statement*.
- Multiple directives can only be associated with an *allocate statement* if list items are specified on each **allocate** directive.
- If a list item has the **SAVE** attribute, is a common block name, or is declared in the scope of a module, then only predefined memory allocator parameters can be used in the **allocator** clause.
- A type parameter inquiry cannot appear in an **allocate** directive.

Fortran

Cross References

- *def-allocator-var* ICV, see Section [2.5.1](#) on page [64](#).
- Memory allocators, see Section [2.11.2](#) on page [152](#).
- **omp_allocator_handle_t** and **omp_allocator_handle_kind**, see Section [3.7.1](#) on page [406](#).

2.11.4 allocate Clause

Summary

The **allocate** clause specifies the memory allocator to be used to obtain storage for private variables of a directive.

Syntax

The syntax of the **allocate** clause is as follows:

```
allocate ([allocator:] list)
```

	C / C++
1	where <i>allocator</i> is an expression of the <code>omp_allocator_handle_t</code> type.
	C / C++
	Fortran
2	where <i>allocator</i> is an integer expression of the <code>omp_allocator_handle_kind</code> <i>kind</i> .
	Fortran

Description

The storage for new list items that arise from list items that appear in the directive will be provided through a memory allocator. If an *allocator* is specified in the clause, that allocator will be used for allocations. For all directives except the **target** directive, if no *allocator* is specified in the clause then the memory allocator that is specified by the *def-allocator-var* ICV will be used for the list items that are specified in the **allocate** clause. The allocation of each *list item* will be byte aligned to at least the alignment required by the base language for the type of that *list item*.

For allocations that arise from this clause the **null_fb** value of the fallback allocator trait will behave as if the **abort_fb** had been specified.

Restrictions

- For any list item that is specified in the **allocate** clause on a directive, a data-sharing attribute clause that may create a private copy of that list item must be specified on the same directive.
- For **task**, **taskloop** or **target** directives, allocation requests to memory allocators with the trait **access** set to **thread** result in unspecified behavior.
- **allocate** clauses that appear on a **target** construct or on constructs in a **target** region must specify an *allocator* expression unless a **requires** directive with the **dynamic_allocators** clause is present in the same compilation unit.

Cross References

- *def-allocator-var* ICV, see Section 2.5.1 on page 64.
- Memory allocators, see Section 2.11.2 on page 152.
- `omp_allocator_handle_t` and `omp_allocator_handle_kind`, see Section 3.7.1 on page 406.

1 2.12 Device Directives

2 2.12.1 Device Initialization

3 Execution Model Events

4 The *device-initialize* event occurs in a thread that encounters the first **target**, **target data**, or
5 **target enter data** construct or a device memory routine that is associated with a particular
6 target device after the thread initiates initialization of OpenMP on the device and the device's
7 OpenMP initialization, which may include device-side tool initialization, completes.

8 The *device-load* event for a code block for a target device occurs in some thread before any thread
9 executes code from that code block on that target device.

10 The *device-unload* event for a target device occurs in some thread whenever a code block is
11 unloaded from the device.

12 The *device-finalize* event for a target device that has been initialized occurs in some thread before
13 an OpenMP implementation shuts down.

14 Tool Callbacks

15 A thread dispatches a registered **ompt_callback_device_initialize** callback for each
16 occurrence of a *device-initialize* event in that thread. This callback has type signature
17 **ompt_callback_device_initialize_t**.

18 A thread dispatches a registered **ompt_callback_device_load** callback for each occurrence
19 of a *device-load* event in that thread. This callback has type signature
20 **ompt_callback_device_load_t**.

21 A thread dispatches a registered **ompt_callback_device_unload** callback for each
22 occurrence of a *device-unload* event in that thread. This callback has type signature
23 **ompt_callback_device_unload_t**.

24 A thread dispatches a registered **ompt_callback_device_finalize** callback for each
25 occurrence of a *device-finalize* event in that thread. This callback has type signature
26 **ompt_callback_device_finalize_t**.

27 Restrictions

28 No thread may offload execution of an OpenMP construct to a device until a dispatched
29 **ompt_callback_device_initialize** callback completes.

30 No thread may offload execution of an OpenMP construct to a device after a dispatched
31 **ompt_callback_device_finalize** callback occurs.

Cross References

- `ompt_callback_device_load_t`, see Section 4.5.2.21 on page 484.
- `ompt_callback_device_unload_t`, see Section 4.5.2.22 on page 486.
- `ompt_callback_device_initialize_t`, see Section 4.5.2.19 on page 482.
- `ompt_callback_device_finalize_t`, see Section 4.5.2.20 on page 484.

2.12.2 target data Construct

Summary

Map variables to a device data environment for the extent of the region.

Syntax

C / C++

The syntax of the **target data** construct is as follows:

```
#pragma omp target data clause[ [ [, ] clause] ... ] new-line
    structured-block
```

where *clause* is one of the following:

```
if([ target data :] scalar-expression)
device(integer-expression)
map([[map-type-modifier[, ] [map-type-modifier[, ] ...] map-type: ] locator-list)
use_device_ptr(ptr-list)
use_device_addr(list)
```

C / C++

Fortran

The syntax of the **target data** construct is as follows:

```
!$omp target data clause[ [ [, ] clause] ... ]
    structured-block
!$omp end target data
```

where *clause* is one of the following:

```
if ([ target data :] scalar-logical-expression)  
device (scalar-integer-expression)  
map ([map-type-modifier[ , ] [map-type-modifier[ , ] ...] map-type : ] locator-list)  
use_device_ptr (ptr-list)  
use_device_addr (list)
```

Fortran

Binding

The binding task set for a **target data** region is the generating task. The **target data** region binds to the region of the generating task.

Description

When a **target data** construct is encountered, the encountering task executes the region. If there is no **device** clause, the default device is determined by the *default-device-var* ICV. When an **if** clause is present and the **if** clause expression evaluates to *false*, the device is the host. Variables are mapped for the extent of the region, according to any data-mapping attribute clauses, from the data environment of the encountering task to the device data environment.

Pointers that appear in a **use_device_ptr** clause are privatized and the device pointers to the corresponding list items in the device data environment are assigned into the private versions.

List items that appear in a **use_device_addr** clause have the address of the corresponding object in the device data environment inside the construct. For objects, any reference to the value of the object will be to the corresponding object on the device, while references to the address will result in a valid device address that points to that object. Array sections privatize the base of the array section and assign the private copy to the address of the corresponding array section in the device data environment.

If one or more of the **use_device_ptr** or **use_device_addr** clauses and one or more **map** clauses are present on the same construct, the address conversions of **use_device_addr** and **use_device_ptr** clauses will occur as if performed after all variables are mapped according to those **map** clauses.

Execution Model Events

The events associated with entering a target data region are the same events as associated with a target enter data construct, described in Section 2.12.3 on page 164.

The events associated with exiting a target data region are the same events as associated with a target exit data construct, described in Section 2.12.4 on page 166.

Tool Callbacks

The tool callbacks dispatched when entering a target data region are the same as the tool callbacks dispatched when encountering a target enter data construct, described in Section 2.12.3 on page 164.

The tool callbacks dispatched when exiting a target data region are the same as the tool callbacks dispatched when encountering a target exit data construct, described in Section 2.12.4 on page 166.

Restrictions

- A program must not depend on any ordering of the evaluations of the clauses of the **target data** directive, except as explicitly stated for **map** clauses relative to **use_device_ptr** and **use_device_addr** clauses, or on any side effects of the evaluations of the clauses.
- At most one **device** clause can appear on the directive. The **device** clause expression must evaluate to a non-negative integer value less than the value of **omp_get_num_devices()** or to the value of **omp_get_initial_device()**.
- At most one **if** clause can appear on the directive.
- A *map-type* in a **map** clause must be **to**, **from**, **tofrom** or **alloc**.
- At least one **map**, **use_device_addr** or **use_device_ptr** clause must appear on the directive.
- A list item in a **use_device_ptr** clause must hold the address of an object that has a corresponding list item in the device data environment.
- A list item in a **use_device_addr** clause must have a corresponding list item in the device data environment.
- A list item that specifies a given variable may not appear in more than one **use_device_ptr** clause.
- A reference to a list item in a **use_device_addr** clause must be to the address of the list item.

Cross References

- *default-device-var*, see Section 2.5 on page 63.
- **if** Clause, see Section 2.15 on page 220.
- **map** clause, see Section 2.19.7.1 on page 315.
- **omp_get_num_devices** routine, see Section 3.2.36 on page 371.
- **ompt_callback_target_t**, see Section 4.5.2.26 on page 490.

2.12.3 target enter data Construct

Summary

The **target enter data** directive specifies that variables are mapped to a device data environment. The **target enter data** directive is a stand-alone directive.

Syntax

C / C++

The syntax of the **target enter data** construct is as follows:

```
#pragma omp target enter data [ clause[ [, ] clause]... ] new-line
```

where *clause* is one of the following:

```
if([ target enter data : ] scalar-expression)
device(integer-expression)
map([map-type-modifier[, ] [map-type-modifier[, ] ...] map-type: locator-list)
depend([depend-modifier, ] dependence-type : locator-list)
nowait
```

C / C++

Fortran

The syntax of the **target enter data** is as follows:

```
!$omp target enter data [ clause[ [, ] clause]... ]
```

where *clause* is one of the following:

```
if([ target enter data : ] scalar-logical-expression)
device(scalar-integer-expression)
map([map-type-modifier[, ] [map-type-modifier[, ] ...] map-type: locator-list)
depend([depend-modifier, ] dependence-type : locator-list)
nowait
```

Fortran

Binding

The binding task set for a **target enter data** region is the generating task, which is the *target task* generated by the **target enter data** construct. The **target enter data** region binds to the corresponding *target task* region.

Description

When a **target enter data** construct is encountered, the list items are mapped to the device data environment according to the **map** clause semantics.

The **target enter data** construct is a task generating construct. The generated task is a *target task*. The generated task region encloses the **target enter data** region.

All clauses are evaluated when the **target enter data** construct is encountered. The data environment of the *target task* is created according to the data-sharing attribute clauses on the **target enter data** construct, per-data environment ICVs, and any default data-sharing attribute rules that apply to the **target enter data** construct. A variable that is mapped in the **target enter data** construct has a default data-sharing attribute of shared in the data environment of the *target task*.

Assignment operations associated with mapping a variable (see Section 2.19.7.1 on page 315) occur when the *target task* executes.

If the **nowait** clause is present, execution of the *target task* may be deferred. If the **nowait** clause is not present, the *target task* is an included task.

If a **depend** clause is present, it is associated with the *target task*.

If no **device** clause is present, the default device is determined by the *default-device-var* ICV.

When an **if** clause is present and the **if** clause expression evaluates to *false*, the device is the host.

Execution Model Events

Events associated with a *target task* are the same as for the **task** construct defined in Section 2.10.1 on page 135.

The *target-enter-data-begin* event occurs when a thread enters a **target enter data** region.

The *target-enter-data-end* event occurs when a thread exits a **target enter data** region.

Tool Callbacks

Callbacks associated with events for *target tasks* are the same as for the **task** construct defined in Section 2.10.1 on page 135; (*flags & ompt_task_target*) always evaluates to *true* in the dispatched callback.

A thread dispatches a registered **ompt_callback_target** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_target_enter_data** as its *kind* argument for each occurrence of a *target-enter-data-begin* event in that thread in the context of the target task on the host. Similarly, a thread dispatches a registered **ompt_callback_target** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_target_enter_data** as its *kind* argument for each occurrence of a *target-enter-data-end* event in that thread in the

context of the target task on the host. These callbacks have type signature `ompt_callback_target_t`.

Restrictions

- A program must not depend on any ordering of the evaluations of the clauses of the **target enter data** directive, or on any side effects of the evaluations of the clauses.
- At least one **map** clause must appear on the directive.
- At most one **device** clause can appear on the directive. The **device** clause expression must evaluate to a non-negative integer value less than the value of `omp_get_num_devices()` or to the value of `omp_get_initial_device()`.
- At most one **if** clause can appear on the directive.
- A *map-type* must be specified in all **map** clauses and must be either **to** or **alloc**.
- At most one **nowait** clause can appear on the directive.

Cross References

- *default-device-var*, see Section 2.5.1 on page 64.
- **task**, see Section 2.10.1 on page 135.
- **task scheduling constraints**, see Section 2.10.6 on page 149.
- **target data**, see Section 2.12.2 on page 161.
- **target exit data**, see Section 2.12.4 on page 166.
- **if** Clause, see Section 2.15 on page 220.
- **map** clause, see Section 2.19.7.1 on page 315.
- `omp_get_num_devices` routine, see Section 3.2.36 on page 371.
- `ompt_callback_target_t`, see Section 4.5.2.26 on page 490.

2.12.4 target exit data Construct

Summary

The **target exit data** directive specifies that list items are unmapped from a device data environment. The **target exit data** directive is a stand-alone directive.

Syntax

C / C++

The syntax of the **target exit data** construct is as follows:

```
#pragma omp target exit data [ clause [ [, ] clause ] ... ] new-line
```

where *clause* is one of the following:

```
if([ target exit data : ] scalar-expression)  
device(integer-expression)  
map([map-type-modifier [, ] [map-type-modifier [, ] ...] map-type : locator-list)  
depend([depend-modifier [, ] dependence-type : locator-list)  
nowait
```

C / C++

Fortran

The syntax of the **target exit data** is as follows:

```
!$omp target exit data [ clause [ [, ] clause ] ... ]
```

where *clause* is one of the following:

```
if([ target exit data : ] scalar-logical-expression)  
device(scalar-integer-expression)  
map([map-type-modifier [, ] [map-type-modifier [, ] ...] map-type : locator-list)  
depend([depend-modifier [, ] dependence-type : locator-list)  
nowait
```

Fortran

Binding

The binding task set for a **target exit data** region is the generating task, which is the *target task* generated by the **target exit data** construct. The **target exit data** region binds to the corresponding *target task* region.

Description

When a **target exit data** construct is encountered, the list items in the **map** clauses are unmapped from the device data environment according to the **map** clause semantics.

The **target exit data** construct is a task generating construct. The generated task is a *target task*. The generated task region encloses the **target exit data** region.

All clauses are evaluated when the **target exit data** construct is encountered. The data environment of the *target task* is created according to the data-sharing attribute clauses on the **target exit data** construct, per-data environment ICVs, and any default data-sharing attribute rules that apply to the **target exit data** construct. A variable that is mapped in the **target exit data** construct has a default data-sharing attribute of shared in the data environment of the *target task*.

Assignment operations associated with mapping a variable (see Section 2.19.7.1 on page 315) occur when the *target task* executes.

If the **nowait** clause is present, execution of the *target task* may be deferred. If the **nowait** clause is not present, the *target task* is an included task.

If a **depend** clause is present, it is associated with the *target task*.

If no **device** clause is present, the default device is determined by the *default-device-var* ICV.

When an **if** clause is present and the **if** clause expression evaluates to *false*, the device is the host.

Execution Model Events

Events associated with a *target task* are the same as for the **task** construct defined in Section 2.10.1 on page 135.

The *target-exit-data-begin* event occurs when a thread enters a **target exit data** region.

The *target-exit-data-end* event occurs when a thread exits a **target exit data** region.

Tool Callbacks

Callbacks associated with events for *target tasks* are the same as for the **task** construct defined in Section 2.10.1 on page 135; (*flags & ompt_task_target*) always evaluates to *true* in the dispatched callback.

A thread dispatches a registered **ompt_callback_target** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_target_exit_data** as its *kind* argument for each occurrence of a *target-exit-data-begin* event in that thread in the context of the target task on the host. Similarly, a thread dispatches a registered **ompt_callback_target** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_target_exit_data** as its *kind* argument for each occurrence of a *target-exit-data-end* event in that thread in the context of the target task on the host. These callbacks have type signature **ompt_callback_target_t**.

Restrictions

- A program must not depend on any ordering of the evaluations of the clauses of the **target exit data** directive, or on any side effects of the evaluations of the clauses.
- At least one **map** clause must appear on the directive.
- At most one **device** clause can appear on the directive. The **device** clause expression must evaluate to a non-negative integer value less than the value of `omp_get_num_devices()` or to the value of `omp_get_initial_device()`.
- At most one **if** clause can appear on the directive.
- A *map-type* must be specified in all **map** clauses and must be either **from**, **release**, or **delete**.
- At most one **nowait** clause can appear on the directive.

Cross References

- *default-device-var*, see Section 2.5.1 on page 64.
- **task**, see Section 2.10.1 on page 135.
- **task scheduling constraints**, see Section 2.10.6 on page 149.
- **target data**, see Section 2.12.2 on page 161.
- **target enter data**, see Section 2.12.3 on page 164.
- **if** Clause, see Section 2.15 on page 220.
- **map** clause, see Section 2.19.7.1 on page 315.
- `omp_get_num_devices` routine, see Section 3.2.36 on page 371.
- `ompt_callback_target_t`, see Section 4.5.2.26 on page 490.

1 2.12.5 target Construct

2 Summary

3 Map variables to a device data environment and execute the construct on that device.

4 Syntax

C / C++

5 The syntax of the **target** construct is as follows:

```
6 #pragma omp target [clause [ , ] clause] ... ] new-line  
7 structured-block
```

8 where *clause* is one of the following:

```
9 if([ target :] scalar-expression)  
10 device([ device-modifier :] integer-expression)  
11 private(list)  
12 firstprivate(list)  
13 in_reduction(reduction-identifier : list)  
14 map([[map-type-modifier[ , ] [map-type-modifier[ , ] ...] map-type : ] locator-list)  
15 is_device_ptr(list)  
16 defaultmap(implicit-behavior[:variable-category])  
17 nowait  
18 depend([depend-modifier, ] dependence-type : locator-list)  
19 allocate( [[allocator :] list)  
20 uses_allocators(allocator[ (allocator-traits-array) ]  
21 [ , allocator[ (allocator-traits-array) ] ...])
```

22 and where *device-modifier* is one of the following:

```
23 ancestor  
24 device_num
```

25 and where *allocator* is an identifier of **omp_allocator_handle_t** type and
26 *allocator-traits-array* is an identifier of **const omp_alloctrail_t *** type.

C / C++

The syntax of the **target** construct is as follows:

```
!$omp target [clause [ , ] clause ... ]
           structured-block
!$omp end target
```

where *clause* is one of the following:

```
if([ target :] scalar-logical-expression)
device([ device-modifier :] scalar-integer-expression)
private(list)
firstprivate(list)
in_reduction(reduction-identifier : list)
map([ [map-type-modifier [ , ] [map-type-modifier [ , ] ...] map-type : ] locator-list)
is_device_ptr(list)
defaultmap(implicit-behavior[:variable-category])
nowait
depend([depend-modifier , ] dependence-type : locator-list)
allocate([allocator :] list)
uses_allocators(allocator[ (allocator-traits-array) ]
                 [, allocator[ (allocator-traits-array) ] ...])
```

and where *device-modifier* is one of the following:

```
ancestor
device_num
```

and where *allocator* is an integer expression of **omp_allocator_handle_kind** *kind* and *allocator-traits-array* is an array of **type(omp_alloctrail)** type.

Binding

The binding task set for a **target** region is the generating task, which is the *target task* generated by the **target** construct. The **target** region binds to the corresponding *target task* region.

Description

The **target** construct provides a superset of the functionality provided by the **target data** directive, except for the **use_device_ptr** and **use_device_addr** clauses.

The functionality added to the **target** directive is the inclusion of an executable region to be executed by a device. That is, the **target** directive is an executable directive.

The **target** construct is a task generating construct. The generated task is a *target task*. The generated task region encloses the **target** region.

All clauses are evaluated when the **target** construct is encountered. The data environment of the *target task* is created according to the data-sharing attribute clauses on the **target** construct, per-data environment ICVs, and any default data-sharing attribute rules that apply to the **target** construct. If a variable or part of a variable is mapped by the **target** construct and does not appear as a list item in an **in_reduction** clause on the construct, the variable has a default data-sharing attribute of shared in the data environment of the *target task*.

Assignment operations associated with mapping a variable (see Section 2.19.7.1 on page 315) occur when the *target task* executes.

If a **device** clause in which the **device_num** *device-modifier* appears is present on the construct, the **device** clause expression specifies the device number of the target device. If *device-modifier* does not appear in the clause, the behavior of the clause is as if *device-modifier* is **device_num**.

If a **device** clause in which the **ancestor** *device-modifier* appears is present on the **target** construct and the **device** clause expression evaluates to 1, execution of the **target** region occurs on the parent device of the enclosing **target** region. If the **target** construct is not encountered in a **target** region, the current device is treated as the parent device. The encountering thread waits for completion of the **target** region on the parent device before resuming. For any list item that appears in a **map** clause on the same construct, if the corresponding list item exists in the device data environment of the parent device, it is treated as if it has a reference count of positive infinity.

If the **nowait** clause is present, execution of the *target task* may be deferred. If the **nowait** clause is not present, the *target task* is an included task.

If a **depend** clause is present, it is associated with the *target task*.

When an **if** clause is present and the **if** clause expression evaluates to *false*, the **target** region is executed by the host device in the host data environment.

The **is_device_ptr** clause is used to indicate that a list item is a device pointer already in the device data environment and that it should be used directly. Support for device pointers created outside of OpenMP, specifically outside of the **omp_target_alloc** routine and the **use_device_ptr** clause, is implementation defined.

If a function (C, C++, Fortran) or subroutine (Fortran) is referenced in a **target** construct then that function or subroutine is treated as if its name had appeared in a **to** clause on a **declare target** directive.

Each memory *allocator* specified in the **uses_allocators** clause will be made available in the **target** region. For each non-predefined allocator that is specified, a new allocator handle will be associated with an allocator that is created with the specified *traits* as if by a call to **omp_init_allocator** at the beginning of the **target** region. Each non-predefined allocator will be destroyed as if by a call to **omp_destroy_allocator** at the end of the **target** region.

C / C++

If a list item in a **map** clause has a base pointer and it is a scalar variable with a predetermined data-sharing attribute of *firstprivate* (see Section 2.19.1.1 on page 270), then on entry to the **target** region:

- If the list item is not a zero-length array section, the corresponding private variable is initialized such that the corresponding list item in the device data environment can be accessed through the pointer in the **target** region.
- If the list item is a zero-length array section, the corresponding private variable is initialized such that the corresponding storage location of the array section can be referenced through the pointer in the **target** region. If the corresponding storage location is not present in the device data environment, the corresponding private variable is initialized to NULL.

C / C++

Execution Model Events

Events associated with a *target task* are the same as for the **task** construct defined in Section 2.10.1 on page 135.

Events associated with the *initial task* that executes the **target** region are defined in Section 2.10.5 on page 148.

The *target-begin* event occurs when a thread enters a **target** region.

The *target-end* event occurs when a thread exits a **target** region.

The *target-submit* event occurs prior to creating an initial task on a target device for a **target** region.

Tool Callbacks

Callbacks associated with events for *target tasks* are the same as for the **task** construct defined in Section 2.10.1 on page 135; (*flags & omp_target*) always evaluates to *true* in the dispatched callback.

A thread dispatches a registered **omp_callback_target** callback with **omp_scope_begin** as its *endpoint* argument and **omp_target** as its *kind* argument for each occurrence of a *target-begin* event in that thread in the context of the target task on the host. Similarly, a thread dispatches a registered **omp_callback_target** callback with **omp_scope_end** as its *endpoint* argument and **omp_target** as its *kind* argument for each occurrence of a *target-end* event in that thread in the context of the target task on the host. These callbacks have type signature **omp_callback_target_t**.

A thread dispatches a registered **omp_callback_target_submit** callback for each occurrence of a *target-submit* event in that thread. The callback has type signature **omp_callback_target_submit_t**.

Restrictions

- If a **target update**, **target data**, **target enter data**, or **target exit data** construct is encountered during execution of a **target** region, the behavior is unspecified.
- The result of an **omp_set_default_device**, **omp_get_default_device**, or **omp_get_num_devices** routine called within a **target** region is unspecified.
- The effect of an access to a **threadprivate** variable in a target region is unspecified.
- If a list item in a **map** clause is a structure element, any other element of that structure that is referenced in the **target** construct must also appear as a list item in a **map** clause.
- A variable referenced in a **target** region but not the **target** construct that is not declared in the **target** region must appear in a **declare target** directive.
- At most one **defaultmap** clause for each category can appear on the directive.
- At most one **nowait** clause can appear on the directive.
- A *map-type* in a **map** clause must be **to**, **from**, **tofrom** or **alloc**.
- A list item that appears in an **is_device_ptr** clause must be a valid device pointer in the device data environment.
- At most one **device** clause can appear on the directive. The **device** clause expression must evaluate to a non-negative integer value less than the value of **omp_get_num_devices()** or to the value of **omp_get_initial_device()**.
- If a **device** clause in which the *ancestor device-modifier* appears is present on the construct, then the following restrictions apply:

- 1 – A **requires** directive with the **reverse_offload** clause must be specified;
- 2 – The **device** clause expression must evaluate to 1;
- 3 – Only the **device**, **firstprivate**, **private**, **defaultmap**, and **map** clauses may
- 4 appear on the construct;
- 5 – No OpenMP constructs or calls to OpenMP API runtime routines are allowed inside the
- 6 corresponding **target** region.
- 7 • Memory allocators that do not appear in a **uses_allocators** clause cannot appear as an
- 8 allocator in an **allocate** clause or be used in the **target** region unless a **requires**
- 9 directive with the **dynamic_allocators** clause is present in the same compilation unit.
- 10 • Memory allocators that appear in a **uses_allocators** clause cannot appear in other
- 11 data-sharing attribute clauses or data-mapping attribute clauses in the same construct.
- 12 • Predefined allocators appearing in a **uses_allocators** clause cannot have *traits* specified.
- 13 • Non-predefined allocators appearing in a **uses_allocators** clause must have *traits* specified.
- 14 • Arrays that contain allocator traits that appear in a **uses_allocators** clause must be
- 15 constant arrays, have constant values and be defined in the same scope as the construct in which
- 16 the clause appears.
- 17 • Any IEEE floating-point exception status flag, halting mode, or rounding mode set prior to a
- 18 **target** region is unspecified in the region.
- 19 • Any IEEE floating-point exception status flag, halting mode, or rounding mode set in a **target**
- 20 region is unspecified upon exiting the region.
- 21

C / C++

- 21 • An attached pointer must not be modified in a **target** region.
- 22

C

- 22 • A list item that appears in an **is_device_ptr** clause must have a type of pointer or array.
- 23

C++

- 23 • A list item that appears in an **is_device_ptr** clause must have a type of pointer, array,
- 24 reference to pointer or reference to array.
- 25 • The effect of invoking a virtual member function of an object on a device other than the device
- 26 on which the object was constructed is implementation defined.
- 27 • A **throw** executed inside a **target** region must cause execution to resume within the same
- 28 **target** region, and the same thread that threw the exception must catch it.
- 28

C++

Fortran

- An attached pointer that is associated with a given pointer target must not become associated with a different pointer target in a **target** region.
- A list item that appears in an **is_device_ptr** clause must be a dummy argument that does not have the **ALLOCATABLE**, **POINTER** or **VALUE** attribute.
- If a list item in a **map** clause is an array section, and the array section is derived from a variable with a **POINTER** or **ALLOCATABLE** attribute then the behavior is unspecified if the corresponding list item's variable is modified in the region.

Fortran

Cross References

- *default-device-var*, see Section 2.5 on page 63.
- **task** construct, see Section 2.10.1 on page 135.
- **task** scheduling constraints, see Section 2.10.6 on page 149
- Memory allocators, see Section 2.11.2 on page 152.
- **target data** construct, see Section 2.12.2 on page 161.
- **if** Clause, see Section 2.15 on page 220.
- **private** and **firstprivate** clauses, see Section 2.19.4 on page 282.
- Data-Mapping Attribute Rules and Clauses, see Section 2.19.7 on page 314.
- **omp_get_num_devices** routine, see Section 3.2.36 on page 371.
- **omp_alloctrail_t** and **omp_alloctrail** types, see Section 3.7.1 on page 406.
- **omp_set_default_allocator** routine, see Section 3.7.4 on page 411.
- **omp_get_default_allocator** routine, see Section 3.7.5 on page 412.
- **ompt_callback_target_t**, see Section 4.5.2.26 on page 490.
- **ompt_callback_target_submit_t**, Section 4.5.2.28 on page 494.

2.12.6 target update Construct

Summary

The **target update** directive makes the corresponding list items in the device data environment consistent with their original list items, according to the specified motion clauses. The **target update** construct is a stand-alone directive.

Syntax

C / C++

The syntax of the **target update** construct is as follows:

```
#pragma omp target update clause[ [ [, ] clause] ... ] new-line
```

where *clause* is either *motion-clause* or one of the following:

```
if([ target update :] scalar-expression)  
device(integer-expression)  
nowait  
depend([depend-modifier, ] dependence-type : locator-list)
```

and *motion-clause* is one of the following:

```
to ( [ mapper (mapper-identifier) : ] locator-list )  
from ( [ mapper (mapper-identifier) : ] locator-list )
```

C / C++

Fortran

The syntax of the **target update** construct is as follows:

```
!$omp target update clause[ [ [, ] clause] ... ]
```

where *clause* is either *motion-clause* or one of the following:

```
if([target update :] scalar-logical-expression)  
device(scalar-integer-expression)  
nowait  
depend([depend-modifier, ] dependence-type : locator-list)
```

and *motion-clause* is one of the following:

```
to ( [ mapper (mapper-identifier) : ] locator-list )  
from ( [ mapper (mapper-identifier) : ] locator-list )
```

Fortran

Binding

The binding task set for a **target update** region is the generating task, which is the *target task* generated by the **target update** construct. The **target update** region binds to the corresponding *target task* region.

Description

For each list item in a **to** or **from** clause there is a corresponding list item and an original list item. If the corresponding list item is not present in the device data environment then no assignment occurs to or from the original list item. Otherwise, each corresponding list item in the device data environment has an original list item in the current task's data environment. If a **mapper()** modifier appears in a **to** clause, each list item is replaced with the list items that the given mapper specifies are to be mapped with a **to** or **tofrom** map-type. If a **mapper()** modifier appears in a **from** clause, each list item is replaced with the list items that the given mapper specifies are to be mapped with a **from** or **tofrom** map-type.

For each list item in a **from** or a **to** clause:

- For each part of the list item that is an attached pointer:

C / C++

- On exit from the region that part of the original list item will have the value it had on entry to the region;
- On exit from the region that part of the corresponding list item will have the value it had on entry to the region;

C / C++

Fortran

- On exit from the region that part of the original list item, if associated, will be associated with the same pointer target with which it was associated on entry to the region;
- On exit from the region that part of the corresponding list item, if associated, will be associated with the same pointer target with which it was associated on entry to the region.

Fortran

- For each part of the list item that is not an attached pointer:

- If the clause is **from**, the value of that part of the corresponding list item is assigned to that part of the original list item;
- If the clause is **to**, the value of that part of the original list item is assigned to that part of the corresponding list item.

- To avoid data races:

- Concurrent reads or updates of any part of the original list item must be synchronized with the update of the original list item that occurs as a result of the **from** clause;
- Concurrent reads or updates of any part of the corresponding list item must be synchronized with the update of the corresponding list item that occurs as a result of the **to** clause.

C / C++

The list items that appear in the **to** or **from** clauses may use shape-operators.

C / C++

The list items that appear in the **to** or **from** clauses may include array sections with *stride* expressions.

The **target update** construct is a task generating construct. The generated task is a *target task*. The generated task region encloses the **target update** region.

All clauses are evaluated when the **target update** construct is encountered. The data environment of the *target task* is created according to the data-sharing attribute clauses on the **target update** construct, per-data environment ICVs, and any default data-sharing attribute rules that apply to the **target update** construct. A variable that is mapped in the **target update** construct has a default data-sharing attribute of **shared** in the data environment of the *target task*.

Assignment operations associated with mapping a variable (see Section 2.19.7.1 on page 315) occur when the *target task* executes.

If the **nowait** clause is present, execution of the *target task* may be deferred. If the **nowait** clause is not present, the *target task* is an included task.

If a **depend** clause is present, it is associated with the *target task*.

The device is specified in the **device** clause. If there is no **device** clause, the device is determined by the *default-device-var* ICV. When an **if** clause is present and the **if** clause expression evaluates to *false* then no assignments occur.

Execution Model Events

Events associated with a *target task* are the same as for the **task** construct defined in Section 2.10.1 on page 135.

The *target-update-begin* event occurs when a thread enters a **target update** region.

The *target-update-end* event occurs when a thread exits a **target update** region.

Tool Callbacks

Callbacks associated with events for *target tasks* are the same as for the **task** construct defined in Section 2.10.1 on page 135; (*flags* & **ompt_task_target**) always evaluates to *true* in the dispatched callback.

A thread dispatches a registered **ompt_callback_target** callback with **ompt_scope_begin** as its *endpoint* argument and **ompt_target_update** as its *kind* argument for each occurrence of a *target-update-begin* event in that thread in the context of the target task on the host. Similarly, a thread dispatches a registered **ompt_callback_target** callback with **ompt_scope_end** as its *endpoint* argument and **ompt_target_update** as its *kind* argument for each occurrence of a *target-update-end* event in that thread in the context of the target task on the host. These callbacks have type signature **ompt_callback_target_t**.

Restrictions

- A program must not depend on any ordering of the evaluations of the clauses of the **target update** directive, or on any side effects of the evaluations of the clauses.
- At least one *motion-clause* must be specified.
- A list item can only appear in a **to** or **from** clause, but not both.
- A list item in a **to** or **from** clause must have a mappable type.
- At most one **device** clause can appear on the directive. The **device** clause expression must evaluate to a non-negative integer value less than the value of **omp_get_num_devices()** or to the value of **omp_get_initial_device()**.
- At most one **if** clause can appear on the directive.
- At most one **nowait** clause can appear on the directive.

Cross References

- Array shaping, Section [2.1.4](#) on page [43](#)
- Array sections, Section [2.1.5](#) on page [44](#)
- *default-device-var*, see Section [2.5](#) on page [63](#).
- **task** construct, see Section [2.10.1](#) on page [135](#).
- **task** scheduling constraints, see Section [2.10.6](#) on page [149](#)
- **target data**, see Section [2.12.2](#) on page [161](#).
- **if** Clause, see Section [2.15](#) on page [220](#).
- **omp_get_num_devices** routine, see Section [3.2.36](#) on page [371](#).
- **ompt_callback_task_create_t**, see Section [4.5.2.7](#) on page [467](#).
- **ompt_callback_target_t**, see Section [4.5.2.26](#) on page [490](#).

2.12.7 declare target Directive

Summary

The **declare target** directive specifies that variables, functions (C, C++ and Fortran), and subroutines (Fortran) are mapped to a device. The **declare target** directive is a declarative directive.

Syntax

C / C++

The syntax of the **declare target** directive takes either of the following forms:

```
#pragma omp declare target new-line  
declaration-definition-seq  
#pragma omp end declare target new-line
```

or

```
#pragma omp declare target (extended-list) new-line
```

or

```
#pragma omp declare target clause[ [, ] clause ... ] new-line
```

where *clause* is one of the following:

```
to(extended-list)  
link(list)  
device_type(host | nohost | any)
```

C / C++

Fortran

The syntax of the **declare target** directive is as follows:

```
!$omp declare target (extended-list)
```

or

```
!$omp declare target [clause[ [, ] clause] ... ]
```

where *clause* is one of the following:

```
to(extended-list)  
link(list)  
device_type(host | nohost | any)
```

Fortran

Description

The **declare target** directive ensures that procedures and global variables can be executed or accessed on a device. Variables are mapped for all device executions, or for specific device executions through a **link** clause.

If an *extended-list* is present with no clause then the **to** clause is assumed.

The **device_type** clause specifies if a version of the procedure should be made available on host, device or both. If **host** is specified only a host version of the procedure is made available. If **nohost** is specified then only a device version of the procedure is made available. If **any** is specified then both device and host versions of the procedure are made available.

C / C++

If a function appears in a **to** clause in the same translation unit in which the definition of the function occurs then a device-specific version of the function is created.

If a variable appears in a **to** clause in the same translation unit in which the definition of the variable occurs then the original list item is allocated a corresponding list item in the device data environment of all devices.

C / C++

Fortran

If an internal procedure appears in a **to** clause then a device-specific version of the procedure is created.

If a variable that is host associated appears in a **to** clause then the original list item is allocated a corresponding list item in the device data environment of all devices.

Fortran

If a variable appears in a **to** clause then the corresponding list item in the device data environment of each device is initialized once, in the manner specified by the program, but at an unspecified point in the program prior to the first reference to that list item. The list item is never removed from those device data environments as if its reference count is initialized to positive infinity.

Including list items in a **link** clause supports compilation of functions called in a **target** region that refer to the list items. The list items are not mapped by the **declare target** directive. Instead, they are mapped according to the data mapping rules described in Section 2.19.7 on page 314.

C / C++

If a function is referenced in a function that appears as a list item in a **to** clause on a **declare target** directive then the name of the referenced function is treated as if it had appeared in a **to** clause on a **declare target** directive.

If a variable with static storage duration or a function (except *lambda* for C++) is referenced in the initializer expression list of a variable with static storage duration that appears as a list item in a **to** clause on a **declare target** directive then the name of the referenced variable or function is treated as if it had appeared in a **to** clause on a **declare target** directive.

The form of the **declare target** directive that has no clauses and requires a matching **end declare target** directive defines an implicit *extended-list* to an implicit **to** clause. The implicit *extended-list* consists of the variable names of any variable declarations at file or namespace scope that appear between the two directives and of the function names of any function declarations at file, namespace or class scope that appear between the two directives.

The *declaration-definition-seq* defined by a **declare target** directive and an **end declare target** directive may contain **declare target** directives. If a **device_type** clause is present on the contained **declare target** directive, then its argument determines which versions are made available. If a list item appears both in an implicit and explicit list, the explicit list determines which versions are made available.

C / C++

Fortran

If a procedure is referenced in a procedure that appears as a list item in a **to** clause on a **declare target** directive then the name of the procedure is treated as if it had appeared in a **to** clause on a **declare target** directive.

If a **declare target** does not have any clauses then an implicit *extended-list* to an implicit **to** clause of one item is formed from the name of the enclosing subroutine subprogram, function subprogram or interface body to which it applies.

If a **declare target** directive has a **device_type** clause then any enclosed internal procedures cannot contain any **declare target** directives. The enclosing **device_type** clause implicitly applies to internal procedures.

Fortran

Restrictions

- A threadprivate variable cannot appear in a **declare target** directive.
- A variable declared in a **declare target** directive must have a mappable type.
- The same list item must not appear multiple times in clauses on the same directive.
- The same list item must not explicitly appear in both a **to** clause on one **declare target** directive and a **link** clause on another **declare target** directive.

C++

- The function names of overloaded functions or template functions may only be specified within an implicit *extended-list*.
- If a *lambda declaration and definition* appears between a **declare target** directive and the matching **end declare target** directive, all variables that are captured by the *lambda* expression must also appear in a **to** clause.

C++

Fortran

- If a list item is a procedure name, it must not be a generic name, procedure pointer or entry name.
- Any **declare target** directive with clauses must appear in a specification part of a subroutine subprogram, function subprogram, program or module.
- Any **declare target** directive without clauses must appear in a specification part of a subroutine subprogram, function subprogram or interface body to which it applies.
- If a **declare target** directive is specified in an interface block for a procedure, it must match a **declare target** directive in the definition of the procedure.
- If an external procedure is a type-bound procedure of a derived type and a **declare target** directive is specified in the definition of the external procedure, such a directive must appear in the interface block that is accessible to the derived type definition.
- If any procedure is declared via a procedure declaration statement that is not in the type-bound procedure part of a derived-type definition, any **declare target** with the procedure name must appear in the same specification part.
- A variable that is part of another variable (as an array, structure element or type parameter inquiry) cannot appear in a **declare target** directive.
- The **declare target** directive must appear in the declaration section of a scoping unit in which the common block or variable is declared.
- If a **declare target** directive that specifies a common block name appears in one program unit, then such a directive must also appear in every other program unit that contains a **COMMON** statement that specifies the same name, after the last such **COMMON** statement in the program unit.
- If a list item is declared with the **BIND** attribute, the corresponding C entities must also be specified in a **declare target** directive in the C program.
- A blank common block cannot appear in a **declare target** directive.
- A variable can only appear in a **declare target** directive in the scope in which it is declared. It must not be an element of a common block or appear in an **EQUIVALENCE** statement.
- A variable that appears in a **declare target** directive must be declared in the Fortran scope of a module or have the **SAVE** attribute, either explicitly or implicitly.

Fortran

Cross References

- **target data** construct, see Section 2.12.2 on page 161.
- **target** construct, see Section 2.12.5 on page 170.

2.13 Combined Constructs

Combined constructs are shortcuts for specifying one construct immediately nested inside another construct. The semantics of the combined constructs are identical to that of explicitly specifying the first construct containing one instance of the second construct and no other statements.

For combined constructs, tool callbacks are invoked as if the constructs were explicitly nested.

2.13.1 Parallel Worksharing-Loop Construct

Summary

The parallel worksharing-loop construct is a shortcut for specifying a **parallel** construct containing a worksharing-loop construct with one or more associated loops and no other statements.

Syntax

C / C++

The syntax of the parallel worksharing-loop construct is as follows:

```
#pragma omp parallel for [clause[ [, ] clause] ... ] new-line  
    for-loops
```

where *clause* can be any of the clauses accepted by the **parallel** or **for** directives, except the **nowait** clause, with identical meanings and restrictions.

C / C++

The syntax of the parallel worksharing-loop construct is as follows:

```
!$omp parallel do [clause[ [, ] clause] ... ]
    do-loops
/$omp end parallel do
```

where *clause* can be any of the clauses accepted by the **parallel** or **do** directives, with identical meanings and restrictions.

If an **end parallel do** directive is not specified, an **end parallel do** directive is assumed at the end of the *do-loops*. **nowait** may not be specified on an **end parallel do** directive.

Description

The semantics are identical to explicitly specifying a **parallel** directive immediately followed by a worksharing-loop directive.

Restrictions

- The restrictions for the **parallel** construct and the worksharing-loop construct apply.

Cross References

- **parallel** construct, see Section 2.6 on page 74.
- Worksharing-loop construct, see Section 2.9.2 on page 101.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.13.2 parallel loop Construct

Summary

The **parallel loop** construct is a shortcut for specifying a **parallel** construct containing a **loop** construct with one or more associated loops and no other statements.

Syntax

C / C++

The syntax of the **parallel loop** construct is as follows:

```
#pragma omp parallel loop [clause[ [, ] clause] ... ] new-line  
    for-loops
```

where *clause* can be any of the clauses accepted by the **parallel** or **loop** directives, with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **parallel loop** construct is as follows:

```
!$omp parallel loop [clause[ [, ] clause] ... ]  
    do-loops  
[!$omp end parallel loop]
```

where *clause* can be any of the clauses accepted by the **parallel** or **loop** directives, with identical meanings and restrictions.

If an **end parallel loop** directive is not specified, an **end parallel loop** directive is assumed at the end of the *do-loops*. **nowait** may not be specified on an **end parallel loop** directive.

Fortran

Description

The semantics are identical to explicitly specifying a **parallel** directive immediately followed by a **loop** directive.

Restrictions

- The restrictions for the **parallel** construct and the **loop** construct apply.

Cross References

- **parallel** construct, see Section 2.6 on page 74.
- **loop** construct, see Section 2.9.5 on page 128.
- Data attribute clauses, see Section 2.19.4 on page 282.

1 2.13.3 parallel sections Construct

2 Summary

3 The **parallel sections** construct is a shortcut for specifying a **parallel** construct
4 containing a **sections** construct and no other statements.

5 Syntax

C / C++

6 The syntax of the **parallel sections** construct is as follows:

```
7 #pragma omp parallel sections [clause[ [, ] clause] ... ] new-line  
8 {  
9     [#pragma omp section new-line]  
10     structured-block  
11     [#pragma omp section new-line]  
12     structured-block]  
13     ...  
14 }
```

15 where *clause* can be any of the clauses accepted by the **parallel** or **sections** directives,
16 except the **nowait** clause, with identical meanings and restrictions.

C / C++

Fortran

17 The syntax of the **parallel sections** construct is as follows:

```
18 !$omp parallel sections [clause[ [, ] clause] ... ]  
19     [!$omp section]  
20     structured-block  
21     [!$omp section  
22     structured-block]  
23     ...  
24 !$omp end parallel sections
```

25 where *clause* can be any of the clauses accepted by the **parallel** or **sections** directives, with
26 identical meanings and restrictions.

27 The last section ends at the **end parallel sections** directive. **nowait** cannot be specified
28 on an **end parallel sections** directive.

Fortran

1 **Description**

2 C / C++

3 The semantics are identical to explicitly specifying a **parallel** directive immediately followed by a **sections** directive.

4 C / C++

5 Fortran

6 The semantics are identical to explicitly specifying a **parallel** directive immediately followed by a **sections** directive, and an **end sections** directive immediately followed by an **end parallel** directive.

7 Fortran

7 **Restrictions**

8 The restrictions for the **parallel** construct and the **sections** construct apply.

9 **Cross References**

- 10 • **parallel** construct, see Section 2.6 on page 74.
- 11 • **sections** construct, see Section 2.8.1 on page 86.
- 12 • Data attribute clauses, see Section 2.19.4 on page 282.

13 Fortran

13 **2.13.4 parallel workshare Construct**

14 **Summary**

15 The **parallel workshare** construct is a shortcut for specifying a **parallel** construct containing a **workshare** construct and no other statements.

17 **Syntax**

18 The syntax of the **parallel workshare** construct is as follows:

```
19       !$omp parallel workshare [clause[ [, ] clause] ... ]  
20               structured-block  
21       !$omp end parallel workshare
```

22 where *clause* can be any of the clauses accepted by the **parallel** directive, with identical meanings and restrictions. **nowait** may not be specified on an **end parallel workshare** directive.

Description

The semantics are identical to explicitly specifying a **parallel** directive immediately followed by a **workshare** directive, and an **end workshare** directive immediately followed by an **end parallel** directive.

Restrictions

The restrictions for the **parallel** construct and the **workshare** construct apply.

Cross References

- **parallel** construct, see Section 2.6 on page 74.
- **workshare** construct, see Section 2.8.3 on page 92.
- Data attribute clauses, see Section 2.19.4 on page 282.

Fortran

2.13.5 Parallel Worksharing-Loop SIMD Construct

Summary

The parallel worksharing-loop SIMD construct is a shortcut for specifying a **parallel** construct containing a worksharing-loop SIMD construct and no other statements.

Syntax

C / C++

The syntax of the parallel worksharing-loop SIMD construct is as follows:

```
#pragma omp parallel for simd [clause[ [, ] clause] ... ] new-line  
    for-loops
```

where *clause* can be any of the clauses accepted by the **parallel** or **for simd** directives, except the **nowait** clause, with identical meanings and restrictions.

C / C++

Fortran

The syntax of the parallel worksharing-loop SIMD construct is as follows:

```
!$omp parallel do simd [clause[ [, ] clause] ... ]  
    do-loops  
/$omp end parallel do simd/
```

where *clause* can be any of the clauses accepted by the **parallel** or **do simd** directives, with identical meanings and restrictions.

If an **end parallel do simd** directive is not specified, an **end parallel do simd** directive is assumed at the end of the *do-loops*. **nowait** may not be specified on an **end parallel do simd** directive.

Fortran

Description

The semantics of the parallel worksharing-loop SIMD construct are identical to explicitly specifying a **parallel** directive immediately followed by a worksharing-loop SIMD directive.

Restrictions

The restrictions for the **parallel** construct and the worksharing-loop SIMD construct apply.

Cross References

- **parallel** construct, see Section 2.6 on page 74.
- Worksharing-loop SIMD construct, see Section 2.9.3.2 on page 114.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.13.6 parallel master Construct

Summary

The **parallel master** construct is a shortcut for specifying a **parallel** construct containing a **master** construct and no other statements.

Syntax

C / C++

The syntax of the **parallel master** construct is as follows:

```
#pragma omp parallel master [clause[ [, ] clause] ... ] new-line  
    structured-block
```

where *clause* can be any of the clauses accepted by the **parallel** or **master** directives, with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **parallel master** construct is as follows:

```
!$omp parallel master [clause[ [, ] clause] ... ]  
    structured-block  
!$omp end parallel master
```

where *clause* can be any of the clauses accepted by the **parallel** or **master** directives, with identical meanings and restrictions.

Fortran

Description

The semantics are identical to explicitly specifying a **parallel** directive immediately followed by a **master** directive.

Restrictions

The restrictions for the **parallel** construct and the **master** construct apply.

Cross References

- **parallel** construct, see Section 2.6 on page 74.
- **master** construct, see Section 2.16 on page 221.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.13.7 master taskloop Construct

Summary

The **master taskloop** construct is a shortcut for specifying a **master** construct containing a **taskloop** construct and no other statements.

Syntax

C / C++

The syntax of the **master taskloop** construct is as follows:

```
#pragma omp master taskloop [clause[ [, ] clause] ... ] new-line  
    for-loops
```

where *clause* can be any of the clauses accepted by the **master** or **taskloop** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **master taskloop** construct is as follows:

```
!$omp master taskloop [clause[ [, ] clause] ... ]  
    do-loops  
[!$omp end master taskloop]
```

where *clause* can be any of the clauses accepted by the **master** or **taskloop** directives with identical meanings and restrictions.

If an **end master taskloop** directive is not specified, an **end master taskloop** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **master** directive immediately followed by a **taskloop** directive.

Restrictions

The restrictions for the **master** and **taskloop** constructs apply.

Cross References

- **taskloop** construct, see Section 2.10.2 on page 140.
- **master** construct, see Section 2.16 on page 221.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.13.8 master taskloop simd Construct

Summary

The **master taskloop simd** construct is a shortcut for specifying a **master** construct containing a **taskloop simd** construct and no other statements.

Syntax

C / C++

The syntax of the **master taskloop simd** construct is as follows:

```
#pragma omp master taskloop simd [clause[ [, ] clause] ... ] new-line  
for-loops
```

where *clause* can be any of the clauses accepted by the **master** or **taskloop simd** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **master taskloop simd** construct is as follows:

```
!$omp master taskloop simd [clause[ [, ] clause] ... ]  
do-loops  
[!$omp end master taskloop simd]
```

where *clause* can be any of the clauses accepted by the **master** or **taskloop simd** directives with identical meanings and restrictions.

If an **end master taskloop simd** directive is not specified, an **end master taskloop simd** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **master** directive immediately followed by a **taskloop simd** directive.

Restrictions

The restrictions for the **master** and **taskloop simd** constructs apply.

Cross References

- **taskloop simd** construct, see Section 2.10.3 on page 146.
- **master** construct, see Section 2.16 on page 221.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.13.9 parallel master taskloop Construct

Summary

The **parallel master taskloop** construct is a shortcut for specifying a **parallel** construct containing a **master taskloop** construct and no other statements.

Syntax

C / C++

The syntax of the **parallel master taskloop** construct is as follows:

```
#pragma omp parallel master taskloop [clause[ [, ] clause] ... ] new-line  
for-loops
```

where *clause* can be any of the clauses accepted by the **parallel** or **master taskloop** directives, except the **in_reduction** clause, with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **parallel master taskloop** construct is as follows:

```
!$omp parallel master taskloop [clause[ [, ] clause] ... ]  
do-loops  
[!$omp end parallel master taskloop]
```

where *clause* can be any of the clauses accepted by the **parallel** or **master taskloop** directives, except the **in_reduction** clause, with identical meanings and restrictions.

If an **end parallel master taskloop** directive is not specified, an **end parallel master taskloop** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **parallel** directive immediately followed by a **master taskloop** directive.

Restrictions

The restrictions for the **parallel** construct and the **master taskloop** construct apply.

Cross References

- **parallel** construct, see Section 2.6 on page 74.
- **master taskloop** construct, see Section 2.13.7 on page 192.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.13.10 parallel master taskloop simd Construct

Summary

The **parallel master taskloop simd** construct is a shortcut for specifying a **parallel** construct containing a **master taskloop simd** construct and no other statements.

Syntax

C / C++

The syntax of the **parallel master taskloop simd** construct is as follows:

```
#pragma omp parallel master taskloop simd [clause[ [, ] clause] ... ] new-line  
for-loops
```

where *clause* can be any of the clauses accepted by the **parallel** or **master taskloop simd** directives, except the **in_reduction** clause, with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **parallel master taskloop simd** construct is as follows:

```
!$omp parallel master taskloop simd [clause[ [, ] clause] ... ]  
do-loops  
[!$omp end parallel master taskloop simd]
```

where *clause* can be any of the clauses accepted by the **parallel** or **master taskloop simd** directives, except the **in_reduction** clause, with identical meanings and restrictions.

If an **end parallel master taskloop simd** directive is not specified, an **end parallel master taskloop simd** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **parallel** directive immediately followed by a **master taskloop simd** directive.

Restrictions

The restrictions for the **parallel** construct and the **master taskloop simd** construct apply.

Cross References

- **parallel** construct, see Section 2.6 on page 74.
- **master taskloop simd** construct, see Section 2.13.8 on page 194.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.13.11 teams distribute Construct

Summary

The **teams distribute** construct is a shortcut for specifying a **teams** construct containing a **distribute** construct and no other statements.

Syntax

C / C++

The syntax of the **teams distribute** construct is as follows:

```
#pragma omp teams distribute [clause[ [, ] clause] ... ] new-line  
    for-loops
```

where *clause* can be any of the clauses accepted by the **teams** or **distribute** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **teams distribute** construct is as follows:

```
!$omp teams distribute [clause[ [, ] clause] ... ]  
    do-loops  
!$omp end teams distribute
```

where *clause* can be any of the clauses accepted by the **teams** or **distribute** directives with identical meanings and restrictions.

If an **end teams distribute** directive is not specified, an **end teams distribute** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **teams** directive immediately followed by a **distribute** directive.

Restrictions

The restrictions for the **teams** and **distribute** constructs apply.

Cross References

- **teams** construct, see Section [2.7](#) on page [82](#).
- **distribute** construct, see Section [2.9.4.1](#) on page [120](#).
- Data attribute clauses, see Section [2.19.4](#) on page [282](#).

2.13.12 teams distribute simd Construct

Summary

The **teams distribute simd** construct is a shortcut for specifying a **teams** construct containing a **distribute simd** construct and no other statements.

Syntax

C / C++

The syntax of the **teams distribute simd** construct is as follows:

```
#pragma omp teams distribute simd [clause[ [, ] clause] ... ] new-line  
for-loops
```

where *clause* can be any of the clauses accepted by the **teams** or **distribute simd** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **teams distribute simd** construct is as follows:

```
!$omp teams distribute simd [clause[ [, ] clause] ... ]  
do-loops  
[!$omp end teams distribute simd]
```

where *clause* can be any of the clauses accepted by the **teams** or **distribute simd** directives with identical meanings and restrictions.

If an **end teams distribute simd** directive is not specified, an **end teams distribute simd** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **teams** directive immediately followed by a **distribute simd** directive.

Restrictions

The restrictions for the **teams** and **distribute simd** constructs apply.

Cross References

- **teams** construct, see Section 2.7 on page 82.
- **distribute simd** construct, see Section 2.9.4.2 on page 123.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.13.13 Teams Distribute Parallel Worksharing-Loop Construct

Summary

The teams distribute parallel worksharing-loop construct is a shortcut for specifying a **teams** construct containing a distribute parallel worksharing-loop construct and no other statements.

Syntax

C / C++

The syntax of the teams distribute parallel worksharing-loop construct is as follows:

```
#pragma omp teams distribute parallel for \
    [clause[ [, ] clause] ... ] new-line
    for-loops
```

where *clause* can be any of the clauses accepted by the **teams** or **distribute parallel for** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the teams distribute parallel worksharing-loop construct is as follows:

```
!$omp teams distribute parallel do [clause[ [, ] clause] ... ]
    do-loops
[ !$omp end teams distribute parallel do ]
```

where *clause* can be any of the clauses accepted by the **teams** or **distribute parallel do** directives with identical meanings and restrictions.

If an **end teams distribute parallel do** directive is not specified, an **end teams distribute parallel do** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **teams** directive immediately followed by a distribute parallel worksharing-loop directive.

Restrictions

The restrictions for the **teams** and distribute parallel worksharing-loop constructs apply.

Cross References

- **teams** construct, see Section 2.7 on page 82.
- Distribute parallel worksharing-loop construct, see Section 2.9.4.3 on page 125.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.13.14 Teams Distribute Parallel Worksharing-Loop SIMD Construct

Summary

The teams distribute parallel worksharing-loop SIMD construct is a shortcut for specifying a **teams** construct containing a distribute parallel worksharing-loop SIMD construct and no other statements.

Syntax

C / C++

The syntax of the teams distribute parallel worksharing-loop SIMD construct is as follows:

```
#pragma omp teams distribute parallel for simd \  
    [clause[ [, ] clause] ... ] new-line  
for-loops
```

where *clause* can be any of the clauses accepted by the **teams** or **distribute parallel for simd** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the teams distribute parallel worksharing-loop SIMD construct is as follows:

```
!$omp teams distribute parallel do simd [clause[ [, ] clause] ... ]  
do-loops  
[!$omp end teams distribute parallel do simd]
```

where *clause* can be any of the clauses accepted by the **teams** or **distribute parallel do simd** directives with identical meanings and restrictions.

If an **end teams distribute parallel do simd** directive is not specified, an **end teams distribute parallel do simd** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **teams** directive immediately followed by a distribute parallel worksharing-loop SIMD directive.

Restrictions

The restrictions for the **teams** and distribute parallel worksharing-loop SIMD constructs apply.

Cross References

- **teams** construct, see Section 2.7 on page 82.
- Distribute parallel worksharing-loop SIMD construct, see Section 2.9.4.4 on page 126.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.13.15 teams loop Construct

Summary

The **teams loop** construct is a shortcut for specifying a **teams** construct containing a **loop** construct and no other statements.

Syntax

C / C++

The syntax of the **teams loop** construct is as follows:

```
#pragma omp teams loop [clause[ [, ] clause] ... ] new-line  
for-loops
```

where *clause* can be any of the clauses accepted by the **teams** or **loop** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **teams loop** construct is as follows:

```
!$omp teams loop [clause[ [, ] clause] ... ]  
do-loops  
[!$omp end teams loop]
```

where *clause* can be any of the clauses accepted by the **teams** or **loop** directives with identical meanings and restrictions.

If an **end teams loop** directive is not specified, an **end teams loop** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **teams** directive immediately followed by a **loop** directive.

Restrictions

The restrictions for the **teams** and **loop** constructs apply.

Cross References

- **teams** construct, see Section 2.7 on page 82.
- **loop** construct, see Section 2.9.5 on page 128.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.13.16 target parallel Construct

Summary

The **target parallel** construct is a shortcut for specifying a **target** construct containing a **parallel** construct and no other statements.

Syntax

C / C++

The syntax of the **target parallel** construct is as follows:

```
#pragma omp target parallel [clause[ [, ] clause] ... ] new-line
    structured-block
```

where *clause* can be any of the clauses accepted by the **target** or **parallel** directives, except for **copyin**, with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **target parallel** construct is as follows:

```
!$omp target parallel [clause[ [, ] clause] ... ]
    structured-block
!$omp end target parallel
```

where *clause* can be any of the clauses accepted by the **target** or **parallel** directives, except for **copyin**, with identical meanings and restrictions.

Fortran

Description

The semantics are identical to explicitly specifying a **target** directive immediately followed by a **parallel** directive.

Restrictions

The restrictions for the **target** and **parallel** constructs apply except for the following explicit modifications:

- If any **if** clause on the directive includes a *directive-name-modifier* then all **if** clauses on the directive must include a *directive-name-modifier*.
- At most one **if** clause without a *directive-name-modifier* can appear on the directive.
- At most one **if** clause with the **parallel** *directive-name-modifier* can appear on the directive.
- At most one **if** clause with the **target** *directive-name-modifier* can appear on the directive.

Cross References

- **parallel** construct, see Section 2.6 on page 74.
- **target** construct, see Section 2.12.5 on page 170.
- **if** Clause, see Section 2.15 on page 220.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.13.17 Target Parallel Worksharing-Loop Construct

Summary

The target parallel worksharing-loop construct is a shortcut for specifying a **target** construct containing a parallel worksharing-loop construct and no other statements.

Syntax

C / C++

The syntax of the target parallel worksharing-loop construct is as follows:

```
#pragma omp target parallel for [clause[ [, ] clause] ... ] new-line  
for-loops
```

where *clause* can be any of the clauses accepted by the **target** or **parallel for** directives, except for **copyin**, with identical meanings and restrictions.

C / C++

Fortran

The syntax of the target parallel worksharing-loop construct is as follows:

```
!$omp target parallel do [clause[ [, ] clause] ... ]  
do-loops  
[!$omp end target parallel do]
```

where *clause* can be any of the clauses accepted by the **target** or **parallel do** directives, except for **copyin**, with identical meanings and restrictions.

If an **end target parallel do** directive is not specified, an **end target parallel do** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **target** directive immediately followed by a parallel worksharing-loop directive.

Restrictions

The restrictions for the **target** and parallel worksharing-loop constructs apply except for the following explicit modifications:

- If any **if** clause on the directive includes a *directive-name-modifier* then all **if** clauses on the directive must include a *directive-name-modifier*.
- At most one **if** clause without a *directive-name-modifier* can appear on the directive.
- At most one **if** clause with the **parallel** *directive-name-modifier* can appear on the directive.
- At most one **if** clause with the **target** *directive-name-modifier* can appear on the directive.

Cross References

- **target** construct, see Section 2.12.5 on page 170.
- Parallel Worksharing-Loop construct, see Section 2.13.1 on page 185.
- **if** Clause, see Section 2.15 on page 220.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.13.18 Target Parallel Worksharing-Loop SIMD Construct

Summary

The target parallel worksharing-loop SIMD construct is a shortcut for specifying a **target** construct containing a parallel worksharing-loop SIMD construct and no other statements.

Syntax

C / C++

The syntax of the target parallel worksharing-loop SIMD construct is as follows:

```
#pragma omp target parallel for simd \  
    [clause[ [, ] clause] ... ] new-line  
    for-loops
```

where *clause* can be any of the clauses accepted by the **target** or **parallel for simd** directives, except for **copyin**, with identical meanings and restrictions.

C / C++

Fortran

The syntax of the target parallel worksharing-loop SIMD construct is as follows:

```
!$omp target parallel do simd [clause[ [, ] clause] ... ]  
    do-loops  
[!$omp end target parallel do simd]
```

where *clause* can be any of the clauses accepted by the **target** or **parallel do simd** directives, except for **copyin**, with identical meanings and restrictions.

If an **end target parallel do simd** directive is not specified, an **end target parallel do simd** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **target** directive immediately followed by a parallel worksharing-loop SIMD directive.

Restrictions

The restrictions for the **target** and parallel worksharing-loop SIMD constructs apply except for the following explicit modifications:

- If any **if** clause on the directive includes a *directive-name-modifier* then all **if** clauses on the directive must include a *directive-name-modifier*.
- At most one **if** clause without a *directive-name-modifier* can appear on the directive.
- At most one **if** clause with the **parallel** *directive-name-modifier* can appear on the directive.
- At most one **if** clause with the **target** *directive-name-modifier* can appear on the directive.

Cross References

- **target** construct, see Section 2.12.5 on page 170.
- Parallel worksharing-loop SIMD construct, see Section 2.13.5 on page 190.
- **if** Clause, see Section 2.15 on page 220.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.13.19 target parallel loop Construct

Summary

The **target parallel loop** construct is a shortcut for specifying a **target** construct containing a **parallel loop** construct and no other statements.

Syntax

C / C++

The syntax of the **target parallel loop** construct is as follows:

```
#pragma omp target parallel loop [clause[ [, ] clause] ... ] new-line  
for-loops
```

where *clause* can be any of the clauses accepted by the **target** or **parallel loop** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **target parallel loop** construct is as follows:

```
!$omp target parallel loop [clause[ [, ] clause] ... ]  
do-loops  
/[!$omp end target parallel loop/
```

where *clause* can be any of the clauses accepted by the **teams** or **parallel loop** directives with identical meanings and restrictions.

If an **end target parallel loop** directive is not specified, an **end target parallel loop** directive is assumed at the end of the *do-loops*. **nowait** may not be specified on an **end target parallel loop** directive.

Fortran

Description

The semantics are identical to explicitly specifying a **target** directive immediately followed by a **parallel loop** directive.

Restrictions

The restrictions for the **target** and **parallel loop** constructs apply.

Cross References

- **target** construct, see Section 2.12.5 on page 170.
- **parallel loop** construct, see Section 2.13.2 on page 186.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.13.20 target simd Construct

Summary

The **target simd** construct is a shortcut for specifying a **target** construct containing a **simd** construct and no other statements.

Syntax

C / C++

The syntax of the **target simd** construct is as follows:

```
#pragma omp target simd [clause[ [, ] clause] ... ] new-line  
    for-loops
```

where *clause* can be any of the clauses accepted by the **target** or **simd** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **target simd** construct is as follows:

```
!$omp target simd [clause[ [, ] clause] ... ]  
    do-loops  
/$omp end target simd
```

where *clause* can be any of the clauses accepted by the **target** or **simd** directives with identical meanings and restrictions.

If an **end target simd** directive is not specified, an **end target simd** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **target** directive immediately followed by a **simd** directive.

Restrictions

The restrictions for the **target** and **simd** constructs apply.

Cross References

- **simd** construct, see Section [2.9.3.1](#) on page [110](#).
- **target** construct, see Section [2.12.5](#) on page [170](#).
- Data attribute clauses, see Section [2.19.4](#) on page [282](#).

2.13.21 target teams Construct

Summary

The **target teams** construct is a shortcut for specifying a **target** construct containing a **teams** construct and no other statements.

Syntax

C / C++

The syntax of the **target teams** construct is as follows:

```
#pragma omp target teams [clause[ [, ] clause] ... ] new-line  
    structured-block
```

where *clause* can be any of the clauses accepted by the **target** or **teams** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **target teams** construct is as follows:

```
!$omp target teams [clause[ [, ] clause] ... ]  
    structured-block  
!$omp end target teams
```

where *clause* can be any of the clauses accepted by the **target** or **teams** directives with identical meanings and restrictions.

Fortran

Description

The semantics are identical to explicitly specifying a **target** directive immediately followed by a **teams** directive.

Restrictions

The restrictions for the **target** and **teams** constructs apply.

Cross References

- **teams** construct, see Section 2.7 on page 82.
- **target** construct, see Section 2.12.5 on page 170.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.13.22 target teams distribute Construct

Summary

The **target teams distribute** construct is a shortcut for specifying a **target** construct containing a **teams distribute** construct and no other statements.

Syntax

C / C++

The syntax of the **target teams distribute** construct is as follows:

```
#pragma omp target teams distribute [clause[ [, ] clause] ... ] new-line  
for-loops
```

where *clause* can be any of the clauses accepted by the **target** or **teams distribute** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **target teams distribute** construct is as follows:

```
!$omp target teams distribute [clause[ [, ] clause] ... ]  
do-loops  
[!$omp end target teams distribute]
```

where *clause* can be any of the clauses accepted by the **target** or **teams distribute** directives with identical meanings and restrictions.

If an **end target teams distribute** directive is not specified, an **end target teams distribute** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **target** directive immediately followed by a **teams distribute** directive.

Restrictions

The restrictions for the **target** and **teams distribute** constructs.

Cross References

- **target** construct, see Section [2.12.2](#) on page [161](#).
- **teams distribute** construct, see Section [2.13.11](#) on page [197](#).
- Data attribute clauses, see Section [2.19.4](#) on page [282](#).

2.13.23 target teams distribute simd Construct

Summary

The **target teams distribute simd** construct is a shortcut for specifying a **target** construct containing a **teams distribute simd** construct and no other statements.

Syntax

C / C++

The syntax of the **target teams distribute simd** construct is as follows:

```
#pragma omp target teams distribute simd \  
    [clause[ [, ] clause] ... ] new-line  
    for-loops
```

where *clause* can be any of the clauses accepted by the **target** or **teams distribute simd** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **target teams distribute simd** construct is as follows:

```
!$omp target teams distribute simd [clause[ [, ] clause] ... ]  
    do-loops  
[!$omp end target teams distribute simd]
```

where *clause* can be any of the clauses accepted by the **target** or **teams distribute simd** directives with identical meanings and restrictions.

If an **end target teams distribute simd** directive is not specified, an **end target teams distribute simd** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **target** directive immediately followed by a **teams distribute simd** directive.

Restrictions

The restrictions for the **target** and **teams distribute simd** constructs apply.

Cross References

- **target** construct, see Section 2.12.2 on page 161.
- **teams distribute simd** construct, see Section 2.13.12 on page 198.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.13.24 target teams loop Construct

Summary

The **target teams loop** construct is a shortcut for specifying a **target** construct containing a **teams loop** construct and no other statements.

Syntax

C / C++

The syntax of the **target teams loop** construct is as follows:

```
#pragma omp target teams loop [clause[ [, ] clause] ... ] new-line  
    for-loops
```

where *clause* can be any of the clauses accepted by the **target** or **teams loop** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **target teams loop** construct is as follows:

```
!$omp target teams loop [clause[ [, ] clause] ... ]  
    do-loops  
/$omp end target teams loop/
```

where *clause* can be any of the clauses accepted by the **target** or **teams loop** directives with identical meanings and restrictions.

If an **end target teams loop** directive is not specified, an **end target teams loop** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **target** directive immediately followed by a **teams loop** directive.

Restrictions

The restrictions for the **target** and **teams loop** constructs.

Cross References

- **target** construct, see Section 2.12.5 on page 170.
- Teams loop construct, see Section 2.13.15 on page 202.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.13.25 Target Teams Distribute Parallel Worksharing-Loop Construct

Summary

The target teams distribute parallel worksharing-loop construct is a shortcut for specifying a **target** construct containing a teams distribute parallel worksharing-loop construct and no other statements.

Syntax

C / C++

The syntax of the target teams distribute parallel worksharing-loop construct is as follows:

```
#pragma omp target teams distribute parallel for \
                [clause[ [, ] clause] ... ] new-line
for-loops
```

where *clause* can be any of the clauses accepted by the **target** or **teams distribute parallel for** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the target teams distribute parallel worksharing-loop construct is as follows:

```
!$omp target teams distribute parallel do [clause[ [, ] clause] ... ]
do-loops
[!$omp end target teams distribute parallel do]
```

where *clause* can be any of the clauses accepted by the **target** or **teams distribute parallel do** directives with identical meanings and restrictions.

If an **end target teams distribute parallel do** directive is not specified, an **end target teams distribute parallel do** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **target** directive immediately followed by a teams distribute parallel worksharing-loop directive.

Restrictions

The restrictions for the **target** and teams distribute parallel worksharing-loop constructs apply except for the following explicit modifications:

- If any **if** clause on the directive includes a *directive-name-modifier* then all **if** clauses on the directive must include a *directive-name-modifier*.
- At most one **if** clause without a *directive-name-modifier* can appear on the directive.
- At most one **if** clause with the **parallel** *directive-name-modifier* can appear on the directive.
- At most one **if** clause with the **target** *directive-name-modifier* can appear on the directive.

Cross References

- **target** construct, see Section 2.12.5 on page 170.
- Teams distribute parallel worksharing-loop construct, see Section 2.13.13 on page 200.
- **if** Clause, see Section 2.15 on page 220.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.13.26 Target Teams Distribute Parallel Worksharing-Loop SIMD Construct

Summary

The target teams distribute parallel worksharing-loop SIMD construct is a shortcut for specifying a **target** construct containing a teams distribute parallel worksharing-loop SIMD construct and no other statements.

Syntax

C / C++

The syntax of the target teams distribute parallel worksharing-loop SIMD construct is as follows:

```
#pragma omp target teams distribute parallel for simd \  
[clause[ [, ] clause] ... ] new-line  
for-loops
```

where *clause* can be any of the clauses accepted by the **target** or **teams distribute parallel for simd** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the target teams distribute parallel worksharing-loop SIMD construct is as follows:

```
!$omp target teams distribute parallel do simd [clause[ [, ] clause] ... ]  
do-loops  
[!$omp end target teams distribute parallel do simd]
```

where *clause* can be any of the clauses accepted by the **target** or **teams distribute parallel do simd** directives with identical meanings and restrictions.

If an **end target teams distribute parallel do simd** directive is not specified, an **end target teams distribute parallel do simd** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **target** directive immediately followed by a **teams distribute parallel worksharing-loop SIMD** directive.

Restrictions

The restrictions for the **target** and **teams distribute parallel worksharing-loop SIMD** constructs apply except for the following explicit modifications:

- If any **if** clause on the directive includes a *directive-name-modifier* then all **if** clauses on the directive must include a *directive-name-modifier*.
- At most one **if** clause without a *directive-name-modifier* can appear on the directive.
- At most one **if** clause with the **parallel** *directive-name-modifier* can appear on the directive.
- At most one **if** clause with the **target** *directive-name-modifier* can appear on the directive.

Cross References

- **target** construct, see Section 2.12.5 on page 170.
- Teams distribute parallel worksharing-loop SIMD construct, see Section 2.13.14 on page 201.
- **if** Clause, see Section 2.15 on page 220.
- Data attribute clauses, see Section 2.19.4 on page 282.

2.14 Clauses on Combined and Composite Constructs

This section specifies the handling of clauses on combined or composite constructs and the handling of implicit clauses from variables with predetermined data sharing if they are not predetermined only on a particular construct. Some clauses are permitted only on a single construct of the constructs that constitute the combined or composite construct, in which case the effect is as if the clause is applied to that specific construct. As detailed in this section, other clauses have the effect as if they are applied to one or more constituent constructs.

The **collapse** clause is applied once to the combined or composite construct.

The effect of the **private** clause is as if it is applied only to the innermost constituent construct that permits it.

The effect of the **firstprivate** clause is as if it is applied to one or more constructs as follows:

- To the **distribute** construct if it is among the constituent constructs;
- To the **teams** construct if it is among the constituent constructs and the **distribute** construct is not;
- To the worksharing-loop construct if it is among the constituent constructs;
- To the **taskloop** construct if it is among the constituent constructs;
- To the **parallel** construct if it is among the constituent constructs and the worksharing-loop construct or the **taskloop** construct is not;
- To the outermost constituent construct if not already applied to it by the above rules and the outermost constituent construct is not a **teams** construct, a **parallel** construct, a **master** construct, or a **target** construct; and
- To the **target** construct if it is among the constituent constructs and the same list item does not appear in a **lastprivate** or **map** clause.

If the **parallel** construct is among the constituent constructs and the effect is not as if the **firstprivate** clause is applied to it by the above rules, then the effect is as if the **shared** clause with the same list item is applied to the **parallel** construct. If the **teams** construct is among the constituent constructs and the effect is not as if the **firstprivate** clause is applied to it by the above rules, then the effect is as if the **shared** clause with the same list item is applied to the **teams** construct.

The effect of the **lastprivate** clause is as if it is applied to one or more constructs as follows:

- To the worksharing-loop construct if it is among the constituent constructs;
- To the **taskloop** construct if it is among the constituent constructs;
- To the **distribute** construct if it is among the constituent constructs; and
- To the innermost constituent construct that permits it unless it is a worksharing-loop or **distribute** construct.

If the **parallel** construct is among the constituent constructs and the list item is not also specified in the **firstprivate** clause, then the effect of the **lastprivate** clause is as if the **shared** clause with the same list item is applied to the **parallel** construct. If the **teams** construct is among the constituent constructs and the list item is not also specified in the **firstprivate** clause, then the effect of the **lastprivate** clause is as if the **shared** clause with the same list item is applied to the **teams** construct. If the **target** construct is among the constituent constructs and the list item is not specified in a **map** clause, the effect of the **lastprivate** clause is as if the same list item appears in a **map** clause with a *map-type* of **tofrom**.

The effect of the **shared**, **default**, **order**, or **allocate** clause is as if it is applied to all constituent constructs that permit the clause.

The effect of the **reduction** clause is as if it is applied to all constructs that permit the clause, except for the following constructs:

- The **parallel** construct, when combined with the **sections**, worksharing-loop, **loop**, or **taskloop** construct; and
- The **teams** construct, when combined with the **loop** construct.

For the **parallel** and **teams** constructs above, the effect of the **reduction** clause instead is as if each list item or, for any list item that is an array item, its corresponding base array or base pointer appears in a **shared** clause for the construct. If the **task reduction-modifier** is specified, the effect is as if it only modifies the behavior of the **reduction** clause on the innermost construct that constitutes the combined construct and that accepts the modifier (see Section 2.19.5.4 on page 300). If the **inscan reduction-modifier** is specified, the effect is as if it modifies the behavior of the **reduction** clause on all constructs of the combined construct to which the clause is applied and that accept the modifier. If a construct to which the **inscan reduction-modifier** is applied is combined with the **target** construct, the effect is as if the same list item also appears in a **map** clause with a *map-type* of **tofrom**.

The **in_reduction** clause is permitted on a single construct among those that constitute the combined or composite construct and the effect is as if the clause is applied to that construct, but if that construct is a **target** construct, the effect is also as if the same list item appears in a **map** clause with a *map-type* of **tofrom** and a *map-type-modifier* of **always**.

The effect of the **if** clause is described in Section 2.15 on page 220.

The effect of the **linear** clause is as if it is applied to the innermost constituent construct. Additionally, if the list item is not the iteration variable of a **simd** or worksharing-loop SIMD construct, the effect on the outer constituent constructs is as if the list item was specified in **firstprivate** and **lastprivate** clauses on the combined or composite construct, with the rules specified above applied. If a list item of the **linear** clause is the iteration variable of a **simd** or worksharing-loop SIMD construct and it is not declared in the construct, the effect on the outer constituent constructs is as if the list item was specified in a **lastprivate** clause on the combined or composite construct with the rules specified above applied.

The effect of the **nowait** clause is as if it is applied to the outermost constituent construct that permits it.

If the clauses have expressions on them, such as for various clauses where the argument of the clause is an expression, or *lower-bound*, *length*, or *stride* expressions inside array sections (or *subscript* and *stride* expressions in *subscript-triplet* for Fortran), or *linear-step* or *alignment* expressions, the expressions are evaluated immediately before the construct to which the clause has been split or duplicated per the above rules (therefore inside of the outer constituent constructs). However, the expressions inside the **num_teams** and **thread_limit** clauses are always evaluated before the outermost constituent construct.

The restriction that a list item may not appear in more than one data sharing clause with the exception of specifying a variable in both **firstprivate** and **lastprivate** clauses applies after the clauses are split or duplicated per the above rules.

2.15 **if** Clause

Summary

The semantics of an **if** clause are described in the section on the construct to which it applies. The **if** clause *directive-name-modifier* names the associated construct to which an expression applies, and is particularly useful for composite and combined constructs.

Syntax

C / C++

The syntax of the **if** clause is as follows:

if ([*directive-name-modifier* :] *scalar-expression*)

C / C++

Fortran

The syntax of the **if** clause is as follows:

if ([*directive-name-modifier* :] *scalar-logical-expression*)

Fortran

Description

The effect of the **if** clause depends on the construct to which it is applied. For combined or composite constructs, the **if** clause only applies to the semantics of the construct named in the *directive-name-modifier* if one is specified. If no *directive-name-modifier* is specified for a combined or composite construct then the **if** clause applies to all constructs to which an **if** clause can apply.

2.16 master Construct

Summary

The **master** construct specifies a structured block that is executed by the master thread of the team.

Syntax

C / C++

The syntax of the **master** construct is as follows:

#pragma omp master *new-line*
structured-block

C / C++

Fortran

The syntax of the **master** construct is as follows:

!\$omp master
structured-block
!\$omp end master

Fortran

Binding

The binding thread set for a **master** region is the current team. A **master** region binds to the innermost enclosing **parallel** region.

Description

Only the master thread of the team that executes the binding **parallel** region participates in the execution of the structured block of the **master** region. Other threads in the team do not execute the associated structured block. There is no implied barrier either on entry to, or exit from, the **master** construct.

Execution Model Events

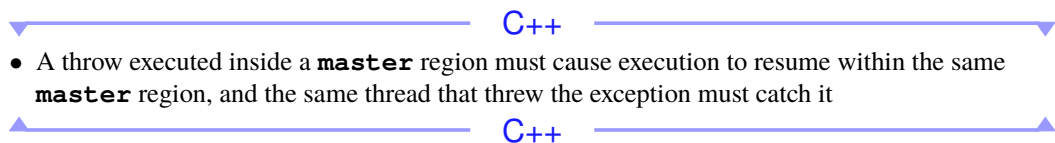
The *master-begin* event occurs in the master thread of a team that encounters the **master** construct on entry to the master region.

The *master-end* event occurs in the master thread of a team that encounters the **master** construct on exit from the master region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_master** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *master-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_master** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of a *master-end* event in that thread. These callbacks occur in the context of the task executed by the master thread and have the type signature **ompt_callback_master_t**.

Restrictions

- A blue-bordered box with a blue arrow pointing down on the left and a blue arrow pointing up on the right. The text "C++" is centered at the top and bottom of the box.
- A throw executed inside a **master** region must cause execution to resume within the same **master** region, and the same thread that threw the exception must catch it

Cross References

- **parallel** construct, see Section 2.6 on page 74.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.4.4.11 on page 443.
- **ompt_callback_master_t**, see Section 4.5.2.12 on page 473.

1 2.17 Synchronization Constructs and Clauses

2 A synchronization construct orders the completion of code executed by different threads. This
3 ordering is imposed by synchronizing flush operations that are executed as part of the region that
4 corresponds to the construct.

5 Synchronization through the use of synchronizing flush operations and atomic operations is
6 described in Section 1.4.4 on page 25 and Section 1.4.6 on page 28. Section 2.17.8.1 on page 246
7 defines the behavior of synchronizing flush operations that are implied at various other locations in
8 an OpenMP program.

9 2.17.1 critical Construct

10 Summary

11 The **critical** construct restricts execution of the associated structured block to a single thread at
12 a time.

13 Syntax

14 ▼ C / C++ ▼

The syntax of the **critical** construct is as follows:

```
15 #pragma omp critical [ (name) [[,] hint (hint-expression) ] ] new-line  
16     structured-block
```

17 where *hint-expression* is an integer constant expression that evaluates to a valid synchronization
18 hint (as described in Section 2.17.12 on page 260).

▲ C / C++ ▲

▼ Fortran ▼

19 The syntax of the **critical** construct is as follows:

```
20 !$omp critical [ (name) [[,] hint (hint-expression) ] ]  
21     structured-block  
22 !$omp end critical [ (name) ]
```

23 where *hint-expression* is a constant expression that evaluates to a scalar value with kind
24 **omp_sync_hint_kind** and a value that is a valid synchronization hint (as described
25 in Section 2.17.12 on page 260).

▲ Fortran ▲

Binding

The binding thread set for a **critical** region is all threads in the contention group.

Description

The region that corresponds to a **critical** construct is executed as if only a single thread at a time among all threads in the contention group enters the region for execution, without regard to the team(s) to which the threads belong. An optional *name* may be used to identify the **critical** construct. All **critical** constructs without a name are considered to have the same unspecified name.

C / C++

Identifiers used to identify a **critical** construct have external linkage and are in a name space that is separate from the name spaces used by labels, tags, members, and ordinary identifiers.

C / C++

Fortran

The names of **critical** constructs are global entities of the program. If a name conflicts with any other entity, the behavior of the program is unspecified.

Fortran

The threads of a contention group execute the **critical** region as if only one thread of the contention group executes the **critical** region at a time. The **critical** construct enforces these execution semantics with respect to all **critical** constructs with the same name in all threads in the contention group.

If present, the **hint** clause gives the implementation additional information about the expected runtime properties of the **critical** region that can optionally be used to optimize the implementation. The presence of a **hint** clause does not affect the isolation guarantees provided by the **critical** construct. If no **hint** clause is specified, the effect is as if **hint (omp_sync_hint_none)** had been specified.

Execution Model Events

The *critical-acquiring* event occurs in a thread that encounters the **critical** construct on entry to the **critical** region before initiating synchronization for the region.

The *critical-acquired* event occurs in a thread that encounters the **critical** construct after it enters the region, but before it executes the structured block of the **critical** region.

The *critical-released* event occurs in a thread that encounters the **critical** construct after it completes any synchronization on exit from the **critical** region.

Tool Callbacks

A thread dispatches a registered `ompt_callback_mutex_acquire` callback for each occurrence of a *critical-acquiring* event in that thread. This callback has the type signature `ompt_callback_mutex_acquire_t`.

A thread dispatches a registered `ompt_callback_mutex_acquired` callback for each occurrence of a *critical-acquired* event in that thread. This callback has the type signature `ompt_callback_mutex_t`.

A thread dispatches a registered `ompt_callback_mutex_released` callback for each occurrence of a *critical-released* event in that thread. This callback has the type signature `ompt_callback_mutex_t`.

The callbacks occur in the task that encounters the critical construct. The callbacks should receive `ompt_mutex_critical` as their *kind* argument if practical, but a less specific kind is acceptable.

Restrictions

The following restrictions apply to the critical construct:

- Unless the effect is as if `hint(omp_sync_hint_none)` was specified, the `critical` construct must specify a name.
- If the `hint` clause is specified, each of the `critical` constructs with the same *name* must have a `hint` clause for which the *hint-expression* evaluates to the same value.

C++

- A throw executed inside a `critical` region must cause execution to resume within the same `critical` region, and the same thread that threw the exception must catch it.

C++

Fortran

- If a *name* is specified on a `critical` directive, the same *name* must also be specified on the `end critical` directive.
- If no *name* appears on the `critical` directive, no *name* can appear on the `end critical` directive.

Fortran

Cross References

- Synchronization Hints, see Section 2.17.12 on page 260.
- `ompt_mutex_critical`, see Section 4.4.4.16 on page 445.
- `ompt_callback_mutex_acquire_t`, see Section 4.5.2.14 on page 476.
- `ompt_callback_mutex_t`, see Section 4.5.2.15 on page 477.

2.17.2 barrier Construct

Summary

The **barrier** construct specifies an explicit barrier at the point at which the construct appears. The **barrier** construct is a stand-alone directive.

Syntax

C / C++

The syntax of the **barrier** construct is as follows:

```
#pragma omp barrier new-line
```

C / C++

Fortran

The syntax of the **barrier** construct is as follows:

```
!$omp barrier
```

Fortran

Binding

The binding thread set for a **barrier** region is the current team. A **barrier** region binds to the innermost enclosing **parallel** region.

Description

All threads of the team that is executing the binding **parallel** region must execute the **barrier** region and complete execution of all explicit tasks bound to this **parallel** region before any are allowed to continue execution beyond the barrier.

The **barrier** region includes an implicit task scheduling point in the current task region.

Execution Model Events

The *explicit-barrier-begin* event occurs in each thread that encounters the **barrier** construct on entry to the **barrier** region.

The *explicit-barrier-wait-begin* event occurs when a task begins an interval of active or passive waiting in a **barrier** region.

The *explicit-barrier-wait-end* event occurs when a task ends an interval of active or passive waiting and resumes execution in a **barrier** region.

The *explicit-barrier-end* event occurs in each thread that encounters the **barrier** construct after the barrier synchronization on exit from the **barrier** region.

A *cancellation* event occurs if cancellation is activated at an implicit cancellation point in a **barrier** region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_sync_region** callback with **ompt_sync_region_barrier_explicit** — or **ompt_sync_region_barrier**, if the implementation cannot make a distinction — as its *kind* argument and **ompt_scope_begin** as its *endpoint* argument for each occurrence of an *explicit-barrier-begin* event in the task that encounters the **barrier** construct. Similarly, a thread dispatches a registered **ompt_callback_sync_region** callback with **ompt_sync_region_barrier_explicit** — or **ompt_sync_region_barrier**, if the implementation cannot make a distinction — as its *kind* argument and **ompt_scope_end** as its *endpoint* argument for each occurrence of an *explicit-barrier-end* event in the task that encounters the **barrier** construct. These callbacks occur in the task that encounters the **barrier** construct and have the type signature **ompt_callback_sync_region_t**.

A thread dispatches a registered **ompt_callback_sync_region_wait** callback with **ompt_sync_region_barrier_explicit** — or **ompt_sync_region_barrier**, if the implementation cannot make a distinction — as its *kind* argument and **ompt_scope_begin** as its *endpoint* argument for each occurrence of an *explicit-barrier-wait-begin* event. Similarly, a thread dispatches a registered **ompt_callback_sync_region_wait** callback with **ompt_sync_region_barrier_explicit** — or **ompt_sync_region_barrier**, if the implementation cannot make a distinction — as its *kind* argument and **ompt_scope_end** as its *endpoint* argument for each occurrence of an *explicit-barrier-wait-end* event. These callbacks occur in the context of the task that encountered the **barrier** construct and have type signature **ompt_callback_sync_region_t**.

A thread dispatches a registered **ompt_callback_cancel** callback with **ompt_cancel_detected** as its *flags* argument for each occurrence of a *cancellation* event in that thread. The callback occurs in the context of the encountering task. The callback has type signature **ompt_callback_cancel_t**.

Restrictions

The following restrictions apply to the **barrier** construct:

- Each **barrier** region must be encountered by all threads in a team or by none at all, unless cancellation has been requested for the innermost enclosing parallel region.
- The sequence of worksharing regions and **barrier** regions encountered must be the same for every thread in a team.

Cross References

- `ompt_scope_begin` and `ompt_scope_end`, see Section 4.4.4.11 on page 443.
- `ompt_sync_region_barrier`, see Section 4.4.4.13 on page 444.
- `ompt_callback_sync_region_t`, see Section 4.5.2.13 on page 474.
- `ompt_callback_cancel_t`, see Section 4.5.2.18 on page 481.

2.17.3 Implicit Barriers

This section describes the OMPT events and tool callbacks associated with implicit barriers, which occur at the end of various regions as defined in the description of the constructs to which they correspond. Implicit barriers are task scheduling points. For a description of task scheduling points, associated events, and tool callbacks, see Section 2.10.6 on page 149.

Execution Model Events

The *implicit-barrier-begin* event occurs in each implicit task at the beginning of an implicit barrier region.

The *implicit-barrier-wait-begin* event occurs when a task begins an interval of active or passive waiting in an implicit barrier region.

The *implicit-barrier-wait-end* event occurs when a task ends an interval of active or waiting and resumes execution of an implicit barrier region.

The *implicit-barrier-end* event occurs in each implicit task after the barrier synchronization on exit from an implicit barrier region.

A *cancellation* event occurs if cancellation is activated at an implicit cancellation point in an implicit barrier region.

Tool Callbacks

A thread dispatches a registered `ompt_callback_sync_region` callback with `ompt_sync_region_barrier_implicit` — or `ompt_sync_region_barrier`, if the implementation cannot make a distinction — as its *kind* argument and `ompt_scope_begin` as its *endpoint* argument for each occurrence of an *implicit-barrier-begin* event in that thread. Similarly, a thread dispatches a registered `ompt_callback_sync_region` callback with `ompt_sync_region_barrier_implicit` — or `ompt_sync_region_barrier`, if the implementation cannot make a distinction — as its *kind* argument and `ompt_scope_end` as its *endpoint* argument for each occurrence of an *implicit-barrier-end* event in that thread. These callbacks occur in the implicit task that executes the parallel region and have the type signature `ompt_callback_sync_region_t`.

A thread dispatches a registered `ompt_callback_sync_region_wait` callback with `ompt_sync_region_barrier_implicit` — or `ompt_sync_region_barrier`, if the implementation cannot make a distinction — as its *kind* argument and `ompt_scope_begin` as its *endpoint* argument for each occurrence of a *implicit-barrier-wait-begin* event in that thread. Similarly, a thread dispatches a registered `ompt_callback_sync_region_wait` callback with `ompt_sync_region_barrier_explicit` — or `ompt_sync_region_barrier`, if the implementation cannot make a distinction — as its *kind* argument and `ompt_scope_end` as its *endpoint* argument for each occurrence of an *implicit-barrier-wait-end* event in that thread. These callbacks occur in the implicit task that executes the parallel region and have type signature `ompt_callback_sync_region_t`.

A thread dispatches a registered `ompt_callback_cancel` callback with `ompt_cancel_detected` as its *flags* argument for each occurrence of a *cancellation* event in that thread. The callback occurs in the context of the encountering task. The callback has type signature `ompt_callback_cancel_t`.

Restrictions

If a thread is in the state `ompt_state_wait_barrier_implicit_parallel`, a call to `ompt_get_parallel_info` may return a pointer to a copy of the data object associated with the parallel region rather than a pointer to the associated data object itself. Writing to the data object returned by `ompt_get_parallel_info` when a thread is in the `ompt_state_wait_barrier_implicit_parallel` results in unspecified behavior.

Cross References

- `ompt_scope_begin` and `ompt_scope_end`, see Section 4.4.4.11 on page 443.
- `ompt_sync_region_barrier`, see Section 4.4.4.13 on page 444
- `ompt_cancel_detected`, see Section 4.4.4.24 on page 450.
- `ompt_callback_sync_region_t`, see Section 4.5.2.13 on page 474.
- `ompt_callback_cancel_t`, see Section 4.5.2.18 on page 481.

1 2.17.4 Implementation-Specific Barriers

2 An OpenMP implementation can execute implementation-specific barriers that are not implied by
3 the OpenMP specification; therefore, no *execution model events* are bound to these barriers. The
4 implementation can handle these barriers like implicit barriers and dispatch all events as for
5 implicit barriers. These callbacks are dispatched with
6 **ompt_sync_region_barrier_implementation** — or
7 **ompt_sync_region_barrier**, if the implementation cannot make a distinction — as the *kind*
8 argument.

9 2.17.5 taskwait Construct

10 Summary

11 The **taskwait** construct specifies a wait on the completion of child tasks of the current task. The
12 **taskwait** construct is a stand-alone directive.

13 Syntax

14  C / C++

14 The syntax of the **taskwait** construct is as follows:

15 **#pragma omp taskwait** [*clause* [,] *clause*] ...] *new-line*

16 where *clause* is one of the following:

17 **depend** ([*depend-modifier*,]*dependence-type* : *locator-list*)

18  C / C++

19  Fortran

18 The syntax of the **taskwait** construct is as follows:

19 **!\$omp taskwait** [*clause* [,] *clause*] ...]

20 where *clause* is one of the following:

21 **depend** ([*depend-modifier*,]*dependence-type* : *locator-list*)

22  Fortran

22 Binding

23 The **taskwait** region binds to the current task region. The binding thread set of the **taskwait**
24 region is the current team.

Description

If no **depend** clause is present on the **taskwait** construct, the current task region is suspended at an implicit task scheduling point associated with the construct. The current task region remains suspended until all child tasks that it generated before the **taskwait** region complete execution.

Otherwise, if one or more **depend** clauses are present on the **taskwait** construct, the behavior is as if these clauses were applied to a **task** construct with an empty associated structured block that generates a *mergeable* and *included task*. Thus, the current task region is suspended until the *predecessor tasks* of this task complete execution.

Execution Model Events

The *taskwait-begin* event occurs in each thread that encounters the **taskwait** construct on entry to the **taskwait** region.

The *taskwait-wait-begin* event occurs when a task begins an interval of active or passive waiting in a **taskwait** region.

The *taskwait-wait-end* event occurs when a task ends an interval of active or passive waiting and resumes execution in a **taskwait** region.

The *taskwait-end* event occurs in each thread that encounters the **taskwait** construct after the taskwait synchronization on exit from the **taskwait** region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_sync_region** callback with **ompt_sync_region_taskwait** as its *kind* argument and **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *taskwait-begin* event in the task that encounters the **taskwait** construct. Similarly, a thread dispatches a registered **ompt_callback_sync_region** callback with **ompt_sync_region_taskwait** as its *kind* argument and **ompt_scope_end** as its *endpoint* argument for each occurrence of a *taskwait-end* event in the task that encounters the **taskwait** construct. These callbacks occur in the task that encounters the **taskwait** construct and have the type signature **ompt_callback_sync_region_t**.

A thread dispatches a registered **ompt_callback_sync_region_wait** callback with **ompt_sync_region_taskwait** as its *kind* argument and **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *taskwait-wait-begin* event. Similarly, a thread dispatches a registered **ompt_callback_sync_region_wait** callback with **ompt_sync_region_taskwait** as its *kind* argument and **ompt_scope_end** as its *endpoint* argument for each occurrence of a *taskwait-wait-end* event. These callbacks occur in the context of the task that encounters the **taskwait** construct and have type signature **ompt_callback_sync_region_t**.

Restrictions

The following restrictions apply to the **taskwait** construct:

- The **mutexinoutset** *dependence-type* may not appear in a **depend** clause on a **taskwait** construct.
- If the *dependence-type* of a **depend** clause is **depobj** then the dependence objects cannot represent dependences of the **mutexinoutset** dependence type.

Cross References

- **task** construct, see Section 2.10.1 on page 135.
- Task scheduling, see Section 2.10.6 on page 149.
- **depend** clause, see Section 2.17.11 on page 255.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.4.4.11 on page 443.
- **ompt_sync_region_taskwait**, see Section 4.4.4.13 on page 444.
- **ompt_callback_sync_region_t**, see Section 4.5.2.13 on page 474.

2.17.6 taskgroup Construct

Summary

The **taskgroup** construct specifies a wait on completion of child tasks of the current task and their descendent tasks.

Syntax

C / C++

The syntax of the **taskgroup** construct is as follows:

```
#pragma omp taskgroup [clause[ [, ] clause] ...] new-line  
    structured-block
```

where *clause* is one of the following:

```
    task_reduction (reduction-identifier : list)  
    allocate ([allocator: ]list)
```

C / C++

Fortran

The syntax of the **taskgroup** construct is as follows:

```
!$omp taskgroup [clause [ [, ] clause ] ...]  
    structured-block  
!$omp end taskgroup
```

where *clause* is one of the following:

```
task_reduction (reduction-identifier : list)  
allocate ([allocator: ]list)
```

Fortran

Binding

The binding task set of a **taskgroup** region is all tasks of the current team that are generated in the region. A **taskgroup** region binds to the innermost enclosing **parallel** region.

Description

When a thread encounters a **taskgroup** construct, it starts executing the region. All child tasks generated in the **taskgroup** region and all of their descendants that bind to the same **parallel** region as the **taskgroup** region are part of the *taskgroup set* associated with the **taskgroup** region.

There is an implicit task scheduling point at the end of the **taskgroup** region. The current task is suspended at the task scheduling point until all tasks in the *taskgroup set* complete execution.

Execution Model Events

The *taskgroup-begin* event occurs in each thread that encounters the **taskgroup** construct on entry to the **taskgroup** region.

The *taskgroup-wait-begin* event occurs when a task begins an interval of active or passive waiting in a **taskgroup** region.

The *taskgroup-wait-end* event occurs when a task ends an interval of active or passive waiting and resumes execution in a **taskgroup** region.

The *taskgroup-end* event occurs in each thread that encounters the **taskgroup** construct after the taskgroup synchronization on exit from the **taskgroup** region.

Tool Callbacks

A thread dispatches a registered `ompt_callback_sync_region` callback with `ompt_sync_region_taskgroup` as its *kind* argument and `ompt_scope_begin` as its *endpoint* argument for each occurrence of a *taskgroup-begin* event in the task that encounters the `taskgroup` construct. Similarly, a thread dispatches a registered `ompt_callback_sync_region` callback with `ompt_sync_region_taskgroup` as its *kind* argument and `ompt_scope_end` as its *endpoint* argument for each occurrence of a *taskgroup-end* event in the task that encounters the `taskgroup` construct. These callbacks occur in the task that encounters the `taskgroup` construct and have the type signature `ompt_callback_sync_region_t`.

A thread dispatches a registered `ompt_callback_sync_region_wait` callback with `ompt_sync_region_taskgroup` as its *kind* argument and `ompt_scope_begin` as its *endpoint* argument for each occurrence of a *taskgroup-wait-begin* event. Similarly, a thread dispatches a registered `ompt_callback_sync_region_wait` callback with `ompt_sync_region_taskgroup` as its *kind* argument and `ompt_scope_end` as its *endpoint* argument for each occurrence of a *taskgroup-wait-end* event. These callbacks occur in the context of the task that encounters the `taskgroup` construct and have type signature `ompt_callback_sync_region_t`.

Cross References

- Task scheduling, see Section [2.10.6](#) on page [149](#).
- `task_reduction` Clause, see Section [2.19.5.5](#) on page [303](#).
- `ompt_scope_begin` and `ompt_scope_end`, see Section [4.4.4.11](#) on page [443](#).
- `ompt_sync_region_taskgroup`, see Section [4.4.4.13](#) on page [444](#).
- `ompt_callback_sync_region_t`, see Section [4.5.2.13](#) on page [474](#).

2.17.7 atomic Construct

Summary

The `atomic` construct ensures that a specific storage location is accessed atomically, rather than exposing it to the possibility of multiple, simultaneous reading and writing threads that may result in indeterminate values.

Syntax

In the following syntax, *atomic-clause* is a clause that indicates the semantics for which atomicity is enforced, *memory-order-clause* is a clause that indicates the memory ordering behavior of the construct and *clause* is a clause other than *atomic-clause*. Specifically, *atomic-clause* is one of the following:

```
read
write
update
capture
```

memory-order-clause is one of the following:

```
seq_cst
acq_rel
release
acquire
relaxed
```

and *clause* is either *memory-order-clause* or one of the following:

```
hint (hint-expression)
```

C / C++

The syntax of the **atomic** construct takes one of the following forms:

```
#pragma omp atomic [clause[[,] clause] ... ] [,] atomic-clause
                    [[,] clause [[,] clause] ... ] new-line
expression-stmt
```

or

```
#pragma omp atomic [clause[[,] clause] ... ] new-line
expression-stmt
```

or

```
#pragma omp atomic [clause[[,] clause] ... ] [,] capture
                    [[,] clause [[,] clause] ... ] new-line
structured-block
```

where *expression-stmt* is an expression statement with one of the following forms:

- If *atomic-clause* is **read**:

```
v = x;
```

- If *atomic-clause* is **write**:

```
x = expr;
```

- If *atomic-clause* is **update** or not present:

```
x++;
x--;
++x;
--x;
x binop= expr;
x = x binop expr;
x = expr binop x;
```

- If *atomic-clause* is **capture**:

```
v = x++;
v = x--;
v = ++x;
v = --x;
v = x binop= expr;
v = x = x binop expr;
v = x = expr binop x;
```

and where *structured-block* is a structured block with one of the following forms:

```
{ v = x; x binop= expr; }
{ x binop= expr; v = x; }
{ v = x; x = x binop expr; }
{ v = x; x = expr binop x; }
{ x = x binop expr; v = x; }
{ x = expr binop x; v = x; }
{ v = x; x = expr; }
{ v = x; x++; }
{ v = x; ++x; }
{ ++x; v = x; }
{ x++; v = x; }
{ v = x; x--; }
{ v = x; --x; }
{ --x; v = x; }
{ x--; v = x; }
```

In the preceding expressions:

- *x* and *v* (as applicable) are both *l-value* expressions with scalar type.
- During the execution of an atomic region, multiple syntactic occurrences of *x* must designate the same storage location.

- Neither of v and $expr$ (as applicable) may access the storage location designated by x .
- Neither of x and $expr$ (as applicable) may access the storage location designated by v .
- $expr$ is an expression with scalar type.
- $binop$ is one of $+$, $*$, $-$, $/$, $\&$, $^$, $|$, $<<$, or $>>$.
- $binop$, $binop=$, $++$, and $--$ are not overloaded operators.
- The expression $x \ binop \ expr$ must be numerically equivalent to $x \ binop \ (expr)$. This requirement is satisfied if the operators in $expr$ have precedence greater than $binop$, or by using parentheses around $expr$ or subexpressions of $expr$.
- The expression $expr \ binop \ x$ must be numerically equivalent to $(expr) \ binop \ x$. This requirement is satisfied if the operators in $expr$ have precedence equal to or greater than $binop$, or by using parentheses around $expr$ or subexpressions of $expr$.
- For forms that allow multiple occurrences of x , the number of times that x is evaluated is unspecified.
- $hint-expression$ is a constant integer expression that evaluates to a valid synchronization hint.

C / C++

Fortran

The syntax of the **atomic** construct takes any of the following forms:

```
!$omp atomic [clause[[,] clause] ... ] [, ] read [[,] clause [[[,] clause] ... ]]
    capture-statement
[!$omp end atomic]
```

or

```
!$omp atomic [clause[[,] clause] ... ] [, ] write [[,] clause [[[,] clause] ... ]]
    write-statement
[!$omp end atomic]
```

or

```
!$omp atomic [clause[[,] clause] ... ] [, ] update [[,] clause [[[,] clause] ... ]]
    update-statement
[!$omp end atomic]
```

or

```
!$omp atomic [clause[[,] clause] ... ]
    update-statement
[!$omp end atomic]
```

or

```

1  !$omp atomic [clause[[,] clause] ... ] [,] capture [[,] clause [[[,] clause] ... ]]
2      update-statement
3      capture-statement
4  !$omp end atomic

```

or

```

6  !$omp atomic [clause[[,] clause] ... ] [,] capture [[,] clause [[[,] clause] ... ]]
7      capture-statement
8      update-statement
9  !$omp end atomic

```

or

```

11 !$omp atomic [clause[[,] clause] ... ] [,] capture [[,] clause [[[,] clause] ... ]]
12     capture-statement
13     write-statement
14 !$omp end atomic

```

where *write-statement* has the following form (if *atomic-clause* is **capture** or **write**):

```

16     x = expr

```

where *capture-statement* has the following form (if *atomic-clause* is **capture** or **read**):

```

18     v = x

```

and where *update-statement* has one of the following forms (if *atomic-clause* is **update**, **capture**, or not present):

```

21     x = x operator expr
22
23     x = expr operator x
24
25     x = intrinsic_procedure_name (x, expr_list)
26
27     x = intrinsic_procedure_name (expr_list, x)

```

In the preceding statements:

- *x* and *v* (as applicable) are both scalar variables of intrinsic type.
- *x* must not have the **ALLOCATABLE** attribute.
- During the execution of an atomic region, multiple syntactic occurrences of *x* must designate the same storage location.
- None of *v*, *expr*, and *expr_list* (as applicable) may access the same storage location as *x*.

- None of x , $expr$, and $expr_list$ (as applicable) may access the same storage location as v .
- $expr$ is a scalar expression.
- $expr_list$ is a comma-separated, non-empty list of scalar expressions. If $intrinsic_procedure_name$ refers to **IAND**, **IOR**, or **IEOR**, exactly one expression must appear in $expr_list$.
- $intrinsic_procedure_name$ is one of **MAX**, **MIN**, **IAND**, **IOR**, or **IEOR**.
- $operator$ is one of **+**, *****, **-**, **/**, **.AND.**, **.OR.**, **.EQV.**, or **.NEQV.**
- The expression $x\ operator\ expr$ must be numerically equivalent to $x\ operator\ (expr)$. This requirement is satisfied if the operators in $expr$ have precedence greater than $operator$, or by using parentheses around $expr$ or subexpressions of $expr$.
- The expression $expr\ operator\ x$ must be numerically equivalent to $(expr)\ operator\ x$. This requirement is satisfied if the operators in $expr$ have precedence equal to or greater than $operator$, or by using parentheses around $expr$ or subexpressions of $expr$.
- $intrinsic_procedure_name$ must refer to the intrinsic procedure name and not to other program entities.
- $operator$ must refer to the intrinsic operator and not to a user-defined operator.
- All assignments must be intrinsic assignments.
- For forms that allow multiple occurrences of x , the number of times that x is evaluated is unspecified.
- $hint_expression$ is a constant expression that evaluates to a scalar value with kind **omp_sync_hint_kind** and a value that is a valid synchronization hint.

Fortran

Binding

If the size of x is 8, 16, 32, or 64 bits and x is aligned to a multiple of its size, the binding thread set for the **atomic** region is all threads on the device. Otherwise, the binding thread set for the **atomic** region is all threads in the contention group. **atomic** regions enforce exclusive access with respect to other **atomic** regions that access the same storage location x among all threads in the binding thread set without regard to the teams to which the threads belong.

Description

If *atomic-clause* is not present on the construct, the behavior is as if the **update** clause is specified.

The **atomic** construct with the **read** clause results in an atomic read of the location designated by x regardless of the native machine word size.

The **atomic** construct with the **write** clause results in an atomic write of the location designated by x regardless of the native machine word size.

The **atomic** construct with the **update** clause results in an atomic update of the location designated by x using the designated operator or intrinsic. Only the read and write of the location designated by x are performed mutually atomically. The evaluation of $expr$ or $expr_list$ need not be atomic with respect to the read or write of the location designated by x . No task scheduling points are allowed between the read and the write of the location designated by x .

The **atomic** construct with the **capture** clause results in an atomic captured update — an atomic update of the location designated by x using the designated operator or intrinsic while also capturing the original or final value of the location designated by x with respect to the atomic update. The original or final value of the location designated by x is written in the location designated by v based on the base language semantics of structured block or statements of the **atomic** construct. Only the read and write of the location designated by x are performed mutually atomically. Neither the evaluation of $expr$ or $expr_list$, nor the write to the location designated by v , need be atomic with respect to the read or write of the location designated by x . No task scheduling points are allowed between the read and the write of the location designated by x .

The **atomic** construct may be used to enforce memory consistency between threads, based on the guarantees provided by Section 1.4.6 on page 28. A strong flush on the location designated by x is performed on entry to and exit from the atomic operation, ensuring that the set of all atomic operations in the program applied to the same location has a total completion order. If the **write**, **update**, or **capture** clause is specified and the **release**, **acq_rel**, or **seq_cst** clause is specified then the strong flush on entry to the atomic operation is also a release flush. If the **read** or **capture** clause is specified and the **acquire**, **acq_rel**, or **seq_cst** clause is specified then the strong flush on exit from the atomic operation is also an acquire flush. Therefore, if *memory-order-clause* is specified and is not **relaxed**, release and/or acquire flush operations are implied and permit synchronization between the threads without the use of explicit **flush** directives.

For all forms of the **atomic** construct, any combination of two or more of these **atomic** constructs enforces mutually exclusive access to the locations designated by x among threads in the binding thread set. To avoid data races, all accesses of the locations designated by x that could potentially occur in parallel must be protected with an **atomic** construct.

atomic regions do not guarantee exclusive access with respect to any accesses outside of **atomic** regions to the same storage location x even if those accesses occur during a **critical** or **ordered** region, while an OpenMP lock is owned by the executing task, or during the execution of a **reduction** clause.

However, other OpenMP synchronization can ensure the desired exclusive access. For example, a barrier that follows a series of atomic updates to x guarantees that subsequent accesses do not form a race with the atomic accesses.

A compliant implementation may enforce exclusive access between **atomic** regions that update different storage locations. The circumstances under which this occurs are implementation defined.

If the storage location designated by *x* is not size-aligned (that is, if the byte alignment of *x* is not a multiple of the size of *x*), then the behavior of the **atomic** region is implementation defined.

If present, the **hint** clause gives the implementation additional information about the expected properties of the atomic operation that can optionally be used to optimize the implementation. The presence of a **hint** clause does not affect the semantics of the **atomic** construct, and all hints may be ignored. If no **hint** clause is specified, the effect is as if **hint(omp_sync_hint_none)** had been specified.

Execution Model Events

The *atomic-acquiring* event occurs in the thread that encounters the **atomic** construct on entry to the atomic region before initiating synchronization for the region.

The *atomic-acquired* event occurs in the thread that encounters the **atomic** construct after it enters the region, but before it executes the structured block of the **atomic** region.

The *atomic-released* event occurs in the thread that encounters the **atomic** construct after it completes any synchronization on exit from the **atomic** region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_mutex_acquire** callback for each occurrence of an *atomic-acquiring* event in that thread. This callback has the type signature **ompt_callback_mutex_acquire_t**.

A thread dispatches a registered **ompt_callback_mutex_acquired** callback for each occurrence of an *atomic-acquired* event in that thread. This callback has the type signature **ompt_callback_mutex_t**.

A thread dispatches a registered **ompt_callback_mutex_released** callback with **ompt_mutex_atomic** as the *kind* argument if practical, although a less specific *kind* may be used, for each occurrence of an *atomic-released* event in that thread. This callback has the type signature **ompt_callback_mutex_t** and occurs in the task that encounters the atomic construct.

Restrictions

The following restrictions apply to the **atomic** construct:

- OpenMP constructs may not be encountered during execution of an **atomic** region.
- At most one *memory-order-clause* may appear on the construct.
- At most one **hint** clause may appear on the construct.
- If *atomic-clause* is **read** then *memory-order-clause* must not be **acq_rel** or **release**.

- If *atomic-clause* is **write** then *memory-order-clause* must not be **acq_rel** or **acquire**.
- If *atomic-clause* is **update** or not present then *memory-order-clause* must not be **acq_rel** or **acquire**.

C / C++

- All atomic accesses to the storage locations designated by *x* throughout the program are required to have a compatible type.

C / C++

Fortran

- All atomic accesses to the storage locations designated by *x* throughout the program are required to have the same type and type parameters.

Fortran

Cross References

- **critical** construct, see Section 2.17.1 on page 223.
- **barrier** construct, see Section 2.17.2 on page 226.
- **flush** construct, see Section 2.17.8 on page 242.
- **ordered** construct, see Section 2.17.9 on page 250.
- Synchronization Hints, see Section 2.17.12 on page 260.
- **reduction** clause, see Section 2.19.5.4 on page 300.
- lock routines, see Section 3.3 on page 381.
- **ompt_mutex_atomic**, see Section 4.4.4.16 on page 445.
- **ompt_callback_mutex_acquire_t**, see Section 4.5.2.14 on page 476.
- **ompt_callback_mutex_t**, see Section 4.5.2.15 on page 477.

2.17.8 flush Construct

Summary

The **flush** construct executes the OpenMP flush operation. This operation makes a thread's temporary view of memory consistent with memory and enforces an order on the memory operations of the variables explicitly specified or implied. See the memory model description in Section 1.4 on page 23 for more details. The **flush** construct is a stand-alone directive.

Syntax

C / C++

The syntax of the **flush** construct is as follows:

```
#pragma omp flush [memory-order-clause] [(list)] new-line
```

where *memory-order-clause* is one of the following:

```
acq_rel  
release  
acquire
```

C / C++

Fortran

The syntax of the **flush** construct is as follows:

```
!$omp flush [memory-order-clause] [(list) ]
```

where *memory-order-clause* is one of the following:

```
acq_rel  
release  
acquire
```

Fortran

Binding

The binding thread set for a **flush** region is the encountering thread. Execution of a **flush** region affects the memory and the temporary view of memory of only the thread that executes the region. It does not affect the temporary view of other threads. Other threads must themselves execute a flush operation in order to be guaranteed to observe the effects of the flush operation of the encountering thread.

Description

If *memory-order-clause* is not specified then the **flush** construct results in a strong flush operation with the following behavior. A **flush** construct without a list, executed on a given thread, operates as if the whole thread-visible data state of the program, as defined by the base language, is flushed. A **flush** construct with a list applies the flush operation to the items in the list, and the flush operation does not complete until the operation is complete for all specified list items. An implementation may implement a **flush** with a list by ignoring the list, and treating it the same as a **flush** without a list.

If no list items are specified, the flush operation has the release and/or acquire flush properties:

- If *memory-order-clause* is not specified or is **acq_rel**, the flush operation is both a release flush and an acquire flush.
- If *memory-order-clause* is **release**, the flush operation is a release flush.
- If *memory-order-clause* is **acquire**, the flush operation is an acquire flush.

▼ C / C++ ▲

If a pointer is present in the list, the pointer itself is flushed, not the memory block to which the pointer refers.

▲ C / C++ ▲

▼ Fortran ▲

If the list item or a subobject of the list item has the **POINTER** attribute, the allocation or association status of the **POINTER** item is flushed, but the pointer target is not. If the list item is a Cray pointer, the pointer is flushed, but the object to which it points is not. If the list item is of type **C_PTR**, the variable is flushed, but the storage that corresponds to that address is not flushed. If the list item or the subobject of the list item has the **ALLOCATABLE** attribute and has an allocation status of allocated, the allocated variable is flushed; otherwise the allocation status is flushed.

▲ Fortran ▲

Note – Use of a **flush** construct with a list is extremely error prone and users are strongly discouraged from attempting it. The following examples illustrate the ordering properties of the flush operation. In the following incorrect pseudocode example, the programmer intends to prevent simultaneous execution of the protected section by the two threads, but the program does not work properly because it does not enforce the proper ordering of the operations on variables **a** and **b**. Any shared data accessed in the protected section is not guaranteed to be current or consistent during or after the protected section. The atomic notation in the pseudocode in the following two examples indicates that the accesses to **a** and **b** are atomic write and atomic read operations. Otherwise both examples would contain data races and automatically result in unspecified behavior. The *flush* operations are strong flushes that are applied to the specified flush lists

Incorrect example:

a = b = 0

thread 1

thread 2

atomic(b = 1)

atomic(a = 1)

flush (b)

flush (a)

flush (a)

flush (b)

atomic(tmp = a)

atomic(tmp = b)

if (tmp == 0) then

if (tmp == 0) then

protected section

protected section

end if

end if

The problem with this example is that operations on variables **a** and **b** are not ordered with respect to each other. For instance, nothing prevents the compiler from moving the flush of **b** on thread 1 or the flush of **a** on thread 2 to a position completely after the protected section (assuming that the protected section on thread 1 does not reference **b** and the protected section on thread 2 does not reference **a**). If either re-ordering happens, both threads can simultaneously execute the protected section.

The following pseudocode example correctly ensures that the protected section is executed by not more than one of the two threads at any one time. Execution of the protected section by neither thread is considered correct in this example. This occurs if both flushes complete prior to either thread executing its **if** statement.

Correct example:

a = b = 0

thread 1

thread 2

atomic(b = 1)

atomic(a = 1)

flush (a, b)

flush (a, b)

atomic(tmp = a)

atomic(tmp = b)

if (tmp == 0) then

if (tmp == 0) then

protected section

protected section

end if

end if

The compiler is prohibited from moving the flush at all for either thread, ensuring that the respective assignment is complete and the data is flushed before the **if** statement is executed.

Execution Model Events

The *flush* event occurs in a thread that encounters the **flush** construct.

Tool Callbacks

A thread dispatches a registered **ompt_callback_flush** callback for each occurrence of a *flush* event in that thread. This callback has the type signature **ompt_callback_flush_t**.

Restrictions

The following restrictions apply to the **flush** construct:

- If *memory-order-clause* is **release**, **acquire**, or **acq_rel**, list items must not be specified on the **flush** directive.

Cross References

- **ompt_callback_flush_t**, see Section [4.5.2.17](#) on page 480.

2.17.8.1 Implicit Flushes

Flush operations implied when executing an **atomic** region are described in Section [2.17.7](#).

A **flush** region that corresponds to a **flush** directive with the **release** clause present is implied at the following locations:

- During a barrier region;
- At entry to a **parallel** region;
- At entry to a **teams** region;
- At exit from a **critical** region;
- During an **omp_unset_lock** region;
- During an **omp_unset_nest_lock** region;
- Immediately before every task scheduling point;

- At exit from the task region of each implicit task;
 - At exit from an **ordered** region, if a **threads** clause or a **depend** clause with a **source** dependence type is present, or if no clauses are present; and
 - During a **cancel** region, if the *cancel-var* ICV is *true*.
- A **flush** region that corresponds to a **flush** directive with the **acquire** clause present is implied at the following locations:
- During a barrier region;
 - At exit from a **teams** region;
 - At entry to a **critical** region;
 - If the region causes the lock to be set, during:
 - an **omp_set_lock** region;
 - an **omp_test_lock** region;
 - an **omp_set_nest_lock** region; and
 - an **omp_test_nest_lock** region;
 - Immediately after every task scheduling point;
 - At entry to the task region of each implicit task;
 - At entry to an **ordered** region, if a **threads** clause or a **depend** clause with a **sink** dependence type is present, or if no clauses are present; and
 - Immediately before a cancellation point, if the *cancel-var* ICV is *true* and cancellation has been activated.

Note – A **flush** region is not implied at the following locations:

- At entry to worksharing regions; and
 - At entry to or exit from **master** regions.
-

The synchronization behavior of implicit flushes is as follows:

- When a thread executes an **atomic** region for which the corresponding construct has the **release**, **acq_rel**, or **seq_cst** clause and specifies an atomic operation that starts a given release sequence, the release flush that is performed on entry to the atomic operation synchronizes with an acquire flush that is performed by a different thread and has an associated atomic operation that reads a value written by a modification in the release sequence.

- When a thread executes an **atomic** region for which the corresponding construct has the **acquire**, **acq_rel**, or **seq_cst** clause and specifies an atomic operation that reads a value written by a given modification, a release flush that is performed by a different thread and has an associated release sequence that contains that modification synchronizes with the acquire flush that is performed on exit from the atomic operation.
- When a thread executes a **critical** region that has a given name, the behavior is as if the release flush performed on exit from the region synchronizes with the acquire flush performed on entry to the next **critical** region with the same name that is performed by a different thread, if it exists.
- When a thread team executes a **barrier** region, the behavior is as if the release flush performed by each thread within the region synchronizes with the acquire flush performed by all other threads within the region.
- When a thread executes a **taskwait** region that does not result in the creation of a dependent task, the behavior is as if each thread that executes a remaining child task performs a release flush upon completion of the child task that synchronizes with an acquire flush performed in the **taskwait** region.
- When a thread executes a **taskgroup** region, the behavior is as if each thread that executes a remaining descendant task performs a release flush upon completion of the descendant task that synchronizes with an acquire flush performed on exit from the **taskgroup** region.
- When a thread executes an **ordered** region that does not arise from a stand-alone **ordered** directive, the behavior is as if the release flush performed on exit from the region synchronizes with the acquire flush performed on entry to an **ordered** region encountered in the next logical iteration to be executed by a different thread, if it exists.
- When a thread executes an **ordered** region that arises from a stand-alone **ordered** directive, the behavior is as if the release flush performed in the **ordered** region from a given source iteration synchronizes with the acquire flush performed in all **ordered** regions executed by a different thread that are waiting for dependences on that iteration to be satisfied.
- When a thread team begins execution of a **parallel** region, the behavior is as if the release flush performed by the master thread on entry to the **parallel** region synchronizes with the acquire flush performed on entry to each implicit task that is assigned to a different thread.
- When an initial thread begins execution of a **target** region that is generated by a different thread from a target task, the behavior is as if the release flush performed by the generating thread in the target task synchronizes with the acquire flush performed by the initial thread on entry to its initial task region.
- When an initial thread completes execution of a **target** region that is generated by a different thread from a target task, the behavior is as if the release flush performed by the initial thread on exit from its initial task region synchronizes with the acquire flush performed by the generating thread in the target task.

- 1 • When a thread encounters a **teams** construct, the behavior is as if the release flush performed by
2 the thread on entry to the **teams** region synchronizes with the acquire flush performed on entry
3 to each initial task that is executed by a different initial thread that participates in the execution of
4 the **teams** region.
- 5 • When a thread that encounters a **teams** construct reaches the end of the **teams** region, the
6 behavior is as if the release flush performed by each different participating initial thread at exit
7 from its initial task synchronizes with the acquire flush performed by the thread at exit from the
8 **teams** region.
- 9 • When a task generates an explicit task that begins execution on a different thread, the behavior is
10 as if the thread that is executing the generating task performs a release flush that synchronizes
11 with the acquire flush performed by the thread that begins to execute the explicit task.
- 12 • When an undeferred task completes execution on a given thread that is different from the thread
13 on which its generating task is suspended, the behavior is as if a release flush performed by the
14 thread that completes execution of the undeferred task synchronizes with an acquire flush
15 performed by the thread that resumes execution of the generating task.
- 16 • When a dependent task with one or more predecessor tasks begins execution on a given thread,
17 the behavior is as if each release flush performed by a different thread on completion of a
18 predecessor task synchronizes with the acquire flush performed by the thread that begins to
19 execute the dependent task.
- 20 • When a task begins execution on a given thread and it is mutually exclusive with respect to
21 another sibling task that is executed by a different thread, the behavior is as if each release flush
22 performed on completion of the sibling task synchronizes with the acquire flush performed by
23 the thread that begins to execute the task.
- 24 • When a thread executes a **cancel** region, the *cancel-var* ICV is *true*, and cancellation is not
25 already activated for the specified region, the behavior is as if the release flush performed during
26 the **cancel** region synchronizes with the acquire flush performed by a different thread
27 immediately before a cancellation point in which that thread observes cancellation was activated
28 for the region.
- 29 • When a thread executes an **omp_unset_lock** region that causes the specified lock to be unset,
30 the behavior is as if a release flush is performed during the **omp_unset_lock** region that
31 synchronizes with an acquire flush that is performed during the next **omp_set_lock** or
32 **omp_test_lock** region to be executed by a different thread that causes the specified lock to be
33 set.
- 34 • When a thread executes an **omp_unset_nest_lock** region that causes the specified nested
35 lock to be unset, the behavior is as if a release flush is performed during the
36 **omp_unset_nest_lock** region that synchronizes with an acquire flush that is performed
37 during the next **omp_set_nest_lock** or **omp_test_nest_lock** region to be executed by
38 a different thread that causes the specified nested lock to be set.

1 2.17.9 ordered Construct

2 Summary

3 The **ordered** construct either specifies a structured block in a worksharing-loop, **simd**, or
4 worksharing-loop SIMD region that will be executed in the order of the loop iterations, or it is a
5 stand-alone directive that specifies cross-iteration dependences in a doacross loop nest. The
6 **ordered** construct sequentializes and orders the execution of **ordered** regions while allowing
7 code outside the region to run in parallel.

8 Syntax

9  C / C++

The syntax of the **ordered** construct is as follows:

```
10 #pragma omp ordered [clause[ [, ] clause] ] new-line
11     structured-block
```

12 where *clause* is one of the following:

```
13     threads
14     simd
```

15 or

```
16 #pragma omp ordered clause [[[ , ] clause] ... ] new-line
```

17 where *clause* is one of the following:

```
18     depend(source)
19     depend(sink : vec)
```

20  C / C++

21  Fortran

22 The syntax of the **ordered** construct is as follows:

```
21 !$omp ordered [clause[ [, ] clause] ]
22     structured-block
23 !$omp end ordered
```

24 where *clause* is one of the following:

```
25     threads
26     simd
```

27 or

```
28 !$omp ordered clause [[[ , ] clause] ... ]
```

where *clause* is one of the following:

```
depend(source)
depend(sink : vec)
```

Fortran

If the **depend** clause is specified, the **ordered** construct is a stand-alone directive.

Binding

The binding thread set for an **ordered** region is the current team. An **ordered** region binds to the innermost enclosing **simd** or worksharing-loop SIMD region if the **simd** clause is present, and otherwise it binds to the innermost enclosing worksharing-loop region. **ordered** regions that bind to different regions execute independently of each other.

Description

If no clause is specified, the **ordered** construct behaves as if the **threads** clause had been specified. If the **threads** clause is specified, the threads in the team that is executing the worksharing-loop region execute **ordered** regions sequentially in the order of the loop iterations. If any **depend** clauses are specified then those clauses specify the order in which the threads in the team execute **ordered** regions. If the **simd** clause is specified, the **ordered** regions encountered by any thread will execute one at a time in the order of the loop iterations.

When the thread that is executing the first iteration of the loop encounters an **ordered** construct, it can enter the **ordered** region without waiting. When a thread that is executing any subsequent iteration encounters an **ordered** construct without a **depend** clause, it waits at the beginning of the **ordered** region until execution of all **ordered** regions belonging to all previous iterations has completed. When a thread that is executing any subsequent iteration encounters an **ordered** construct with one or more **depend(sink:vec)** clauses, it waits until its dependences on all valid iterations specified by the **depend** clauses are satisfied before it completes execution of the **ordered** region. A specific dependence is satisfied when a thread that is executing the corresponding iteration encounters an **ordered** construct with a **depend(source)** clause.

Execution Model Events

The *ordered-acquiring* event occurs in the task that encounters the **ordered** construct on entry to the ordered region before it initiates synchronization for the region.

The *ordered-acquired* event occurs in the task that encounters the **ordered** construct after it enters the region, but before it executes the structured block of the **ordered** region.

The *ordered-released* event occurs in the task that encounters the **ordered** construct after it completes any synchronization on exit from the **ordered** region.

The *doacross-sink* event occurs in the task that encounters a **ordered** construct for each **depend(sink:vec)** clause after the dependence is fulfilled.

The *doacross-source* event occurs in the task that encounters a **ordered** construct with a **depend(source:vec)** clause before signaling the dependence to be fulfilled.

Tool Callbacks

A thread dispatches a registered **ompt_callback_mutex_acquire** callback for each occurrence of an *ordered-acquiring* event in that thread. This callback has the type signature **ompt_callback_mutex_acquire_t**.

A thread dispatches a registered **ompt_callback_mutex_acquired** callback for each occurrence of an *ordered-acquired* event in that thread. This callback has the type signature **ompt_callback_mutex_t**.

A thread dispatches a registered **ompt_callback_mutex_released** callback with **ompt_mutex_ordered** as the *kind* argument if practical, although a less specific kind may be used, for each occurrence of an *ordered-released* event in that thread. This callback has the type signature **ompt_callback_mutex_t** and occurs in the task that encounters the atomic construct.

A thread dispatches a registered **ompt_callback_dependencies** callback with all vector entries listed as **ompt_dependence_type_sink** in the *deps* argument for each occurrence of a *doacross-sink* event in that thread. A thread dispatches a registered **ompt_callback_dependencies** callback with all vector entries listed as **ompt_dependence_type_source** in the *deps* argument for each occurrence of a *doacross-source* event in that thread. These callbacks have the type signature **ompt_callback_dependencies_t**.

Restrictions

Restrictions to the **ordered** construct are as follows:

- At most one **threads** clause can appear on an **ordered** construct.
- At most one **simd** clause can appear on an **ordered** construct.
- At most one **depend(source)** clause can appear on an **ordered** construct.
- The construct corresponding to the binding region of an **ordered** region must not specify a **reduction** clause with the **inscan** modifier.
- Either **depend(sink:vec)** clauses or **depend(source)** clauses may appear on an **ordered** construct, but not both.

- The worksharing-loop or worksharing-loop SIMD region to which an **ordered** region corresponding to an **ordered** construct without a **depend** clause binds must have an **ordered** clause without the parameter specified on the corresponding worksharing-loop or worksharing-loop SIMD directive.
- The worksharing-loop region to which an **ordered** region corresponding to an **ordered** construct with any **depend** clauses binds must have an **ordered** clause with the parameter specified on the corresponding worksharing-loop directive.
- An **ordered** construct with the **depend** clause specified must be closely nested inside a worksharing-loop (or parallel worksharing-loop) construct.
- An **ordered** region corresponding to an **ordered** construct without the **simd** clause specified must be closely nested inside a loop region.
- An **ordered** region corresponding to an **ordered** construct with the **simd** clause specified must be closely nested inside a **simd** or worksharing-loop SIMD region.
- An **ordered** region corresponding to an **ordered** construct with both the **simd** and **threads** clauses must be closely nested inside a worksharing-loop SIMD region or must be closely nested inside a worksharing-loop and **simd** region.
- During execution of an iteration of a worksharing-loop or a loop nest within a worksharing-loop, **simd**, or worksharing-loop SIMD region, a thread must not execute more than one **ordered** region corresponding to an **ordered** construct without a **depend** clause.

C++

- A throw executed inside a **ordered** region must cause execution to resume within the same **ordered** region, and the same thread that threw the exception must catch it.

C++

Cross References

- worksharing-loop construct, see Section 2.9.2 on page 101.
- **simd** construct, see Section 2.9.3.1 on page 110.
- parallel Worksharing-loop construct, see Section 2.13.1 on page 185.
- **depend** Clause, see Section 2.17.11 on page 255
- **ompt_mutex_ordered**, see Section 4.4.4.16 on page 445.
- **ompt_callback_mutex_acquire_t**, see Section 4.5.2.14 on page 476.
- **ompt_callback_mutex_t**, see Section 4.5.2.15 on page 477.

1 2.17.10 Depend Objects

2 This section describes constructs that support OpenMP depend objects that can be used to supply
3 user-computed dependences to **depend** clauses. OpenMP depend objects must be accessed only
4 through the **depobj** construct or through the **depend** clause; programs that otherwise access
5 OpenMP depend objects are non-conforming.

6 An OpenMP depend object can be in one of the following states: *uninitialized* or *initialized*.
7 Initially OpenMP depend objects are in the *uninitialized* state.

8 2.17.10.1 depobj Construct

9 Summary

10 The **depobj** construct initializes, updates or destroys an OpenMP depend object. The **depobj**
11 construct is a stand-alone directive.

12 Syntax

13  C / C++ 

14 The syntax of the **depobj** construct is as follows:

```
15 #pragma omp depobj (depobj) clause new-line
```

16 where *depobj* is an lvalue expression of type **omp_depend_t**.

17 where *clause* is one of the following:

```
18 depend (dependence-type : locator)  
19 destroy  
20 update (dependence-type)
```

21  C / C++ 
22  Fortran 

23 The syntax of the **depobj** construct is as follows:

```
24 !$omp depobj (depobj) clause
```

25 where *depobj* is a scalar integer variable of the **omp_depend_kind** kind.

26 where *clause* is one of the following:

```
27 depend (dependence-type : locator)  
28 destroy  
29 update (dependence-type)
```

30  Fortran 

Binding

The binding thread set for **depobj** regions is the encountering thread.

Description

A **depobj** construct with a **depend** clause present sets the state of *depobj* to initialized. The *depobj* is initialized to represent the dependence that the **depend** clause specifies.

A **depobj** construct with a **destroy** clause present changes the state of the *depobj* to uninitialized.

A **depobj** construct with an **update** clause present changes the dependence type of the dependence represented by *depobj* to the one specified by the *update* clause.

Restrictions

- A **depend** clause on a **depobj** construct must not have **source**, **sink** or **depobj** as *dependence-type*.
- A **depend** clause on a **depobj** construct can only specify one locator.
- The *depobj* of a **depobj** construct with the **depend** clause present must be in the uninitialized state.
- The *depobj* of a **depobj** construct with the **destroy** clause present must be in the initialized state.
- The *depobj* of a **depobj** construct with the **update** clause present must be in the initialized state.

Cross References

- **depend** clause, see Section [2.17.11](#) on page [255](#).

2.17.11 depend Clause

Summary

The **depend** clause enforces additional constraints on the scheduling of tasks or loop iterations. These constraints establish dependences only between sibling tasks or between loop iterations.

Syntax

The syntax of the **depend** clause is as follows:

```
depend ([depend-modifier, ]dependence-type : locator-list)
```

where *dependence-type* is one of the following:

```
in  
out  
inout  
mutexinoutset  
depobj
```

where *depend-modifier* is one of the following:

```
iterator (iterators-definition)
```

or

```
depend (dependence-type)
```

where *dependence-type* is:

```
source
```

or

```
depend (dependence-type : vec)
```

where *dependence-type* is:

```
sink
```

and where *vec* is the iteration vector, which has the form:

$$x_1 [\pm d_1], x_2 [\pm d_2], \dots, x_n [\pm d_n]$$

where n is the value specified by the **ordered** clause in the worksharing-loop directive, x_i denotes the loop iteration variable of the i -th nested loop associated with the worksharing-loop directive, and d_i is a constant non-negative integer.

Description

Task dependences are derived from the *dependence-type* of a **depend** clause and its list items when *dependence-type* is **in**, **out**, **inout**, or **mutexinoutset**. When the *dependence-type* is **depobj**, the task dependences are derived from the dependences represented by the depend objects specified in the **depend** clause as if the **depend** clauses of the **depobj** constructs were specified in the current construct.

For the **in** *dependence-type*, if the storage location of at least one of the list items is the same as the storage location of a list item appearing in a **depend** clause with an **out**, **inout**, or **mutexinoutset** *dependence-type* on a construct from which a sibling task was previously generated, then the generated task will be a dependent task of that sibling task.

For the **out** and **inout** *dependence-types*, if the storage location of at least one of the list items is the same as the storage location of a list item appearing in a **depend** clause with an **in**, **out**, **inout**, or **mutexinoutset** *dependence-type* on a construct from which a sibling task was previously generated, then the generated task will be a dependent task of that sibling task.

For the **mutexinoutset** *dependence-type*, if the storage location of at least one of the list items is the same as the storage location of a list item appearing in a **depend** clause with an **in**, **out**, or **inout** *dependence-type* on a construct from which a sibling task was previously generated, then the generated task will be a dependent task of that sibling task.

If a list item appearing in a **depend** clause with a **mutexinoutset** *dependence-type* on a task-generating construct has the same storage location as a list item appearing in a **depend** clause with a **mutexinoutset** *dependence-type* on a different task generating construct, and both constructs generate sibling tasks, the sibling tasks will be mutually exclusive tasks.

The list items that appear in the **depend** clause may reference iterators defined by an *iterators-definition* appearing on an **iterator** modifier.

The list items that appear in the **depend** clause may include array sections.

Fortran

If a list item has the **ALLOCATABLE** attribute and its allocation status is unallocated, the behavior is unspecified. If a list item has the **POINTER** attribute and its association status is disassociated or undefined, the behavior is unspecified.

Fortran

C / C++

The list items that appear in a **depend** clause may use shape-operators.

C / C++

Note – The enforced task dependence establishes a synchronization of memory accesses performed by a dependent task with respect to accesses performed by the predecessor tasks. However, it is the responsibility of the programmer to synchronize properly with respect to other concurrent accesses that occur outside of those tasks.

The **source** *dependence-type* specifies the satisfaction of cross-iteration dependences that arise from the current iteration.

The **sink** *dependence-type* specifies a cross-iteration dependence, where the iteration vector *vec* indicates the iteration that satisfies the dependence.

If the iteration vector *vec* does not occur in the iteration space, the **depend** clause is ignored. If all **depend** clauses on an **ordered** construct are ignored then the construct is ignored.

Note – An iteration vector *vec* that does not indicate a lexicographically earlier iteration may cause a deadlock.

Execution Model Events

The *task-dependences* event occurs in a thread that encounters a task generating construct or a **taskwait** construct with a **depend** clause immediately after the *task-create* event for the new task or the *taskwait-begin* event.

The *task-dependence* event indicates an unfulfilled dependence for the generated task. This event occurs in a thread that observes the unfulfilled dependence before it is satisfied.

Tool Callbacks

A thread dispatches the **ompt_callback_dependences** callback for each occurrence of the *task-dependences* event to announce its dependences with respect to the list items in the **depend** clause. This callback has type signature **ompt_callback_dependences_t**.

A thread dispatches the **ompt_callback_task_dependence** callback for a *task-dependence* event to report a dependence between a predecessor task (*src_task_data*) and a dependent task (*sink_task_data*). This callback has type signature **ompt_callback_task_dependence_t**.

Restrictions

Restrictions to the **depend** clause are as follows:

- List items used in **depend** clauses of the same task or sibling tasks must indicate identical storage locations or disjoint storage locations.
- List items used in **depend** clauses cannot be zero-length array sections.
- Array sections cannot be specified in **depend** clauses with the **depobj** dependence type.
- List items used in **depend** clauses with the **depobj** dependence type must be depend objects in the initialized state.

C / C++

- List items used in **depend** clauses with the **depobj** dependence type must be expressions of the **omp_depend_t** type.

- List items used in **depend** clauses with the **in**, **out**, **inout** or **mutexinoutset** dependence types cannot be expressions of the **omp_depend_t** type.

C / C++

Fortran

- A common block name cannot appear in a **depend** clause.
- List items used in **depend** clauses with the **depobj** dependence type must be integer expressions of the **omp_depend_kind** kind.

Fortran

- For a *vec* element of **sink** *dependence-type* of the form $x_i + d_i$ or $x_i - d_i$ if the loop iteration variable x_i has an integral or pointer type, the expression $x_i + d_i$ or $x_i - d_i$ for any value of the loop iteration variable x_i that can encounter the **ordered** construct must be computable without overflow in the type of the loop iteration variable.

C++

- For a *vec* element of **sink** *dependence-type* of the form $x_i + d_i$ or $x_i - d_i$ if the loop iteration variable x_i is of a random access iterator type other than pointer type, the expression $(x_i - lb_i) + d_i$ or $(x_i - lb_i) - d_i$ for any value of the loop iteration variable x_i that can encounter the **ordered** construct must be computable without overflow in the type that would be used by `std::distance` applied to variables of the type of x_i .

C++

C / C++

- A bit-field cannot appear in a **depend** clause.

C / C++

Cross References

- Array sections, see Section 2.1.5 on page 44.
- Iterators, see Section 2.1.6 on page 47.
- **task** construct, see Section 2.10.1 on page 135.
- Task scheduling constraints, see Section 2.10.6 on page 149.
- **target enter data** construct, see Section 2.12.3 on page 164.
- **target exit data** construct, see Section 2.12.4 on page 166.
- **target** construct, see Section 2.12.5 on page 170.
- **target update** construct, see Section 2.12.6 on page 176.
- **ordered** construct, see Section 2.17.9 on page 250.
- **depobj** construct, see Section 2.17.10.1 on page 254.
- **omp_callback_dependences_t**, see Section 4.5.2.8 on page 468.
- **omp_callback_task_dependence_t**, see Section 4.5.2.9 on page 470.

1 2.17.12 Synchronization Hints

2 Hints about the expected dynamic behavior or suggested implementation can be provided by the
3 programmer to locks (by using the `omp_init_lock_with_hint` or
4 `omp_init_nest_lock_with_hint` functions to initialize the lock), and to `atomic` and
5 `critical` directives by using the `hint` clause. The effect of a hint does not change the semantics
6 of the associated construct; if ignoring the hint changes the program semantics, the result is
7 unspecified.

8 The C/C++ header file (`omp.h`) and the Fortran include file (`omp_lib.h`) and/or Fortran 90
9 module file (`omp_lib`) define the valid hint constants. The valid constants must include the
10 following, which can be extended with implementation-defined values:

C / C++

```
11 typedef enum omp_sync_hint_t {  
12     omp_sync_hint_none = 0x0,  
13     omp_lock_hint_none = omp_sync_hint_none,  
14     omp_sync_hint_uncontended = 0x1,  
15     omp_lock_hint_uncontended = omp_sync_hint_uncontended,  
16     omp_sync_hint_contended = 0x2,  
17     omp_lock_hint_contended = omp_sync_hint_contended,  
18     omp_sync_hint_nonspeculative = 0x4,  
19     omp_lock_hint_nonspeculative = omp_sync_hint_nonspeculative,  
20     omp_sync_hint_speculative = 0x8  
21     omp_lock_hint_speculative = omp_sync_hint_speculative  
22 } omp_sync_hint_t;  
23  
24 typedef omp_sync_hint_t omp_lock_hint_t;
```

C / C++

Fortran

```
25 integer, parameter :: omp_lock_hint_kind = omp_sync_hint_kind  
26  
27 integer (kind=omp_sync_hint_kind), &  
28     parameter :: omp_sync_hint_none = &  
29         int('0', kind=omp_sync_hint_kind)  
30 integer (kind=omp_lock_hint_kind), &  
31     parameter :: omp_lock_hint_none = omp_sync_hint_none  
32 integer (kind=omp_sync_hint_kind), &  
33     parameter :: omp_sync_hint_uncontended = &  
34         int('1', kind=omp_sync_hint_kind)  
35 integer (kind=omp_lock_hint_kind), &  
36     parameter :: omp_lock_hint_uncontended = &  
37         omp_sync_hint_uncontended  
38 integer (kind=omp_sync_hint_kind), &
```



```

1      parameter :: omp_sync_hint_contended = &
2          int(Z'2', kind=omp_sync_hint_kind)
3  integer (kind=omp_lock_hint_kind), &
4      parameter :: omp_lock_hint_contended = &
5          omp_sync_hint_contended
6  integer (kind=omp_sync_hint_kind), &
7      parameter :: omp_sync_hint_nonspeculative = &
8          int(Z'4', kind=omp_sync_hint_kind)
9  integer (kind=omp_lock_hint_kind), &
10     parameter :: omp_lock_hint_nonspeculative = &
11         omp_sync_hint_nonspeculative
12 integer (kind=omp_sync_hint_kind), &
13     parameter :: omp_sync_hint_speculative = &
14         int(Z'8', kind=omp_sync_hint_kind)
15 integer (kind=omp_lock_hint_kind), &
16     parameter :: omp_lock_hint_speculative = &
17         omp_sync_hint_speculative

```

Fortran

The hints can be combined by using the + or | operators in C/C++ or the + operator in Fortran. Combining **omp_sync_hint_none** with any other hint is equivalent to specifying the other hint.

The intended meaning of each hint is:

- **omp_sync_hint_uncontended**: low contention is expected in this operation, that is, few threads are expected to perform the operation simultaneously in a manner that requires synchronization;
- **omp_sync_hint_contended**: high contention is expected in this operation, that is, many threads are expected to perform the operation simultaneously in a manner that requires synchronization;
- **omp_sync_hint_speculative**: the programmer suggests that the operation should be implemented using speculative techniques such as transactional memory; and
- **omp_sync_hint_nonspeculative**: the programmer suggests that the operation should not be implemented using speculative techniques such as transactional memory.

Note – Future OpenMP specifications may add additional hints to the **omp_sync_hint_t** type and the **omp_sync_hint_kind** kind. Implementers are advised to add implementation-defined hints starting from the most significant bit of the **omp_sync_hint_t** type and **omp_sync_hint_kind** kind and to include the name of the implementation in the name of the added hint to avoid name conflicts with other OpenMP implementations.

The `omp_sync_hint_t` and `omp_lock_hint_t` enumeration types and the equivalent types in Fortran are synonyms for each other. The type `omp_lock_hint_t` has been deprecated.

Restrictions

Restrictions to the synchronization hints are as follows:

- The hints `omp_sync_hint_uncontended` and `omp_sync_hint_contended` cannot be combined.
- The hints `omp_sync_hint_nonspeculative` and `omp_sync_hint_speculative` cannot be combined.

The restrictions for combining multiple values of `omp_sync_hint` apply equally to the corresponding values of `omp_lock_hint`, and expressions that mix the two types.

Cross References

- `critical` construct, see Section [2.17.1](#) on page [223](#).
- `atomic` construct, see Section [2.17.7](#) on page [234](#)
- `omp_init_lock_with_hint` and `omp_init_nest_lock_with_hint`, see Section [3.3.2](#) on page [385](#).

1 2.18 Cancellation Constructs

2 2.18.1 `cancel` Construct

3 Summary

4 The **`cancel`** construct activates cancellation of the innermost enclosing region of the type
5 specified. The **`cancel`** construct is a stand-alone directive.

6 Syntax

C / C++

7 The syntax of the **`cancel`** construct is as follows:

8 **`#pragma omp cancel`** *construct-type-clause* [[,] *if-clause*] *new-line*

9 where *construct-type-clause* is one of the following:

```
parallel
sections
for
taskgroup
```

14 and *if-clause* is

```
if ([ cancel : ] scalar-expression)
```

C / C++

Fortran

16 The syntax of the **`cancel`** construct is as follows:

17 **`!$omp cancel`** *construct-type-clause* [[,] *if-clause*]

18 where *construct-type-clause* is one of the following:

```
parallel
sections
do
taskgroup
```

23 and *if-clause* is

```
if ([ cancel : ] scalar-logical-expression)
```

Fortran

Binding

The binding thread set of the **cancel** region is the current team. The binding region of the **cancel** region is the innermost enclosing region of the type corresponding to the *construct-type-clause* specified in the directive (that is, the innermost **parallel**, **sections**, *worksharing-loop*, or **taskgroup** region).

Description

The **cancel** construct activates cancellation of the binding region only if the *cancel-var* ICV is *true*, in which case the **cancel** construct causes the encountering task to continue execution at the end of the binding region if *construct-type-clause* is **parallel**, **for**, **do**, or **sections**. If the *cancel-var* ICV is *true* and *construct-type-clause* is **taskgroup**, the encountering task continues execution at the end of the current task region. If the *cancel-var* ICV is *false*, the **cancel** construct is ignored.

Threads check for active cancellation only at cancellation points that are implied at the following locations:

- **cancel** regions;
- **cancellation point** regions;
- **barrier** regions;
- implicit barriers regions.

When a thread reaches one of the above cancellation points and if the *cancel-var* ICV is *true*, then:

- If the thread is at a **cancel** or **cancellation point** region and *construct-type-clause* is **parallel**, **for**, **do**, or **sections**, the thread continues execution at the end of the canceled region if cancellation has been activated for the innermost enclosing region of the type specified.
- If the thread is at a **cancel** or **cancellation point** region and *construct-type-clause* is **taskgroup**, the encountering task checks for active cancellation of all of the *taskgroup sets* to which the encountering task belongs, and continues execution at the end of the current task region if cancellation has been activated for any of the *taskgroup sets*.
- If the encountering task is at a barrier region, the encountering task checks for active cancellation of the innermost enclosing **parallel** region. If cancellation has been activated, then the encountering task continues execution at the end of the canceled region.

Note – If one thread activates cancellation and another thread encounters a cancellation point, the order of execution between the two threads is non-deterministic. Whether the thread that encounters a cancellation point detects the activated cancellation depends on the underlying hardware and operating system.

When cancellation of tasks is activated through a **cancel** construct with the **taskgroup** *construct-type-clause*, the tasks that belong to the *taskgroup set* of the innermost enclosing **taskgroup** region will be canceled. The task that encountered that construct continues execution at the end of its task region, which implies completion of that task. Any task that belongs to the innermost enclosing **taskgroup** and has already begun execution must run to completion or until a cancellation point is reached. Upon reaching a cancellation point and if cancellation is active, the task continues execution at the end of its task region, which implies the task's completion. Any task that belongs to the innermost enclosing **taskgroup** and that has not begun execution may be discarded, which implies its completion.

When cancellation is active for a **parallel**, **sections**, or worksharing-loop region, each thread of the binding thread set resumes execution at the end of the canceled region if a cancellation point is encountered. If the canceled region is a **parallel** region, any tasks that have been created by a **task** or a **taskloop** construct and their descendent tasks are canceled according to the above **taskgroup** cancellation semantics. If the canceled region is a **sections**, or worksharing-loop region, no task cancellation occurs.

C++

The usual C++ rules for object destruction are followed when cancellation is performed.

C++

Fortran

All private objects or subobjects with **ALLOCATABLE** attribute that are allocated inside the canceled construct are deallocated.

Fortran

If the canceled construct contains a **reduction**, **task_reduction** or **lastprivate** clause, the final value of the list items that appeared in those clauses are undefined.

When an **if** clause is present on a **cancel** construct and the **if** expression evaluates to *false*, the **cancel** construct does not activate cancellation. The cancellation point associated with the **cancel** construct is always encountered regardless of the value of the **if** expression.

Note – The programmer is responsible for releasing locks and other synchronization data structures that might cause a deadlock when a **cancel** construct is encountered and blocked threads cannot be canceled. The programmer is also responsible for ensuring proper synchronizations to avoid deadlocks that might arise from cancellation of OpenMP regions that contain OpenMP synchronization constructs.

Execution Model Events

If a task encounters a **cancel** construct that will activate cancellation then a *cancel* event occurs.

A *discarded-task* event occurs for any discarded tasks.

Tool Callbacks

A thread dispatches a registered **ompt_callback_cancel** callback for each occurrence of a *cancel* event in the context of the encountering task. This callback has type signature **ompt_callback_cancel_t**; (*flags & ompt_cancel_activated*) always evaluates to *true* in the dispatched callback; (*flags & ompt_cancel_parallel*) evaluates to *true* in the dispatched callback if *construct-type-clause* is **parallel**; (*flags & ompt_cancel_sections*) evaluates to *true* in the dispatched callback if *construct-type-clause* is **sections**; (*flags & ompt_cancel_loop*) evaluates to *true* in the dispatched callback if *construct-type-clause* is **for** or **do**; and (*flags & ompt_cancel_taskgroup*) evaluates to *true* in the dispatched callback if *construct-type-clause* is **taskgroup**.

A thread dispatches a registered **ompt_callback_cancel** callback with the *ompt_data_t* associated with the discarded task as its *task_data* argument and **ompt_cancel_discarded_task** as its *flags* argument for each occurrence of a *discarded-task* event. The callback occurs in the context of the task that discards the task and has type signature **ompt_callback_cancel_t**.

Restrictions

The restrictions to the **cancel** construct are as follows:

- The behavior for concurrent cancellation of a region and a region nested within it is unspecified.
- If *construct-type-clause* is **taskgroup**, the **cancel** construct must be closely nested inside a **task** or a **taskloop** construct and the **cancel** region must be closely nested inside a **taskgroup** region. If *construct-type-clause* is **sections**, the **cancel** construct must be closely nested inside a **sections** or **section** construct. Otherwise, the **cancel** construct must be closely nested inside an OpenMP construct that matches the type specified in *construct-type-clause* of the **cancel** construct.
- A worksharing construct that is canceled must not have a **nowait** clause.
- A worksharing-loop construct that is canceled must not have an **ordered** clause.
- During execution of a construct that may be subject to cancellation, a thread must not encounter an orphaned cancellation point. That is, a cancellation point must only be encountered within that construct and must not be encountered elsewhere in its region.

Cross References

- *cancel-var* ICV, see Section 2.5.1 on page 64.
- **if** clause, see Section 2.15 on page 220.
- **cancellation point** construct, see Section 2.18.2 on page 267.

- `omp_get_cancellation` routine, see Section 3.2.9 on page 342.
- `omp_cancel_flag_t` enumeration type, see Section 4.4.4.24 on page 450.
- `ompt_callback_cancel_t`, see Section 4.5.2.18 on page 481.

2.18.2 cancellation point Construct

Summary

The **cancellation point** construct introduces a user-defined cancellation point at which implicit or explicit tasks check if cancellation of the innermost enclosing region of the type specified has been activated. The **cancellation point** construct is a stand-alone directive.

Syntax

C / C++

The syntax of the **cancellation point** construct is as follows:

```
#pragma omp cancellation point construct-type-clause new-line
```

where *construct-type-clause* is one of the following:

```
parallel  
sections  
for  
taskgroup
```

C / C++

Fortran

The syntax of the **cancellation point** construct is as follows:

```
!$omp cancellation point construct-type-clause
```

where *construct-type-clause* is one of the following:

```
parallel  
sections  
do  
taskgroup
```

Fortran

Binding

The binding thread set of the **cancellation point** construct is the current team. The binding region of the **cancellation point** region is the innermost enclosing region of the type corresponding to the *construct-type-clause* specified in the directive (that is, the innermost **parallel**, **sections**, worksharing-loop, or **taskgroup** region).

Description

This directive introduces a user-defined cancellation point at which an implicit or explicit task must check if cancellation of the innermost enclosing region of the type specified in the clause has been requested. This construct does not implement any synchronization between threads or tasks.

When an implicit or explicit task reaches a user-defined cancellation point and if the *cancel-var* ICV is *true*, then:

- If the *construct-type-clause* of the encountered **cancellation point** construct is **parallel**, **for**, **do**, or **sections**, the thread continues execution at the end of the canceled region if cancellation has been activated for the innermost enclosing region of the type specified.
- If the *construct-type-clause* of the encountered **cancellation point** construct is **taskgroup**, the encountering task checks for active cancellation of all *taskgroup sets* to which the encountering task belongs and continues execution at the end of the current task region if cancellation has been activated for any of them.

Execution Model Events

The *cancellation* event occurs if a task encounters a cancellation point and detected the activation of cancellation.

Tool Callbacks

A thread dispatches a registered **ompt_callback_cancel** callback for each occurrence of a *cancel* event in the context of the encountering task. This callback has type signature **ompt_callback_cancel_t; (flags & ompt_cancel_detected)** always evaluates to *true* in the dispatched callback; **(flags & ompt_cancel_parallel)** evaluates to *true* in the dispatched callback if *construct-type-clause* of the encountered **cancellation point** construct is **parallel**; **(flags & ompt_cancel_sections)** evaluates to *true* in the dispatched callback if *construct-type-clause* of the encountered **cancellation point** construct is **sections**; **(flags & ompt_cancel_loop)** evaluates to *true* in the dispatched callback if *construct-type-clause* of the encountered **cancellation point** construct is **for** or **do**; and **(flags & ompt_cancel_taskgroup)** evaluates to *true* in the dispatched callback if *construct-type-clause* of the encountered **cancellation point** construct is **taskgroup**.

Restrictions

- A **cancellation point** construct for which *construct-type-clause* is **taskgroup** must be closely nested inside a **task** or **taskloop** construct, and the **cancellation point** region must be closely nested inside a **taskgroup** region.
- A **cancellation point** construct for which *construct-type-clause* is **sections** must be closely nested inside a **sections** or **section** construct.
- A **cancellation point** construct for which *construct-type-clause* is neither **sections** nor **taskgroup** must be closely nested inside an OpenMP construct that matches the type specified in *construct-type-clause*.

Cross References

- *cancel-var* ICV, see Section 2.5.1 on page 64.
- **cancel** construct, see Section 2.18.1 on page 263.
- **omp_get_cancellation** routine, see Section 3.2.9 on page 342.
- **ompt_callback_cancel_t**, see Section 4.5.2.18 on page 481.

2.19 Data Environment

This section presents directives and clauses for controlling data environments.

2.19.1 Data-Sharing Attribute Rules

This section describes how the data-sharing attributes of variables referenced in data environments are determined. The following two cases are described separately:

- Section 2.19.1.1 on page 270 describes the data-sharing attribute rules for variables referenced in a construct.
- Section 2.19.1.2 on page 273 describes the data-sharing attribute rules for variables referenced in a region, but outside any construct.

1 2.19.1.1 Variables Referenced in a Construct

2 The data-sharing attributes of variables that are referenced in a construct can be *predetermined*,
3 *explicitly determined*, or *implicitly determined*, according to the rules outlined in this section.

4 Specifying a variable in a data-sharing attribute clause, except for the **private** clause, or
5 **copyprivate** clause of an enclosed construct causes an implicit reference to the variable in the
6 enclosing construct. Specifying a variable in a **map** clause of an enclosed construct may cause an
7 implicit reference to the variable in the enclosing construct. Such implicit references are also
8 subject to the data-sharing attribute rules outlined in this section.

9 Certain variables and objects have *predetermined* data-sharing attributes as follows:

▼ C / C++ ▼

- 10 • Variables that appear in **threadprivate** directives are threadprivate.
- 11 • Variables with automatic storage duration that are declared in a scope inside the construct are
12 private.
- 13 • Objects with dynamic storage duration are shared.
- 14 • Static data members are shared.
- 15 • The loop iteration variable(s) in the associated *for-loop(s)* of a **for**, **parallel for**,
16 **taskloop**, or **distribute** construct is (are) private.
- 17 • The loop iteration variable in the associated *for-loop* of a **simd** construct with just one
18 associated *for-loop* is linear with a *linear-step* that is the increment of the associated *for-loop*.
- 19 • The loop iteration variables in the associated *for-loops* of a **simd** construct with multiple
20 associated *for-loops* are lastprivate.
- 21 • The loop iteration variable(s) in the associated *for-loop(s)* of a **loop** construct is (are) lastprivate.
- 22 • Variables with static storage duration that are declared in a scope inside the construct are shared.
- 23 • If a list item in a **map** clause on the **target** construct has a base pointer, and the base pointer is
24 a scalar variable that does not appear in a **map** clause on the construct, the base pointer is
25 firstprivate.
- 26 • If a list item in a **reduction** or **in_reduction** clause on a construct has a base pointer then
27 the base pointer is private.

▲ C / C++ ▲

▼ Fortran ▼

- 28 • Variables and common blocks that appear in **threadprivate** directives are threadprivate.
- 29 • The loop iteration variable(s) in the associated *do-loop(s)* of a **do**, **parallel do**, **taskloop**,
30 or **distribute** construct is (are) private.

- The loop iteration variable in the associated *do-loop* of a **simd** construct with just one associated *do-loop* is linear with a *linear-step* that is the increment of the associated *do-loop*.
- The loop iteration variables in the associated *do-loops* of a **simd** construct with multiple associated *do-loops* are **lastprivate**.
- The loop iteration variable(s) in the associated *do-loop(s)* of a **loop** construct is (are) **lastprivate**.
- A loop iteration variable for a sequential loop in a **parallel** or task generating construct is private in the innermost such construct that encloses the loop.
- Implied-do indices and **forall** indices are private.
- Cray pointees have the same data-sharing attribute as the storage with which their Cray pointers are associated.
- Assumed-size arrays are shared.
- An associate name preserves the association with the selector established at the **ASSOCIATE** or **SELECT TYPE** statement.

Fortran

Variables with predetermined data-sharing attributes may not be listed in data-sharing attribute clauses, except for the cases listed below. For these exceptions only, listing a predetermined variable in a data-sharing attribute clause is allowed and overrides the variable's predetermined data-sharing attributes.

C / C++

- The loop iteration variable(s) in the associated *for-loop(s)* of a **for**, **parallel for**, **taskloop**, **distribute**, or **loop** construct may be listed in a **private** or **lastprivate** clause.
- The loop iteration variable in the associated *for-loop* of a **simd** construct with just one associated *for-loop* may be listed in a **private**, **lastprivate**, or **linear** clause with a *linear-step* that is the increment of the associated *for-loop*.
- The loop iteration variables in the associated *for-loops* of a **simd** construct with multiple associated *for-loops* may be listed in a **private** or **lastprivate** clause.
- Variables with **const**-qualified type with no mutable members may be listed in a **firstprivate** clause, even if they are static data members.

C / C++

Fortran

- The loop iteration variable(s) in the associated *do-loop(s)* of a **do**, **parallel do**, **taskloop**, **distribute**, or **loop** construct may be listed in a **private** or **lastprivate** clause.
- The loop iteration variable in the associated *do-loop* of a **simd** construct with just one associated *do-loop* may be listed in a **private**, **lastprivate**, or **linear** clause with a *linear-step* that is the increment of the associated loop.
- The loop iteration variables in the associated *do-loops* of a **simd** construct with multiple associated *do-loops* may be listed in a **private** or **lastprivate** clause.
- Variables used as loop iteration variables in sequential loops in a **parallel** or task generating construct may be listed in data-sharing attribute clauses on the construct itself, and on enclosed constructs, subject to other restrictions.
- Assumed-size arrays may be listed in a **shared** clause.

Fortran

Additional restrictions on the variables that may appear in individual clauses are described with each clause in Section 2.19.4 on page 282.

Variables with *explicitly determined* data-sharing attributes are those that are referenced in a given construct and are listed in a data-sharing attribute clause on the construct.

Variables with *implicitly determined* data-sharing attributes are those that are referenced in a given construct, do not have predetermined data-sharing attributes, and are not listed in a data-sharing attribute clause on the construct.

Rules for variables with *implicitly determined* data-sharing attributes are as follows:

- In a **parallel**, **teams**, or task generating construct, the data-sharing attributes of these variables are determined by the **default** clause, if present (see Section 2.19.4.1 on page 282).
- In a **parallel** construct, if no **default** clause is present, these variables are shared.
- For constructs other than task generating constructs, if no **default** clause is present, these variables reference the variables with the same names that exist in the enclosing context.
- In a **target** construct, variables that are not mapped after applying data-mapping attribute rules (see Section 2.19.7 on page 314) are firstprivate.

C++

- In an orphaned task generating construct, if no **default** clause is present, formal arguments passed by reference are firstprivate.

C++

Fortran

- In an orphaned task generating construct, if no **default** clause is present, dummy arguments are firstprivate.

Fortran

- In a task generating construct, if no **default** clause is present, a variable for which the data-sharing attribute is not determined by the rules above and that in the enclosing context is determined to be shared by all implicit tasks bound to the current team is shared.
- In a task generating construct, if no **default** clause is present, a variable for which the data-sharing attribute is not determined by the rules above is **firstprivate**.

Additional restrictions on the variables for which data-sharing attributes cannot be implicitly determined in a task generating construct are described in Section 2.19.4.4 on page 286.

2.19.1.2 Variables Referenced in a Region but not in a Construct

The data-sharing attributes of variables that are referenced in a region, but not in a construct, are determined as follows:

C / C++

- Variables with static storage duration that are declared in called routines in the region are shared.
- File-scope or namespace-scope variables referenced in called routines in the region are shared unless they appear in a **threadprivate** directive.
- Objects with dynamic storage duration are shared.
- Static data members are shared unless they appear in a **threadprivate** directive.
- In C++, formal arguments of called routines in the region that are passed by reference have the same data-sharing attributes as the associated actual arguments.
- Other variables declared in called routines in the region are private.

C / C++

Fortran

- Local variables declared in called routines in the region and that have the **save** attribute, or that are data initialized, are shared unless they appear in a **threadprivate** directive.
- Variables belonging to common blocks, or accessed by host or use association, and referenced in called routines in the region are shared unless they appear in a **threadprivate** directive.
- Dummy arguments of called routines in the region that have the **VALUE** attribute are private.
- Dummy arguments of called routines in the region that do not have the **VALUE** attribute are private if the associated actual argument is not shared.
- Dummy arguments of called routines in the region that do not have the **VALUE** attribute are shared if the actual argument is shared and it is a scalar variable, structure, an array that is not a pointer or assumed-shape array, or a simply contiguous array section. Otherwise, the data-sharing attribute of the dummy argument is implementation-defined if the associated actual argument is shared.

- Cray pointees have the same data-sharing attribute as the storage with which their Cray pointers are associated.
- Implied-do indices, **forall** indices, and other local variables declared in called routines in the region are private.

Fortran

2.19.2 threadprivate Directive

Summary

The **threadprivate** directive specifies that variables are replicated, with each thread having its own copy. The **threadprivate** directive is a declarative directive.

Syntax

C / C++

The syntax of the **threadprivate** directive is as follows:

```
#pragma omp threadprivate(list) new-line
```

where *list* is a comma-separated list of file-scope, namespace-scope, or static block-scope variables that do not have incomplete types.

C / C++

Fortran

The syntax of the **threadprivate** directive is as follows:

```
!$omp threadprivate(list)
```

where *list* is a comma-separated list of named variables and named common blocks. Common block names must appear between slashes.

Fortran

Description

Each copy of a threadprivate variable is initialized once, in the manner specified by the program, but at an unspecified point in the program prior to the first reference to that copy. The storage of all copies of a threadprivate variable is freed according to how static variables are handled in the base language, but at an unspecified point in the program.

A program in which a thread references another thread's copy of a threadprivate variable is non-conforming.

The content of a threadprivate variable can change across a task scheduling point if the executing thread switches to another task that modifies the variable. For more details on task scheduling, see Section 1.3 on page 20 and Section 2.10 on page 135.

In **parallel** regions, references by the master thread will be to the copy of the variable in the thread that encountered the **parallel** region.

During a sequential part references will be to the initial thread's copy of the variable. The values of data in the initial thread's copy of a threadprivate variable are guaranteed to persist between any two consecutive references to the variable in the program provided that no **teams** construct that is not nested inside of a **target** construct is encountered between the references and that the initial thread is not nested inside of a **teams** region. For initial threads nested inside of a **teams** region, the values of data in the copies of a threadprivate variable of those initial threads are guaranteed to persist between any two consecutive references to the variable inside of that **teams** region.

The values of data in the threadprivate variables of threads that are not initial threads are guaranteed to persist between two consecutive active **parallel** regions only if all of the following conditions hold:

- Neither **parallel** region is nested inside another explicit **parallel** region;
- The number of threads used to execute both **parallel** regions is the same;
- The thread affinity policies used to execute both **parallel** regions are the same;
- The value of the *dyn-var* internal control variable in the enclosing task region is *false* at entry to both **parallel** regions; and
- No **teams** construct that is not nested inside of a **target** construct is encountered between both **parallel** regions.
- Neither the **omp_pause_resource** nor **omp_pause_resource_all** routine is called.

If these conditions all hold, and if a threadprivate variable is referenced in both regions, then threads with the same thread number in their respective regions will reference the same copy of that variable.

C / C++

If the above conditions hold, the storage duration, lifetime, and value of a thread's copy of a threadprivate variable that does not appear in any **copyin** clause on the second region will be retained. Otherwise, the storage duration, lifetime, and value of a thread's copy of the variable in the second region is unspecified.

C / C++

Fortran

If the above conditions hold, the definition, association, or allocation status of a thread's copy of a threadprivate variable or a variable in a threadprivate common block that is not affected by any **copyin** clause that appears on the second region (a variable is affected by a **copyin** clause if the variable appears in the **copyin** clause or it is in a common block that appears in the **copyin** clause) will be retained. Otherwise, the definition and association status of a thread's copy of the variable in the second region are undefined, and the allocation status of an allocatable variable will be implementation defined.

If a threadprivate variable or a variable in a threadprivate common block is not affected by any **copyin** clause that appears on the first **parallel** region in which it is referenced, the thread's copy of the variable inherits the declared type parameter and the default parameter values from the original variable. The variable or any subobject of the variable is initially defined or undefined according to the following rules:

- If it has the **ALLOCATABLE** attribute, each copy created will have an initial allocation status of unallocated;
- If it has the **POINTER** attribute:
 - If it has an initial association status of disassociated, either through explicit initialization or default initialization, each copy created will have an association status of disassociated;
 - Otherwise, each copy created will have an association status of undefined.
- If it does not have either the **POINTER** or the **ALLOCATABLE** attribute:
 - If it is initially defined, either through explicit initialization or default initialization, each copy created is so defined;
 - Otherwise, each copy created is undefined.

Fortran

C / C++

The address of a threadprivate variable is not an address constant.

C / C++

C++

The order in which any constructors for different threadprivate variables of class type are called is unspecified. The order in which any destructors for different threadprivate variables of class type are called is unspecified.

C++

Restrictions

The restrictions to the **threadprivate** directive are as follows:

- A **threadprivate** variable must not appear in any clause except the **copyin**, **copyprivate**, **schedule**, **num_threads**, **thread_limit**, and **if** clauses.
- A program in which an untied task accesses **threadprivate** storage is non-conforming.

C / C++

- If the value of a variable referenced in an explicit initializer of a **threadprivate** variable is modified prior to the first reference to any instance of the **threadprivate** variable, then the behavior is unspecified.
- A variable that is part of another variable (as an array or structure element) cannot appear in a **threadprivate** clause unless it is a static data member of a C++ class.
- A **threadprivate** directive for file-scope variables must appear outside any definition or declaration, and must lexically precede all references to any of the variables in its list.
- A **threadprivate** directive for namespace-scope variables must appear outside any definition or declaration other than the namespace definition itself, and must lexically precede all references to any of the variables in its list.
- Each variable in the list of a **threadprivate** directive at file, namespace, or class scope must refer to a variable declaration at file, namespace, or class scope that lexically precedes the directive.
- A **threadprivate** directive for static block-scope variables must appear in the scope of the variable and not in a nested scope. The directive must lexically precede all references to any of the variables in its list.
- Each variable in the list of a **threadprivate** directive in block scope must refer to a variable declaration in the same scope that lexically precedes the directive. The variable declaration must use the static storage-class specifier.
- If a variable is specified in a **threadprivate** directive in one translation unit, it must be specified in a **threadprivate** directive in every translation unit in which it is declared.

C / C++

C++

- A **threadprivate** directive for static class member variables must appear in the class definition, in the same scope in which the member variables are declared, and must lexically precede all references to any of the variables in its list.
- A **threadprivate** variable must not have an incomplete type or a reference type.
- A **threadprivate** variable with class type must have:
 - An accessible, unambiguous default constructor in the case of default initialization without a given initializer;

- An accessible, unambiguous constructor that accepts the given argument in the case of direct initialization; and
- An accessible, unambiguous copy constructor in the case of copy initialization with an explicit initializer.

C++

Fortran

- A variable that is part of another variable (as an array, structure element or type parameter inquiry) cannot appear in a **threadprivate** clause.
- The **threadprivate** directive must appear in the declaration section of a scoping unit in which the common block or variable is declared.
- If a **threadprivate** directive that specifies a common block name appears in one program unit, then such a directive must also appear in every other program unit that contains a **COMMON** statement that specifies the same name. It must appear after the last such **COMMON** statement in the program unit.
- If a threadprivate variable or a threadprivate common block is declared with the **BIND** attribute, the corresponding C entities must also be specified in a **threadprivate** directive in the C program.
- A blank common block cannot appear in a **threadprivate** directive.
- A variable can only appear in a **threadprivate** directive in the scope in which it is declared. It must not be an element of a common block or appear in an **EQUIVALENCE** statement.
- A variable that appears in a **threadprivate** directive must be declared in the scope of a module or have the **SAVE** attribute, either explicitly or implicitly.

Fortran

Cross References

- *dyn-var* ICV, see Section 2.5 on page 63.
- Number of threads used to execute a **parallel** region, see Section 2.6.1 on page 78.
- **copyin** clause, see Section 2.19.6.1 on page 310.

1 2.19.3 List Item Privatization

2 For any construct, a list item that appears in a data-sharing attribute clause, including a reduction
3 clause, may be privatized. Each task that references a privatized list item in any statement in the
4 construct receives at least one new list item if the construct has one or more associated loops, and
5 otherwise each such task receives one new list item. Each SIMD lane used in a **simd** construct that
6 references a privatized list item in any statement in the construct receives at least one new list item.
7 Language-specific attributes for new list items are derived from the corresponding original list item.
8 Inside the construct, all references to the original list item are replaced by references to a new list
9 item received by the task or SIMD lane.

10 If the construct has one or more associated loops, within the same logical iteration of the loop(s)
11 the same new list item replaces all references to the original list item. For any two logical iterations,
12 if the references to the original list item are replaced by the same list item then the logical iterations
13 must execute in some sequential order.

14 In the rest of the region, it is unspecified whether references are to a new list item or the original list
15 item. Therefore, if an attempt is made to reference the original item, its value after the region is also
16 unspecified. If a task or a SIMD lane does not reference a privatized list item, it is unspecified
17 whether the task or SIMD lane receives a new list item.

18 The value and/or allocation status of the original list item will change only:

- 19
- If accessed and modified via pointer;
 - 20 • If possibly accessed in the region but outside of the construct;
 - 21 • As a side effect of directives or clauses; or

22

▼ Fortran ▲

- If accessed and modified via construct association.

▲ Fortran ▲

▼ C++ ▼

23 If the construct is contained in a member function, it is unspecified anywhere in the region if
24 accesses through the implicit **this** pointer refer to the new list item or the original list item.

▲ C++ ▲

▼ C / C++ ▼

25 A new list item of the same type, with automatic storage duration, is allocated for the construct.
26 The storage and thus lifetime of these list items last until the block in which they are created exits.
27 The size and alignment of the new list item are determined by the type of the variable. This
28 allocation occurs once for each task generated by the construct and once for each SIMD lane used
29 by the construct.

30 The new list item is initialized, or has an undefined initial value, as if it had been locally declared
31 without an initializer.

▲ C / C++ ▲

C++

1 If the type of a list item is a reference to a type T then the type will be considered to be T for all
2 purposes of this clause.

3 The order in which any default constructors for different private variables of class type are called is
4 unspecified. The order in which any destructors for different private variables of class type are
5 called is unspecified.

C++

Fortran

6 If any statement of the construct references a list item, a new list item of the same type and type
7 parameters is allocated. This allocation occurs once for each task generated by the construct and
8 once for each SIMD lane used by the construct. The initial value of the new list item is undefined.
9 The initial status of a private pointer is undefined.

10 For a list item or the subobject of a list item with the **ALLOCATABLE** attribute:

- 11 • If the allocation status is unallocated, the new list item or the subobject of the new list item will
12 have an initial allocation status of unallocated;
- 13 • If the allocation status is allocated, the new list item or the subobject of the new list item will
14 have an initial allocation status of allocated; and
- 15 • If the new list item or the subobject of the new list item is an array, its bounds will be the same as
16 those of the original list item or the subobject of the original list item.

17 A privatized list item may be storage-associated with other variables when the data-sharing
18 attribute clause is encountered. Storage association may exist because of constructs such as
19 **EQUIVALENCE** or **COMMON**. If A is a variable that is privatized by a construct and B is a variable
20 that is storage-associated with A , then:

- 21 • The contents, allocation, and association status of B are undefined on entry to the region;
- 22 • Any definition of A , or of its allocation or association status, causes the contents, allocation, and
23 association status of B to become undefined; and
- 24 • Any definition of B , or of its allocation or association status, causes the contents, allocation, and
25 association status of A to become undefined.

26 A privatized list item clause may be a selector of an **ASSOCIATE** or **SELECT TYPE** construct. If
27 the construct association is established prior to a **parallel** region, the association between the
28 associate name and the original list item will be retained in the region.

29 Finalization of a list item of a finalizable type or subobjects of a list item of a finalizable type
30 occurs at the end of the region. The order in which any final subroutines for different variables of a
31 finalizable type are called is unspecified.

Fortran

If a list item appears in both **firstprivate** and **lastprivate** clauses, the update required for the **lastprivate** clause occurs after all initializations for the **firstprivate** clause.

Restrictions

The following restrictions apply to any list item that is privatized unless otherwise stated for a given data-sharing attribute clause:

- C

▼

▲

 - A variable that is part of another variable (as an array or structure element) cannot be privatized.
- C++

▼

▲

 - A variable that is part of another variable (as an array or structure element) cannot be privatized except if the data-sharing attribute clause is associated with a construct within a class non-static member function and the variable is an accessible data member of the object for which the non-static member function is invoked.
 - A variable of class type (or array thereof) that is privatized requires an accessible, unambiguous default constructor for the class type.
- C / C++

▼

▲

 - A variable that is privatized must not have a **const**-qualified type unless it is of class type with a **mutable** member. This restriction does not apply to the **firstprivate** clause.
 - A variable that is privatized must not have an incomplete type or be a reference to an incomplete type.
- Fortran

▼

▲

 - A variable that is part of another variable (as an array or structure element) cannot be privatized.
 - A variable that is privatized must either be definable, or an allocatable variable. This restriction does not apply to the **firstprivate** clause.
 - Variables that appear in namelist statements, in variable format expressions, and in expressions for statement function definitions, may not be privatized.
 - Pointers with the **INTENT (IN)** attribute may not be privatized. This restriction does not apply to the **firstprivate** clause.
 - Assumed-size arrays may not be privatized in a **target**, **teams**, or **distribute** construct.

1 **2.19.4 Data-Sharing Attribute Clauses**

2 Several constructs accept clauses that allow a user to control the data-sharing attributes of variables
3 referenced in the construct. Not all of the clauses listed in this section are valid on all directives.
4 The set of clauses that is valid on a particular directive is described with the directive.

5 Most of the clauses accept a comma-separated list of list items (see Section 2.1 on page 38). All list
6 items that appear in a clause must be visible, according to the scoping rules of the base language.
7 With the exception of the **default** clause, clauses may be repeated as needed. A list item may not
8 appear in more than one clause on the same directive, except that it may be specified in both
9 **firstprivate** and **lastprivate** clauses.

10 The reduction data-sharing attribute clauses are explained in Section 2.19.5 on page 293.

▼ C++ ▼

11 If a variable referenced in a data-sharing attribute clause has a type derived from a template, and
12 the program does not otherwise reference that variable then any behavior related to that variable is
13 unspecified.

▲ C++ ▲

▼ Fortran ▼

14 When a named common block appears in a **private**, **firstprivate**, **lastprivate**, or
15 **shared** clause of a directive, none of its members may be declared in another data-sharing
16 attribute clause in that directive. When individual members of a common block appear in a
17 **private**, **firstprivate**, **lastprivate**, **reduction**, or **linear** clause of a directive,
18 the storage of the specified variables is no longer Fortran associated with the storage of the common
19 block itself.

▲ Fortran ▲

20 **2.19.4.1 default Clause**

21 **Summary**

22 The **default** clause explicitly determines the data-sharing attributes of variables that are
23 referenced in a **parallel**, **teams**, or task generating construct and would otherwise be implicitly
24 determined (see Section 2.19.1.1 on page 270).

Syntax

C / C++

The syntax of the **default** clause is as follows:

```
default (shared | none)
```

C / C++

Fortran

The syntax of the **default** clause is as follows:

```
default (private | firstprivate | shared | none)
```

Fortran

Description

The **default (shared)** clause causes all variables referenced in the construct that have implicitly determined data-sharing attributes to be shared.

Fortran

The **default (firstprivate)** clause causes all variables in the construct that have implicitly determined data-sharing attributes to be firstprivate.

The **default (private)** clause causes all variables referenced in the construct that have implicitly determined data-sharing attributes to be private.

Fortran

The **default (none)** clause requires that each variable that is referenced in the construct, and that does not have a predetermined data-sharing attribute, must have its data-sharing attribute explicitly determined by being listed in a data-sharing attribute clause.

Restrictions

The restrictions to the **default** clause are as follows:

- Only a single **default** clause may be specified on a **parallel**, **task**, **taskloop** or **teams** directive.

2.19.4.2 shared Clause

Summary

The **shared** clause declares one or more list items to be shared by tasks generated by a **parallel**, **teams**, or task generating construct.

Syntax

The syntax of the **shared** clause is as follows:

shared (*list*)

Description

All references to a list item within a task refer to the storage area of the original variable at the point the directive was encountered.

The programmer must ensure, by adding proper synchronization, that storage shared by an explicit task region does not reach the end of its lifetime before the explicit task region completes its execution.

Fortran

The association status of a shared pointer becomes undefined upon entry to and exit from the **parallel**, **teams**, or task generating construct if it is associated with a target or a subobject of a target that appears as a privatized list item in a data-sharing attribute clause on the construct.

Note – Passing a shared variable to a procedure may result in the use of temporary storage in place of the actual argument when the corresponding dummy argument does not have the **VALUE** or **CONTIGUOUS** attribute and its data-sharing attribute is implementation-defined as per the rules in Section 2.19.1.2 on page 273. These conditions effectively result in references to, and definitions of, the temporary storage during the procedure reference. Furthermore, the value of the shared variable is copied into the intervening temporary storage before the procedure reference when the dummy argument does not have the **INTENT (OUT)** attribute, and is copied out of the temporary storage into the shared variable when the dummy argument does not have the **INTENT (IN)** attribute. Any references to (or definitions of) the shared storage that is associated with the dummy argument by any other task must be synchronized with the procedure reference to avoid possible data races.

Fortran

Restrictions

The restrictions for the **shared** clause are as follows:

- A variable that is part of another variable (as an array or structure element) cannot appear in a **shared** clause.

C++

- A variable that is part of another variable (as an array or structure element) cannot appear in a **shared** clause except if the **shared** clause is associated with a construct within a class non-static member function and the variable is an accessible data member of the object for which the non-static member function is invoked.

C++

Fortran

- A variable that is part of another variable (as an array, structure element or type parameter inquiry) cannot appear in a **shared** clause.

Fortran

2.19.4.3 private Clause

Summary

The **private** clause declares one or more list items to be private to a task or to a SIMD lane.

Syntax

The syntax of the **private** clause is as follows:

private (*list*)

Description

The **private** clause specifies that its list items are to be privatized according to Section 2.19.3 on page 279. Each task or SIMD lane that references a list item in the construct receives only one new list item, unless the construct has one or more associated loops and the **order (concurrent)** clause is also present.

List items that appear in a **private**, **firstprivate**, or **reduction** clause in a **parallel** construct may also appear in a **private** clause in an enclosed **parallel**, worksharing, **loop**, **task**, **taskloop**, **simd**, or **target** construct.

List items that appear in a **private** or **firstprivate** clause in a **task** or **taskloop** construct may also appear in a **private** clause in an enclosed **parallel**, **loop**, **task**, **taskloop**, **simd**, or **target** construct.

List items that appear in a **private**, **firstprivate**, **lastprivate**, or **reduction** clause in a worksharing construct may also appear in a **private** clause in an enclosed **parallel**, **loop**, **task**, **simd**, or **target** construct.

List items that appear in a **private** clause on a **loop** construct may also appear in a **private** clause in an enclosed **loop**, **parallel**, or **simd** construct.

Restrictions

The restrictions to the **private** clause are as specified in Section 2.19.3.

Cross References

- List Item Privatization, see Section 2.19.3 on page 279.

2.19.4.4 **firstprivate** Clause

Summary

The **firstprivate** clause declares one or more list items to be private to a task, and initializes each of them with the value that the corresponding original item has when the construct is encountered.

Syntax

The syntax of the **firstprivate** clause is as follows:

```
firstprivate (list)
```

Description

The **firstprivate** clause provides a superset of the functionality provided by the **private** clause.

A list item that appears in a **firstprivate** clause is subject to the **private** clause semantics described in Section 2.19.4.3 on page 285, except as noted. In addition, the new list item is initialized from the original list item existing before the construct. The initialization of the new list item is done once for each task that references the list item in any statement in the construct. The initialization is done prior to the execution of the construct.

For a **firstprivate** clause on a **parallel**, **task**, **taskloop**, **target**, or **teams** construct, the initial value of the new list item is the value of the original list item that exists immediately prior to the construct in the task region where the construct is encountered unless otherwise specified. For a **firstprivate** clause on a worksharing construct, the initial value of the new list item for each implicit task of the threads that execute the worksharing construct is the value of the original list item that exists in the implicit task immediately prior to the point in time that the worksharing construct is encountered unless otherwise specified.

To avoid data races, concurrent updates of the original list item must be synchronized with the read of the original list item that occurs as a result of the **firstprivate** clause.

C / C++

For variables of non-array type, the initialization occurs by copy assignment. For an array of elements of non-array type, each element is initialized as if by assignment from an element of the original array to the corresponding element of the new array.

C / C++

C++

For each variable of class type:

- If the **firstprivate** clause is not on a **target** construct then a copy constructor is invoked to perform the initialization; and
- If the **firstprivate** clause is on a **target** construct then it is unspecified how many copy constructors, if any, are invoked.

If copy constructors are called, the order in which copy constructors for different variables of class type are called is unspecified.

C++

Fortran

If the original list item does not have the **POINTER** attribute, initialization of the new list items occurs as if by intrinsic assignment unless the list item has a type bound procedure as a defined assignment. If the original list item that does not have the **POINTER** attribute has the allocation status of unallocated, the new list items will have the same status.

If the original list item has the **POINTER** attribute, the new list items receive the same association status of the original list item as if by pointer assignment.

Fortran

Restrictions

The restrictions to the **firstprivate** clause are as follows:

- A list item that is private within a **parallel** region must not appear in a **firstprivate** clause on a worksharing construct if any of the worksharing regions arising from the worksharing construct ever bind to any of the **parallel** regions arising from the **parallel** construct.
- A list item that is private within a **teams** region must not appear in a **firstprivate** clause on a **distribute** construct if any of the **distribute** regions arising from the **distribute** construct ever bind to any of the **teams** regions arising from the **teams** construct.
- A list item that appears in a **reduction** clause of a **parallel** construct must not appear in a **firstprivate** clause on a worksharing, **task**, or **taskloop** construct if any of the worksharing or **task** regions arising from the worksharing, **task**, or **taskloop** construct ever bind to any of the **parallel** regions arising from the **parallel** construct.

- A list item that appears in a **reduction** clause of a **teams** construct must not appear in a **firstprivate** clause on a **distribute** construct if any of the **distribute** regions arising from the **distribute** construct ever bind to any of the **teams** regions arising from the **teams** construct.
- A list item that appears in a **reduction** clause of a worksharing construct must not appear in a **firstprivate** clause in a **task** construct encountered during execution of any of the worksharing regions arising from the worksharing construct.

C++

- A variable of class type (or array thereof) that appears in a **firstprivate** clause requires an accessible, unambiguous copy constructor for the class type.

C++

C / C++

- If a list item in a **firstprivate** clause on a worksharing construct has a reference type then it must bind to the same object for all threads of the team.

C / C++

Fortran

- If the list item is a polymorphic variable with the **ALLOCATABLE** attribute, the behavior is unspecified.

Fortran

2.19.4.5 lastprivate Clause

Summary

The **lastprivate** clause declares one or more list items to be private to an implicit task or to a SIMD lane, and causes the corresponding original list item to be updated after the end of the region.

Syntax

The syntax of the **lastprivate** clause is as follows:

```
lastprivate ([ lastprivate-modifier : ] list)
```

where *lastprivate-modifier* is:

```
conditional
```

Description

The **lastprivate** clause provides a superset of the functionality provided by the **private** clause.

A list item that appears in a **lastprivate** clause is subject to the **private** clause semantics described in Section 2.19.4.3 on page 285. In addition, when a **lastprivate** clause without the **conditional** modifier appears on a directive, the value of each new list item from the sequentially last iteration of the associated loops, or the lexically last **section** construct, is assigned to the original list item. When the **conditional** modifier appears on the clause, if an assignment to a list item is encountered in the construct then the original list item is assigned the value that is assigned to the new list item in the sequentially last iteration or lexically last section in which such an assignment is encountered.

C / C++

For an array of elements of non-array type, each element is assigned to the corresponding element of the original array.

C / C++

Fortran

If the original list item does not have the **POINTER** attribute, its update occurs as if by intrinsic assignment unless it has a type bound procedure as a defined assignment.

If the original list item has the **POINTER** attribute, its update occurs as if by pointer assignment.

Fortran

When the **conditional** modifier does not appear on the **lastprivate** clause, list items that are not assigned a value by the sequentially last iteration of the loops, or by the lexically last **section** construct, have unspecified values after the construct. Unassigned subcomponents also have unspecified values after the construct.

If the **lastprivate** clause is used on a construct to which neither the **nowait** nor the **nogroup** clauses are applied, the original list item becomes defined at the end of the construct. To avoid data races, concurrent reads or updates of the original list item must be synchronized with the update of the original list item that occurs as a result of the **lastprivate** clause.

Otherwise, If the **lastprivate** clause is used on a construct to which the **nowait** or the **nogroup** clauses are applied, accesses to the original list item may create a data race. To avoid this data race, if an assignment to the original list item occurs then synchronization must be inserted to ensure that the assignment completes and the original list item is flushed to memory.

If a list item that appears in a **lastprivate** clause with the **conditional** modifier is modified in the region by an assignment outside the construct or not to the list item then the value assigned to the original list item is unspecified.

Restrictions

The restrictions to the **lastprivate** clause are as follows:

- A list item that is private within a **parallel** region, or that appears in the **reduction** clause of a **parallel** construct, must not appear in a **lastprivate** clause on a worksharing construct if any of the corresponding worksharing regions ever binds to any of the corresponding **parallel** regions.
- A list item that appears in a **lastprivate** clause with the **conditional** modifier must be a scalar variable.

C++

- A variable of class type (or array thereof) that appears in a **lastprivate** clause requires an accessible, unambiguous default constructor for the class type, unless the list item is also specified in a **firstprivate** clause.
- A variable of class type (or array thereof) that appears in a **lastprivate** clause requires an accessible, unambiguous copy assignment operator for the class type. The order in which copy assignment operators for different variables of class type are called is unspecified.

C++

C / C++

- If a list item in a **lastprivate** clause on a worksharing construct has a reference type then it must bind to the same object for all threads of the team.

C / C++

Fortran

- A variable that appears in a **lastprivate** clause must be definable.
- If the original list item has the **ALLOCATABLE** attribute, the corresponding list item whose value is assigned to the original list item must have an allocation status of allocated upon exit from the sequentially last iteration or lexically last **section** construct.
- If the list item is a polymorphic variable with the **ALLOCATABLE** attribute, the behavior is unspecified.

Fortran

2.19.4.6 linear Clause

Summary

The **linear** clause declares one or more list items to be private and to have a linear relationship with respect to the iteration space of a loop associated with the construct on which the clause appears.

1 **Syntax**

C

2 The syntax of the **linear** clause is as follows:

3 **linear** (*linear-list* [: *linear-step*])

4 where *linear-list* is one of the following

5 *list*
6 *modifier* (*list*)

7 where *modifier* is one of the following:

8 **val**

C

C++

9 The syntax of the **linear** clause is as follows:

10 **linear** (*linear-list* [: *linear-step*])

11 where *linear-list* is one of the following

12 *list*
13 *modifier* (*list*)

14 where *modifier* is one of the following:

15 **ref**
16 **val**
17 **uval**

C++

Fortran

18 The syntax of the **linear** clause is as follows:

19 **linear** (*linear-list* [: *linear-step*])

20 where *linear-list* is one of the following

21 *list*
22 *modifier* (*list*)

where *modifier* is one of the following:

```
ref
val
uval
```

Fortran

Description

The **linear** clause provides a superset of the functionality provided by the **private** clause. A list item that appears in a **linear** clause is subject to the **private** clause semantics described in Section 2.19.4.3 on page 285 except as noted. If *linear-step* is not specified, it is assumed to be 1.

When a **linear** clause is specified on a construct, the value of the new list item on each iteration of the associated loop(s) corresponds to the value of the original list item before entering the construct plus the logical number of the iteration times *linear-step*. The value corresponding to the sequentially last iteration of the associated loop(s) is assigned to the original list item.

When a **linear** clause is specified on a declarative directive, all list items must be formal parameters (or, in Fortran, dummy arguments) of a function that will be invoked concurrently on each SIMD lane. If no *modifier* is specified or the **val** or **uval** modifier is specified, the value of each list item on each lane corresponds to the value of the list item upon entry to the function plus the logical number of the lane times *linear-step*. If the **uval** modifier is specified, each invocation uses the same storage location for each SIMD lane; this storage location is updated with the final value of the logically last lane. If the **ref** modifier is specified, the storage location of each list item on each lane corresponds to an array at the storage location upon entry to the function indexed by the logical number of the lane times *linear-step*.

Restrictions

- The *linear-step* expression must be invariant during the execution of the region that corresponds to the construct. Otherwise, the execution results in unspecified behavior.
- Only a loop iteration variable of a loop that is associated with the construct may appear as a *list-item* in a **linear** clause if a **reduction** clause with the **inscan** modifier also appears on the construct.

C

- A *list-item* that appears in a **linear** clause must be of integral or pointer type.

C

C++

- A *list-item* that appears in a **linear** clause without the **ref** modifier must be of integral or pointer type, or must be a reference to an integral or pointer type.
- The **ref** or **uval** modifier can only be used if the *list-item* is of a reference type.
- If a list item in a **linear** clause on a worksharing construct has a reference type then it must bind to the same object for all threads of the team.
- If the list item is of a reference type and the **ref** modifier is not specified and if any write to the list item occurs before any read of the list item then the result is unspecified.

C++

Fortran

- A *list-item* that appears in a **linear** clause without the **ref** modifier must be of type **integer**.
- The **ref** or **uval** modifier can only be used if the *list-item* is a dummy argument without the **VALUE** attribute.
- Variables that have the **POINTER** attribute and Cray pointers may not appear in a **linear** clause.
- If the list item has the **ALLOCATABLE** attribute and the **ref** modifier is not specified, the allocation status of the list item in the sequentially last iteration must be allocated upon exit from that iteration.
- If the **ref** modifier is specified, variables with the **ALLOCATABLE** attribute, assumed-shape arrays and polymorphic variables may not appear in the **linear** clause.
- If the list item is a dummy argument without the **VALUE** attribute and the **ref** modifier is not specified and if any write to the list item occurs before any read of the list item then the result is unspecified.
- A common block name cannot appear in a **linear** clause.

Fortran


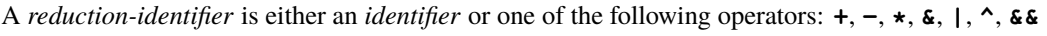
2.19.5 Reduction Clauses and Directives


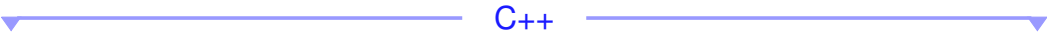
The reduction clauses are data-sharing attribute clauses that can be used to perform some forms of recurrence calculations in parallel. Reduction clauses include reduction scoping clauses and reduction participating clauses. Reduction scoping clauses define the region in which a reduction is computed. Reduction participating clauses define the participants in the reduction. Reduction clauses specify a *reduction-identifier* and one or more list items.

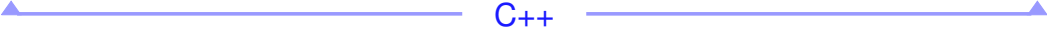

1 **2.19.5.1 Properties Common To All Reduction Clauses**

2 **Syntax**

3 The syntax of a *reduction-identifier* is defined as follows:

4  **C** 
5 A *reduction-identifier* is either an *identifier* or one of the following operators: +, −, *, &, |, ^, && and ||.

6  **C++** 
7 A *reduction-identifier* is either an *id-expression* or one of the following operators: +, −, *, &, |, ^, && and ||.

8  **Fortran** 
9 A *reduction-identifier* is either a base language identifier, or a user-defined operator, or one of the
10 following operators: +, −, *, .and., .or., .eqv., .neqv., or one of the following intrinsic
procedure names: **max**, **min**, **iand**, **ior**, **ieor**.

11  **Fortran** 
12  **C / C++** 

13 Table 2.11 lists each *reduction-identifier* that is implicitly declared at every scope for arithmetic types and its semantic initializer value. The actual initializer value is that value as expressed in the data type of the reduction list item.

TABLE 2.11: Implicitly Declared C/C++ *reduction-identifiers*

Identifier	Initializer	Combiner
+	omp_priv = 0	omp_out += omp_in
−	omp_priv = 0	omp_out += omp_in
*	omp_priv = 1	omp_out *= omp_in
&	omp_priv = ~ 0	omp_out &= omp_in
	omp_priv = 0	omp_out = omp_in
^	omp_priv = 0	omp_out ^= omp_in
&&	omp_priv = 1	omp_out = omp_in && omp_out

table continued on next page

table continued from previous page

Identifier	Initializer	Combiner
<code> </code>	<code>omp_priv = 0</code>	<code>omp_out = omp_in omp_out</code>
<code>max</code>	<code>omp_priv = Least</code> <i>representable number in the</i> <i>reduction list item type</i>	<code>omp_out = omp_in > omp_out ?</code> <code>omp_in : omp_out</code>
<code>min</code>	<code>omp_priv = Largest</code> <i>representable number in the</i> <i>reduction list item type</i>	<code>omp_out = omp_in < omp_out ?</code> <code>omp_in : omp_out</code>



1 Table 2.12 lists each *reduction-identifier* that is implicitly declared for numeric and logical types
2 and its semantic initializer value. The actual initializer value is that value as expressed in the data
3 type of the reduction list item.

TABLE 2.12: Implicitly Declared Fortran *reduction-identifiers*

Identifier	Initializer	Combiner
<code>+</code>	<code>omp_priv = 0</code>	<code>omp_out = omp_in + omp_out</code>
<code>-</code>	<code>omp_priv = 0</code>	<code>omp_out = omp_in + omp_out</code>
<code>*</code>	<code>omp_priv = 1</code>	<code>omp_out = omp_in * omp_out</code>
<code>.and.</code>	<code>omp_priv = .true.</code>	<code>omp_out = omp_in .and. omp_out</code>
<code>.or.</code>	<code>omp_priv = .false.</code>	<code>omp_out = omp_in .or. omp_out</code>
<code>.eqv.</code>	<code>omp_priv = .true.</code>	<code>omp_out = omp_in .eqv. omp_out</code>
<code>.neqv.</code>	<code>omp_priv = .false.</code>	<code>omp_out = omp_in .neqv. omp_out</code>
<code>max</code>	<code>omp_priv = Least</code> <i>representable number in the</i> <i>reduction list item type</i>	<code>omp_out = max(omp_in, omp_out)</code>
<code>min</code>	<code>omp_priv = Largest</code> <i>representable number in the</i> <i>reduction list item type</i>	<code>omp_out = min(omp_in, omp_out)</code>

table continued on next page

table continued from previous page

Identifier	Initializer	Combiner
iand	omp_priv = <i>All bits on</i>	omp_out = iand (omp_in , omp_out)
ior	omp_priv = 0	omp_out = ior (omp_in , omp_out)
ieor	omp_priv = 0	omp_out = ieor (omp_in , omp_out)

Fortran

In the above tables, **omp_in** and **omp_out** correspond to two identifiers that refer to storage of the type of the list item. **omp_out** holds the final value of the combiner operation.

Any *reduction-identifier* that is defined with the **declare reduction** directive is also valid. In that case, the initializer and combiner of the *reduction-identifier* are specified by the *initializer-clause* and the *combiner* in the **declare reduction** directive.

Description

A reduction clause specifies a *reduction-identifier* and one or more list items.

The *reduction-identifier* specified in a reduction clause must match a previously declared *reduction-identifier* of the same name and type for each of the list items. This match is done by means of a name lookup in the base language.

The list items that appear in a reduction clause may include array sections.

C++

If the type is a derived class, then any *reduction-identifier* that matches its base classes is also a match, if there is no specific match for the type.

If the *reduction-identifier* is not an *id-expression*, then it is implicitly converted to one by prepending the keyword operator (for example, **+** becomes *operator+*).

If the *reduction-identifier* is qualified then a qualified name lookup is used to find the declaration.

If the *reduction-identifier* is unqualified then an *argument-dependent name lookup* must be performed using the type of each list item.

C++

If the list item is an array or array section, it will be treated as if a reduction clause would be applied to each separate element of the array section.

If the list item is an array section, the elements of any copy of the array section will be allocated contiguously.

Fortran

If the original list item has the **POINTER** attribute, any copies of the list item are associated with private targets.

Fortran

Any copies associated with the reduction are initialized with the initializer value of the *reduction-identifier*.

Any copies are combined using the combiner associated with the *reduction-identifier*.

Execution Model Events

The *reduction-begin* event occurs before a task begins to perform loads and stores that belong to the implementation of a reduction and the *reduction-end* event occurs after the task has completed loads and stores associated with the reduction. If a task participates in multiple reductions, each reduction may be bracketed by its own pair of *reduction-begin/reduction-end* events or multiple reductions may be bracketed by a single pair of events. The interval defined by a pair of *reduction-begin/reduction-end* events may not contain a task scheduling point.

Tool Callbacks

A thread dispatches a registered **ompt_callback_reduction** with **ompt_sync_region_reduction** in its *kind* argument and **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *reduction-begin* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_reduction** with **ompt_sync_region_reduction** in its *kind* argument and **ompt_scope_end** as its *endpoint* argument for each occurrence of a *reduction-end* event in that thread. These callbacks occur in the context of the task that performs the reduction and has the type signature **ompt_callback_sync_region_t**.

Restrictions

The restrictions common to reduction clauses are as follows:

- Any number of reduction clauses can be specified on the directive, but a list item (or any array element in an array section) can appear only once in reduction clauses for that directive.
- For a *reduction-identifier* declared with the **declare reduction** construct, the directive must appear before its use in a reduction clause.
- If a list item is an array section or an array element, its base expression must be a base language identifier.
- If a list item is an array section, it must specify contiguous storage and it cannot be a zero-length array section.

- If a list item is an array section or an array element, accesses to the elements of the array outside the specified array section or array element result in unspecified behavior.

C

- A variable that is part of another variable, with the exception of array elements, cannot appear in a reduction clause.

C

C++

- A variable that is part of another variable, with the exception of array elements, cannot appear in a reduction clause except if the reduction clause is associated with a construct within a class non-static member function and the variable is an accessible data member of the object for which the non-static member function is invoked.

C++

C / C++

- The type of a list item that appears in a reduction clause must be valid for the *reduction-identifier*. For a **max** or **min** reduction in C, the type of the list item must be an allowed arithmetic data type: **char**, **int**, **float**, **double**, or **_Bool**, possibly modified with **long**, **short**, **signed**, or **unsigned**. For a **max** or **min** reduction in C++, the type of the list item must be an allowed arithmetic data type: **char**, **wchar_t**, **int**, **float**, **double**, or **bool**, possibly modified with **long**, **short**, **signed**, or **unsigned**.

- A list item that appears in a reduction clause must not be **const**-qualified.
- The *reduction-identifier* for any list item must be unambiguous and accessible.

C / C++

Fortran

- A variable that is part of another variable, with the exception of array elements, cannot appear in a reduction clause.

- A type parameter inquiry cannot appear in a reduction clause.

- The type, type parameters and rank of a list item that appears in a reduction clause must be valid for the *combiner* and *initializer*.

- A list item that appears in a reduction clause must be definable.

- A procedure pointer may not appear in a reduction clause.

- A pointer with the **INTENT (IN)** attribute may not appear in the reduction clause.

- An original list item with the **POINTER** attribute or any pointer component of an original list item that is referenced in the *combiner* must be associated at entry to the construct that contains the reduction clause. Additionally, the list item or the pointer component of the list item must not be deallocated, allocated, or pointer assigned within the region.

- An original list item with the **ALLOCATABLE** attribute or any allocatable component of an original list item that corresponds to the special variable identifier in the *combiner* or the *initializer* must be in the allocated state at entry to the construct that contains the reduction clause. Additionally, the list item or the allocatable component of the list item must be neither deallocated nor allocated, explicitly or implicitly, within the region.
- If the *reduction-identifier* is defined in a **declare reduction** directive, the **declare reduction** directive must be in the same subprogram, or accessible by host or use association.
- If the *reduction-identifier* is a user-defined operator, the same explicit interface for that operator must be accessible as at the **declare reduction** directive.
- If the *reduction-identifier* is defined in a **declare reduction** directive, any subroutine or function referenced in the initializer clause or combiner expression must be an intrinsic function, or must have an explicit interface where the same explicit interface is accessible as at the **declare reduction** directive.

Fortran

Cross References

- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.4.4.11 on page 443.
- **ompt_sync_region_reduction**, see Section 4.4.4.13 on page 444.
- **ompt_callback_sync_region_t**, see Section 4.5.2.13 on page 474.

2.19.5.2 Reduction Scoping Clauses

Reduction scoping clauses define the region in which a reduction is computed by tasks or SIMD lanes. All properties common to all reduction clauses, which are defined in Section 2.19.5.1 on page 294, apply to reduction scoping clauses.

The number of copies created for each list item and the time at which those copies are initialized are determined by the particular reduction scoping clause that appears on the construct.

The time at which the original list item contains the result of the reduction is determined by the particular reduction scoping clause.

The location in the OpenMP program at which values are combined and the order in which values are combined are unspecified. Therefore, when comparing sequential and parallel runs, or when comparing one parallel run to another (even if the number of threads used is the same), there is no guarantee that bitwise-identical results will be obtained or that side effects (such as floating-point exceptions) will be identical or take place at the same location in the OpenMP program.

To avoid data races, concurrent reads or updates of the original list item must be synchronized with the update of the original list item that occurs as a result of the reduction computation.

1 2.19.5.3 Reduction Participating Clauses

2 A reduction participating clause specifies a task or a SIMD lane as a participant in a reduction
3 defined by a reduction scoping clause. All properties common to all reduction clauses, which are
4 defined in Section 2.19.5.1 on page 294, apply to reduction participating clauses.

5 Accesses to the original list item may be replaced by accesses to copies of the original list item
6 created by a region that corresponds to a construct with a reduction scoping clause.

7 In any case, the final value of the reduction must be determined as if all tasks or SIMD lanes that
8 participate in the reduction are executed sequentially in some arbitrary order.

9 2.19.5.4 reduction Clause

10 Summary

11 The **reduction** clause specifies a *reduction-identifier* and one or more list items. For each list
12 item, a private copy is created in each implicit task or SIMD lane and is initialized with the
13 initializer value of the *reduction-identifier*. After the end of the region, the original list item is
14 updated with the values of the private copies using the combiner associated with the
15 *reduction-identifier*.

16 Syntax

17 **reduction** ([*reduction-modifier*,] *reduction-identifier* : *list*)

18 Where *reduction-identifier* is defined in Section 2.19.5.1 on page 294, and *reduction-modifier* is
19 one of the following:

```
20 inscan  
21 task  
22 default
```

23 Description

24 The **reduction** clause is a reduction scoping clause and a reduction participating clause, as
25 described in Section 2.19.5.2 on page 299 and Section 2.19.5.3 on page 300.

26 If *reduction-modifier* is not present or the **default** *reduction-modifier* is present, the behavior is
27 as follows. For **parallel** and worksharing constructs, one or more private copies of each list
28 item are created for each implicit task, as if the **private** clause had been used. For the **simd**
29 construct, one or more private copies of each list item are created for each SIMD lane, as if the
30 **private** clause had been used. For the **taskloop** construct, private copies are created
31 according to the rules of the reduction scoping clauses. For the **teams** construct, one or more

private copies of each list item are created for the initial task of each team in the league, as if the **private** clause had been used. For the **loop** construct, private copies are created and used in the construct according to the description and restrictions in Section 2.19.3 on page 279. At the end of a region that corresponds to a construct for which the **reduction** clause was specified, the original list item is updated by combining its original value with the final value of each of the private copies, using the combiner of the specified *reduction-identifier*.

If the **inscan** *reduction-modifier* is present, a scan computation is performed over updates to the list item performed in each logical iteration of the loop associated with the worksharing-loop, worksharing-loop SIMD, or **simd** construct (see Section 2.9.6 on page 132). The list items are privatized in the construct according to the description and restrictions in Section 2.19.3 on page 279. At the end of the region, each original list item is assigned the value of the private copy from the last logical iteration of the loops associated with the construct.

If the **task** *reduction-modifier* is present for a **parallel** or worksharing construct, then each list item is privatized according to the description and restrictions in Section 2.19.3 on page 279, and an unspecified number of additional private copies are created to support task reductions. Any copies associated with the reduction are initialized before they are accessed by the tasks that participate in the reduction, which include all implicit tasks in the corresponding region and all participating explicit tasks that specify an **in_reduction** clause (see Section 2.19.5.6 on page 303). After the end of the region, the original list item contains the result of the reduction.

If **nowait** is not specified for the construct, the reduction computation will be complete at the end of the construct; however, if the **reduction** clause is used on a construct to which **nowait** is also applied, accesses to the original list item will create a race and, thus, have unspecified effect unless synchronization ensures that they occur after all threads have executed all of their iterations or **section** constructs, and the reduction computation has completed and stored the computed value of that list item. This can most simply be ensured through a barrier synchronization.

Restrictions

The restrictions to the **reduction** clause are as follows:

- All restrictions common to all reduction clauses, which are listed in Section 2.19.5.1 on page 294, apply to this clause.
- A list item that appears in a **reduction** clause of a worksharing construct must be shared in the **parallel** region to which a corresponding worksharing region binds.
- If a list item that appears in a **reduction** clause of a worksharing construct or **loop** construct for which the corresponding region binds to a parallel region is an array section or an array element, all threads that participate in the reduction must specify the same storage location.
- A list item that appears in a **reduction** clause with the **inscan** *reduction-modifier* must appear as a list item in an **inclusive** or **exclusive** clause on a **scan** directive enclosed by the construct.

- A **reduction** clause without the **inscan** *reduction-modifier* may not appear on a construct on which a **reduction** clause with the **inscan** *reduction-modifier* appears.
- A **reduction** clause with the **task** *reduction-modifier* may only appear on a **parallel** construct, a worksharing construct or a combined or composite construct for which any of the aforementioned constructs is a constituent construct and **simd** or **loop** are not constituent constructs.
- A **reduction** clause with the **inscan** *reduction-modifier* may only appear on a worksharing-loop construct, a worksharing-loop SIMD construct, a **simd** construct, a parallel worksharing-loop construct or a parallel worksharing-loop SIMD construct.
- A list item that appears in a **reduction** clause of the innermost enclosing worksharing or **parallel** construct may not be accessed in an explicit task generated by a construct for which an **in_reduction** clause over the same list item does not appear.
- The **task** *reduction-modifier* may not appear in a **reduction** clause if the **nowait** clause is specified on the same construct.

C / C++

- If a list item in a **reduction** clause on a worksharing construct or **loop** construct for which the corresponding region binds to a parallel region has a reference type then it must bind to the same object for all threads of the team.
- If a list item in a **reduction** clause on a worksharing construct or **loop** construct for which the corresponding region binds to a parallel region is an array section or an array element then the base pointer must point to the same variable for all threads of the team.
- A variable of class type (or array thereof) that appears in a **reduction** clause with the **inscan** *reduction-modifier* requires an accessible, unambiguous default constructor for the class type. The number of calls to the default constructor while performing the scan computation is unspecified.
- A variable of class type (or array thereof) that appears in a **reduction** clause with the **inscan** *reduction-modifier* requires an accessible, unambiguous copy assignment operator for the class type. The number of calls to the copy assignment operator while performing the scan computation is unspecified.

C / C++

Cross References

- **scan** directive, see Section 2.9.6 on page 132.
- List Item Privatization, see Section 2.19.3 on page 279.
- **private** clause, see Section 2.19.4.3 on page 285.

1 2.19.5.5 **task_reduction** Clause

2 Summary

3 The **task_reduction** clause specifies a reduction among tasks.

4 Syntax

5 **task_reduction** (*reduction-identifier* : *list*)

6 Where *reduction-identifier* is defined in Section [2.19.5.1](#).

7 Description

8 The **task_reduction** clause is a reduction scoping clause, as described in [2.19.5.2](#).

9 For each list item, the number of copies is unspecified. Any copies associated with the reduction
10 are initialized before they are accessed by the tasks participating in the reduction. After the end of
11 the region, the original list item contains the result of the reduction.

12 Restrictions

13 The restrictions to the **task_reduction** clause are as follows:

- 14 • All restrictions common to all reduction clauses, which are listed in Section [2.19.5.1](#) on
15 page [294](#), apply to this clause.

16 2.19.5.6 **in_reduction** Clause

17 Summary

18 The **in_reduction** clause specifies that a task participates in a reduction.

19 Syntax

20 **in_reduction** (*reduction-identifier* : *list*)

21 where *reduction-identifier* is defined in Section [2.19.5.1](#) on page [294](#).

22 Description

23 The **in_reduction** clause is a reduction participating clause, as described in Section [2.19.5.3](#)
24 on page [300](#). For a given a list item, the **in_reduction** clause defines a task to be a participant
25 in a task reduction that is defined by an enclosing region for a matching list item that appears in a
26 **task_reduction** clause or a **reduction** clause with the **task** modifier, where either:

1. The matching list item has the same storage location as the list item in the **in_reduction** clause; or
2. A private copy, derived from the matching list item, that is used to perform the task reduction has the same storage location as the list item in the **in_reduction** clause.

For the **task** construct, the generated task becomes the participating task. For each list item, a private copy may be created as if the **private** clause had been used.

For the **target** construct, the target task becomes the participating task. For each list item, a private copy will be created in the data environment of the target task as if the **private** clause had been used, and this private copy will be implicitly mapped into the device data environment of the target device.

At the end of the task region, if a private copy was created its value is combined with a copy created by a reduction scoping clause or with the original list item.

Restrictions

The restrictions to the **in_reduction** clause are as follows:

- All restrictions common to all reduction clauses, which are listed in Section 2.19.5.1 on page 294, apply to this clause.
- A list item that appears in a **task_reduction** clause or a **reduction** clause with the **task** modifier that is specified on a construct that corresponds to a region in which the region of the participating task is closely nested must match each list item. The construct that corresponds to the innermost enclosing region that meets this condition must specify the same *reduction-identifier* for the matching list item as the **in_reduction** clause.

2.19.5.7 declare reduction Directive

Summary

The following section describes the directive for declaring user-defined reductions. The **declare reduction** directive declares a *reduction-identifier* that can be used in a **reduction** clause. The **declare reduction** directive is a declarative directive.

Syntax

C

```
#pragma omp declare reduction(reduction-identifier : typename-list :  
combiner) [initializer-clause] new-line
```

where:

- *reduction-identifier* is either a base language identifier or one of the following operators: `+`, `-`, `*`, `&`, `|`, `^`, `&&` and `||`
- *typename-list* is a list of type names
- *combiner* is an expression
- *initializer-clause* is **initializer**(*initializer-expr*) where *initializer-expr* is **omp_priv** = *initializer* or *function-name* (*argument-list*)

C

C++

```
#pragma omp declare reduction(reduction-identifier : typename-list :  
combiner) [initializer-clause] new-line
```

where:

- *reduction-identifier* is either an *id-expression* or one of the following operators: `+`, `-`, `*`, `&`, `|`, `^`, `&&` or `||`
- *typename-list* is a list of type names
- *combiner* is an expression
- *initializer-clause* is **initializer**(*initializer-expr*) where *initializer-expr* is **omp_priv** *initializer* or *function-name* (*argument-list*)

C++

Fortran

```
!$omp declare reduction(reduction-identifier : type-list : combiner)  
[initializer-clause]
```

where:

- *reduction-identifier* is either a base language identifier, or a user-defined operator, or one of the following operators: `+`, `-`, `*`, `.and.`, `.or.`, `.eqv.`, `.neqv.`, or one of the following intrinsic procedure names: **max**, **min**, **iand**, **ior**, **ieor**.
- *type-list* is a list of type specifiers that must not be **CLASS (*)** and abstract type
- *combiner* is either an assignment statement or a subroutine name followed by an argument list
- *initializer-clause* is **initializer**(*initializer-expr*), where *initializer-expr* is **omp_priv** = *expression* or *subroutine-name* (*argument-list*)

Fortran

Description

Custom reductions can be defined using the **declare reduction** directive; the *reduction-identifier* and the type identify the **declare reduction** directive. The *reduction-identifier* can later be used in a **reduction** clause that uses variables of the type or types specified in the **declare reduction** directive. If the directive applies to several types then it is considered as if there were multiple **declare reduction** directives, one for each type.

Fortran

If a type with deferred or assumed length type parameter is specified in a **declare reduction** directive, the *reduction-identifier* of that directive can be used in a reduction clause with any variable of the same type and the same kind parameter, regardless of the length type Fortran parameters with which the variable is declared.

Fortran

The visibility and accessibility of this declaration are the same as those of a variable declared at the same point in the program. The enclosing context of the *combiner* and of the *initializer-expr* is that of the **declare reduction** directive. The *combiner* and the *initializer-expr* must be correct in the base language as if they were the body of a function defined at the same point in the program.

Fortran

If the *reduction-identifier* is the same as the name of a user-defined operator or an extended operator, or the same as a generic name that is one of the allowed intrinsic procedures, and if the operator or procedure name appears in an accessibility statement in the same module, the accessibility of the corresponding **declare reduction** directive is determined by the accessibility attribute of the statement.

If the *reduction-identifier* is the same as a generic name that is one of the allowed intrinsic procedures and is accessible, and if it has the same name as a derived type in the same module, the accessibility of the corresponding **declare reduction** directive is determined by the accessibility of the generic name according to the base language.

Fortran

C++

The **declare reduction** directive can also appear at points in the program at which a static data member could be declared. In this case, the visibility and accessibility of the declaration are the same as those of a static data member declared at the same point in the program.

C++

The *combiner* specifies how partial results can be combined into a single value. The *combiner* can use the special variable identifiers **omp_in** and **omp_out** that are of the type of the variables that this *reduction-identifier* reduces. Each of them will denote one of the values to be combined before executing the *combiner*. The special **omp_out** identifier refers to the storage that holds the resulting combined value after executing the *combiner*.

The number of times that the *combiner* is executed, and the order of these executions, for any reduction clause is unspecified.

Fortran

If the *combiner* is a subroutine name with an argument list, the *combiner* is evaluated by calling the subroutine with the specified argument list.

If the *combiner* is an assignment statement, the *combiner* is evaluated by executing the assignment statement.

Fortran

As the *initializer-expr* value of a user-defined reduction is not known *a priori* the *initializer-clause* can be used to specify one. Then the contents of the *initializer-clause* will be used as the initializer for private copies of reduction list items where the **omp_priv** identifier will refer to the storage to be initialized. The special identifier **omp_orig** can also appear in the *initializer-clause* and it will refer to the storage of the original variable to be reduced.

The number of times that the *initializer-expr* is evaluated, and the order of these evaluations, is unspecified.

C / C++

If the *initializer-expr* is a function name with an argument list, the *initializer-expr* is evaluated by calling the function with the specified argument list. Otherwise, the *initializer-expr* specifies how **omp_priv** is declared and initialized.

C / C++

C

If no *initializer-clause* is specified, the private variables will be initialized following the rules for initialization of objects with static storage duration.

C

C++

If no *initializer-expr* is specified, the private variables will be initialized following the rules for *default-initialization*.

C++

Fortran

If the *initializer-expr* is a subroutine name with an argument list, the *initializer-expr* is evaluated by calling the subroutine with the specified argument list.

If the *initializer-expr* is an assignment statement, the *initializer-expr* is evaluated by executing the assignment statement.

If no *initializer-clause* is specified, the private variables will be initialized as follows:

- For **complex**, **real**, or **integer** types, the value 0 will be used.
- For **logical** types, the value **.false.** will be used.

- For derived types for which default initialization is specified, default initialization will be used.
- Otherwise, not specifying an *initializer-clause* results in unspecified behavior.

Fortran

C / C++

If *reduction-identifier* is used in a **target** region then a **declare target** construct must be specified for any function that can be accessed through the *combiner* and *initializer-expr*.

C / C++

Fortran

If *reduction-identifier* is used in a **target** region then a **declare target** construct must be specified for any function or subroutine that can be accessed through the *combiner* and *initializer-expr*.

Fortran

Restrictions

- The only variables allowed in the *combiner* are **omp_in** and **omp_out**.
- The only variables allowed in the *initializer-clause* are **omp_priv** and **omp_orig**.
- If the variable **omp_orig** is modified in the *initializer-clause*, the behavior is unspecified.
- If execution of the *combiner* or the *initializer-expr* results in the execution of an OpenMP construct or an OpenMP API call, then the behavior is unspecified.
- A *reduction-identifier* may not be re-declared in the current scope for the same type or for a type that is compatible according to the base language rules.
- At most one *initializer-clause* can be specified.
- The *typename-list* must not declare new types.

C / C++

- A type name in a **declare reduction** directive cannot be a function type, an array type, a reference type, or a type qualified with **const**, **volatile** or **restrict**.

C / C++

C

- If the *initializer-expr* is a function name with an argument list, then one of the arguments must be the address of **omp_priv**.

C

C++

- If the *initializer-expr* is a function name with an argument list, then one of the arguments must be **omp_priv** or the address of **omp_priv**.

C++

- If the *initializer-expr* is a subroutine name with an argument list, then one of the arguments must be **omp_priv**.
- If the **declare reduction** directive appears in the specification part of a module and the corresponding reduction clause does not appear in the same module, the *reduction-identifier* must be the same as the name of a user-defined operator, one of the allowed operators that is extended or a generic name that is the same as the name of one of the allowed intrinsic procedures.
- If the **declare reduction** directive appears in the specification of a module, if the corresponding **reduction** clause does not appear in the same module, and if the *reduction-identifier* is the same as the name of a user-defined operator or an extended operator, or the same as a generic name that is the same as one of the allowed intrinsic procedures then the interface for that operator or the generic name must be defined in the specification of the same module, or must be accessible by use association.
- Any subroutine or function used in the **initializer** clause or *combiner* expression must be an intrinsic function, or must have an accessible interface.
- Any user-defined operator, defined assignment or extended operator used in the **initializer** clause or *combiner* expression must have an accessible interface.
- If any subroutine, function, user-defined operator, defined assignment or extended operator is used in the **initializer** clause or *combiner* expression, it must be accessible to the subprogram in which the corresponding **reduction** clause is specified.
- If the length type parameter is specified for a type, it must be a constant, a colon or an *****.
- If a type with deferred or assumed length parameter is specified in a **declare reduction** directive, no other **declare reduction** directive with the same type, the same kind parameters and the same *reduction-identifier* is allowed in the same scope.
- Any subroutine used in the **initializer** clause or *combiner* expression must not have any alternate returns appear in the argument list.

Cross References

- Properties Common To All Reduction Clauses, see Section [2.19.5.1](#) on page [294](#).

2.19.6 Data Copying Clauses

This section describes the **copyin** clause (allowed on the **parallel** construct and combined parallel worksharing constructs) and the **copyprivate** clause (allowed on the **single** construct).

These clauses support the copying of data values from private or threadprivate variables on one implicit task or thread to the corresponding variables on other implicit tasks or threads in the team.

The clauses accept a comma-separated list of list items (see Section 2.1 on page 38). All list items appearing in a clause must be visible, according to the scoping rules of the base language. Clauses may be repeated as needed, but a list item that specifies a given variable may not appear in more than one clause on the same directive.

Fortran

An associate name preserves the association with the selector established at the **ASSOCIATE** statement. A list item that appears in a data copying clause may be a selector of an **ASSOCIATE** construct. If the construct association is established prior to a parallel region, the association between the associate name and the original list item will be retained in the region.

Fortran

2.19.6.1 copyin Clause

Summary

The **copyin** clause provides a mechanism to copy the value of a threadprivate variable of the master thread to the threadprivate variable of each other member of the team that is executing the **parallel** region.

Syntax

The syntax of the **copyin** clause is as follows:

copyin (*list*)

Description

C / C++

The copy is done after the team is formed and prior to the start of execution of the associated structured block. For variables of non-array type, the copy occurs by copy assignment. For an array of elements of non-array type, each element is copied as if by assignment from an element of the array of the master thread to the corresponding element of the array of the other thread.

C / C++

C++

For class types, the copy assignment operator is invoked. The order in which copy assignment operators for different variables of class type are called is unspecified.

C++

Fortran

The copy is done, as if by assignment, after the team is formed and prior to the start of execution of the associated structured block.

On entry to any **parallel** region, each thread's copy of a variable that is affected by a **copyin** clause for the **parallel** region will acquire the type parameters, allocation, association, and definition status of the copy of the master thread, according to the following rules:

- If the original list item has the **POINTER** attribute, each copy receives the same association status as that of the copy of the master thread as if by pointer assignment.
- If the original list item does not have the **POINTER** attribute, each copy becomes defined with the value of the copy of the master thread as if by intrinsic assignment unless the list item has a type bound procedure as a defined assignment. If the original list item that does not have the **POINTER** attribute has the allocation status of unallocated, each copy will have the same status.
- If the original list item is unallocated or unassociated, the copy of the other thread inherits the declared type parameters and the default type parameter values from the original list item.

Fortran

Restrictions

The restrictions to the **copyin** clause are as follows:

C / C++

- A list item that appears in a **copyin** clause must be threadprivate.
- A variable of class type (or array thereof) that appears in a **copyin** clause requires an accessible, unambiguous copy assignment operator for the class type.

C / C++

Fortran

- A list item that appears in a **copyin** clause must be threadprivate. Named variables that appear in a threadprivate common block may be specified: it is not necessary to specify the whole common block.
- A common block name that appears in a **copyin** clause must be declared to be a common block in the same scoping unit in which the **copyin** clause appears.
- If the list item is a polymorphic variable with the **ALLOCATABLE** attribute, the behavior is unspecified.

Fortran

Cross References

- **parallel** construct, see Section 2.6 on page 74.
- **threadprivate** directive, see Section 2.19.2 on page 274.

2.19.6.2 copyprivate Clause

Summary

The **copyprivate** clause provides a mechanism to use a private variable to broadcast a value from the data environment of one implicit task to the data environments of the other implicit tasks that belong to the **parallel** region.

To avoid data races, concurrent reads or updates of the list item must be synchronized with the update of the list item that occurs as a result of the **copyprivate** clause.

Syntax

The syntax of the **copyprivate** clause is as follows:

```
copyprivate (list)
```

Description

The effect of the **copyprivate** clause on the specified list items occurs after the execution of the structured block associated with the **single** construct (see Section 2.8.2 on page 89), and before any of the threads in the team have left the barrier at the end of the construct.

▼ C / C++ ▼

In all other implicit tasks that belong to the **parallel** region, each specified list item becomes defined with the value of the corresponding list item in the implicit task associated with the thread that executed the structured block. For variables of non-array type, the definition occurs by copy assignment. For an array of elements of non-array type, each element is copied by copy assignment from an element of the array in the data environment of the implicit task that is associated with the thread that executed the structured block to the corresponding element of the array in the data environment of the other implicit tasks

▲ C / C++ ▲

▼ C++ ▼

For class types, a copy assignment operator is invoked. The order in which copy assignment operators for different variables of class type are called is unspecified.

▲ C++ ▲

Fortran

If a list item does not have the **POINTER** attribute, then in all other implicit tasks that belong to the **parallel** region, the list item becomes defined as if by intrinsic assignment with the value of the corresponding list item in the implicit task that is associated with the thread that executed the structured block. If the list item has a type bound procedure as a defined assignment, the assignment is performed by the defined assignment.

If the list item has the **POINTER** attribute, then, in all other implicit tasks that belong to the **parallel** region, the list item receives, as if by pointer assignment, the same association status of the corresponding list item in the implicit task that is associated with the thread that executed the structured block.

The order in which any final subroutines for different variables of a finalizable type are called is unspecified.

Fortran

Note – The **copyprivate** clause is an alternative to using a shared variable for the value when providing such a shared variable would be difficult (for example, in a recursion requiring a different variable at each level).

Restrictions

The restrictions to the **copyprivate** clause are as follows:

- All list items that appear in the **copyprivate** clause must be either threadprivate or private in the enclosing context.
- A list item that appears in a **copyprivate** clause may not appear in a **private** or **firstprivate** clause on the **single** construct.

C++

- A variable of class type (or array thereof) that appears in a **copyprivate** clause requires an accessible unambiguous copy assignment operator for the class type.

C++

Fortran

- A common block that appears in a **copyprivate** clause must be threadprivate.
- Pointers with the **INTENT (IN)** attribute may not appear in the **copyprivate** clause.
- The list item with the **ALLOCATABLE** attribute must have the allocation status of allocated when the intrinsic assignment is performed.
- If the list item is a polymorphic variable with the **ALLOCATABLE** attribute, the behavior is unspecified.

Fortran

Cross References

- **parallel** construct, see Section 2.6 on page 74.
- **threadprivate** directive, see Section 2.19.2 on page 274.
- **private** clause, see Section 2.19.4.3 on page 285.

2.19.7 Data-Mapping Attribute Rules, Clauses, and Directives

This section describes how the data-mapping and data-sharing attributes of any variable referenced in a **target** region are determined. When specified, explicit data-sharing attributes, **map** or **is_device_ptr** clauses on **target** directives determine these attributes. Otherwise, the first matching rule from the following implicit data-mapping rules applies for variables referenced in a **target** construct that are not declared in the construct and do not appear in data-sharing attribute, **map** or **is_device_ptr** clauses.

- If a variable appears in a **to** or **link** clause on a **declare target** directive then it is treated as if it had appeared in a **map** clause with a *map-type* of **tofrom**.
- If a list item appears in a **reduction**, **lastprivate** or **linear** clause on a combined **target** construct then it is treated as if it also appears in a **map** clause with a *map-type* of **tofrom**.
- If a list item appears in an **in_reduction** clause on a **target** construct then it is treated as if it also appears in a **map** clause with a *map-type* of **tofrom** and a *map-type-modifier* of **always**.
- If a **defaultmap** clause is present for the category of the variable and specifies an implicit behavior other than **default**, the data-mapping attribute is determined by that clause.

C++

- If the **target** construct is within a class non-static member function, and a variable is an accessible data member of the object for which the non-static data member function is invoked, the variable is treated as if the **this[:1]** expression had appeared in a **map** clause with a *map-type* of **tofrom**. Additionally, if the variable is of a type pointer or reference to pointer, it is also treated as if it has appeared in a **map** clause as a zero-length array section.
- If the **this** keyword is referenced inside a **target** construct within a class non-static member function, it is treated as if the **this[:1]** expression had appeared in a **map** clause with a *map-type* of **tofrom**.

C++

- C / C++
- 1 • A variable that is of type pointer is treated as if it is the base pointer of a zero-length array
2 section that appeared as a list item in a **map** clause.
- C / C++
- C++
- 3 • A variable that is of type reference to pointer is treated as if it had appeared in a **map** clause as a
4 zero-length array section.
- C++
- 5 • If a variable is not a scalar then it is treated as if it had appeared in a **map** clause with a *map-type*
6 of **tofrom**.
- Fortran
- 7 • If a scalar variable has the **TARGET**, **ALLOCATABLE** or **POINTER** attribute then it is treated as
8 if it has appeared in a **map** clause with a *map-type* of **tofrom**.
- Fortran
- 9 • If none of the above rules applies then a scalar variable is not mapped, but instead has an implicit
10 data-sharing attribute of mapped, but instead has an implicit data-sharing attribute of firstprivate
11 (see Section 2.19.1.1 on page 270).

12 2.19.7.1 map Clause

13 Summary

14 The **map** clause specifies how an original list item is mapped from the current task's data
15 environment to a corresponding list item in the device data environment of the device identified by
16 the construct.

17 Syntax

18 The syntax of the map clause is as follows:

19 **map** ([[*map-type-modifier* [,] [*map-type-modifier* [,] ...] *map-type* :] *locator-list*)

20 where *map-type* is one of the following:

```
21 to
22 from
23 tofrom
24 alloc
25 release
26 delete
```

and *map-type-modifier* is one of the following:

```
always
close
mapper (mapper-identifier)
```

Description

The list items that appear in a **map** clause may include array sections and structure elements.

The *map-type* and *map-type-modifier* specify the effect of the **map** clause, as described below.

For a given construct, the effect of a **map** clause with the **to**, **from**, or **tofrom** *map-type* is ordered before the effect of a **map** clause with the **alloc**, **release**, or **delete** *map-type*. If a **mapper** is specified for the type being mapped, or explicitly specified with the **mapper** *map-type-modifier*, then the effective **map-type** of a list item will be determined according to the rules of map-type decay.

If a **mapper** is specified for the type being mapped, or explicitly specified with the **mapper** *map-type-modifier*, then all map clauses that appear on the **declare mapper** directive are treated as though they appeared on the construct with the **map** clause. Array sections of a mapper type are mapped as normal, then each element in the array section is mapped according to the rules of the mapper.

C / C++

If a list item in a **map** clause is a variable of structure type then it is treated as if each structure element contained in the variable is a list item in the clause.

C / C++

Fortran

If a list item in a **map** clause is a derived type variable then it is treated as if each component is a list item in the clause.

Each pointer component that is a list item that results from a mapped derived type variable is treated as if its association status is undefined, unless the pointer component appears as another list item or as the base pointer of another list item in a **map** clause on the same construct.

Fortran

If a list item in a **map** clause is a structure element then all other structure elements of the containing structure variable form a *structure sibling list*. The **map** clause and the structure sibling list are associated with the same construct. If a corresponding list item of the structure sibling list item is present in the device data environment when the construct is encountered then:

- If the structure sibling list item does not appear in a **map** clause on the construct then:
 - If the construct is a **target**, **target data**, or **target enter data** construct then the structure sibling list item is treated as if it is a list item in a **map** clause on the construct with a *map-type* of **alloc**.
 - If the construct is **target exit data** construct, then the structure sibling list item is treated as if it is a list item in a **map** clause on the construct with a *map-type* of **release**.

Fortran

- If the structure sibling list item is a pointer then it is treated as if its association status is undefined, unless it appears as the base pointer of another list item in a **map** clause on the same construct.

Fortran

- If the **map** clause in which the structure element appears as a list item has a *map-type* of **delete** and the structure sibling list item does not appear as a list item in a **map** clause on the construct with a *map-type* of **delete** then the structure sibling list item is treated as if it is a list item in a **map** clause on the construct with a *map-type* of **delete**.

If $item_1$ is a list item in a **map** clause, and $item_2$ is another list item in a **map** clause on the same construct that has a base pointer that is, or is part of, $item_1$, then:

- If the **map** clause(s) appear on a **target**, **target data**, or **target enter data** construct, then on entry to the corresponding region the effect of the **map** clause on $item_1$ is ordered to occur before the effect of the **map** clause on $item_2$.
- If the **map** clause(s) appear on a **target**, **target data**, or **target exit data** construct then on exit from the corresponding region the effect of the **map** clause on $item_2$ is ordered to occur before the effect of the **map** clause on $item_1$.

Fortran

If a list item in a **map** clause is an associated pointer and the pointer is not the base pointer of another list item in a **map** clause on the same construct, then it is treated as if its pointer target is implicitly mapped in the same clause. For the purposes of the **map** clause, the mapped pointer target is treated as if its base pointer is the associated pointer.

Fortran

If a list item in a **map** clause has a base pointer, and a pointer variable is present in the device data environment that corresponds to the base pointer when the effect of the **map** clause occurs, then if the corresponding pointer or the corresponding list item is created in the device data environment on entry to the construct, then:

C / C++

1. The corresponding pointer variable is assigned an address such that the corresponding list item can be accessed through the pointer in a **target** region.

C / C++

Fortran

1. The corresponding pointer variable is associated with a pointer target that has the same rank and bounds as the pointer target of the original pointer, such that the corresponding list item can be accessed through the pointer in a **target** region.

Fortran

2. The corresponding pointer variable becomes an attached pointer for the corresponding list item.
3. If the original base pointer and the corresponding attached pointer share storage, then the original list item and the corresponding list item must share storage.

C++

If a *lambda* is mapped explicitly or implicitly, variables that are captured by the *lambda* behave as follows:

- the variables that are of pointer type are treated as if they had appeared in a **map** clause as zero-length array sections; and
- the variables that are of reference type are treated as if they had appeared in a **map** clause.

If a member variable is captured by a *lambda* in class scope, and the *lambda* is later mapped explicitly or implicitly with its full static type, the **this** pointer is treated as if it had appeared on a **map** clause.

C++

The original and corresponding list items may share storage such that writes to either item by one task followed by a read or write of the other item by another task without intervening synchronization can result in data races.

If the **map** clause appears on a **target**, **target data**, or **target enter data** construct then on entry to the region the following sequence of steps occurs as if performed as a single atomic operation:

1. If a corresponding list item of the original list item is not present in the device data environment, then:
 - a) A new list item with language-specific attributes is derived from the original list item and created in the device data environment;
 - b) The new list item becomes the corresponding list item of the original list item in the device data environment;
 - c) The corresponding list item has a reference count that is initialized to zero; and
 - d) The value of the corresponding list item is undefined;
2. If the corresponding list item's reference count was not already incremented because of the effect of a **map** clause on the construct then:
 - a) The corresponding list item's reference count is incremented by one;

3. If the corresponding list item's reference count is one or the **always map-type-modifier** is present, and if the *map-type* is **to** or **tofrom**, then:

C / C++

a) For each part of the list item that is an attached pointer, that part of the corresponding list item will have the value that it had immediately prior to the effect of the **map** clause; and

C / C++

Fortran

a) For each part of the list item that is an attached pointer, that part of the corresponding list item, if associated, will be associated with the same pointer target that it was associated with immediately prior to the effect of the **map** clause.

Fortran

b) For each part of the list item that is not an attached pointer, the value of that part of the original list item is assigned to that part of the corresponding list item.

Note – If the effect of the **map** clauses on a construct would assign the value of an original list item to a corresponding list item more than once, then an implementation is allowed to ignore additional assignments of the same value to the corresponding list item.

In all cases on entry to the region, concurrent reads or updates of any part of the corresponding list item must be synchronized with any update of the corresponding list item that occurs as a result of the **map** clause to avoid data races.

If the **map** clause appears on a **target**, **target data**, or **target exit data** construct and a corresponding list item of the original list item is not present in the device data environment on exit from the region then the list item is ignored. Alternatively, if the **map** clause appears on a **target**, **target data**, or **target exit data** construct and a corresponding list item of the original list item is present in the device data environment on exit from the region, then the following sequence of steps occurs as if performed as a single atomic operation:

1. If the *map-type* is not **delete** and the corresponding list item's reference count is finite and was not already decremented because of the effect of a **map** clause on the construct then:

a) The corresponding list item's reference count is decremented by one;

2. If the *map-type* is **delete** and the corresponding list item's reference count is finite then:

a) The corresponding list item's reference count is set to zero;

3. If the *map-type* is **from** or **tofrom** and if the corresponding list item's reference count is zero or the **always map-type-modifier** is present then:

C / C++

- a) For each part of the list item that is an attached pointer, that part of the original list item will have the value that it had immediately prior to the effect of the **map** clause;

C / C++

Fortran

- a) For each part of the list item that is an attached pointer, that part of the corresponding list item, if associated, will be associated with the same pointer target with which it was associated immediately prior to the effect of the **map** clause; and

Fortran

- b) For each part of the list item that is not an attached pointer, the value of that part of the corresponding list item is assigned to that part of the original list item; and
4. If the corresponding list item's reference count is zero then the corresponding list item is removed from the device data environment.

Note – If the effect of the **map** clauses on a construct would assign the value of a corresponding list item to an original list item more than once, then an implementation is allowed to ignore additional assignments of the same value to the original list item.

In all cases on exit from the region, concurrent reads or updates of any part of the original list item must be synchronized with any update of the original list item that occurs as a result of the **map** clause to avoid data races.

If a single contiguous part of the original storage of a list item with an implicit data-mapping attribute has corresponding storage in the device data environment prior to a task encountering the construct that is associated with the **map** clause, only that part of the original storage will have corresponding storage in the device data environment as a result of the **map** clause.

If a list item with an implicit data-mapping attribute does not have any corresponding storage in the device data environment prior to a task encountering the construct associated with the **map** clause, and one or more contiguous parts of the original storage are either list items or base pointers to list items that are explicitly mapped on the construct, only those parts of the original storage will have corresponding storage in the device data environment as a result of the **map** clauses on the construct.

C / C++

If a new list item is created then a new list item of the same type, with automatic storage duration, is allocated for the construct. The size and alignment of the new list item are determined by the static type of the variable. This allocation occurs if the region references the list item in any statement. Initialization and assignment of the new list item are through bitwise copy.

C / C++

Fortran

If a new list item is created then a new list item of the same type, type parameter, and rank is allocated. The new list item inherits all default values for the type parameters from the original list item. The value of the new list item becomes that of the original list item in the map initialization and assignment.

If the allocation status of the original list item with the **ALLOCATABLE** attribute is changed in the host device data environment and the corresponding list item is already present in the device data environment, the allocation status of the corresponding list item is unspecified until a mapping operation is performed with a **map** clause on entry to a **target**, **target data**, or **target enter data** region.

Fortran

The *map-type* determines how the new list item is initialized.

If a *map-type* is not specified, the *map-type* defaults to **tofrom**.

The **close** *map-type-modifier* is a hint to the runtime to allocate memory close to the target device.

Execution Model Events

The *target-map* event occurs when a thread maps data to or from a target device.

The *target-data-op* event occurs when a thread initiates a data operation on a target device.

Tool Callbacks

A thread dispatches a registered **ompt_callback_target_map** callback for each occurrence of a *target-map* event in that thread. The callback occurs in the context of the target task and has type signature **ompt_callback_target_map_t**.

A thread dispatches a registered **ompt_callback_target_data_op** callback for each occurrence of a *target-data-op* event in that thread. The callback occurs in the context of the target task and has type signature **ompt_callback_target_data_op_t**.

Restrictions

The restrictions to the **map** clause are as follows:

- A list item cannot appear in both a **map** clause and a data-sharing attribute clause on the same construct unless the construct is a combined construct.
- Each of the *map-type-modifier* modifiers can appear at most once on the **map** clause.

C / C++

- List items of the **map** clauses on the same construct must not share original storage unless they are the same lvalue expression or array section.

C / C++

- If a list item is an array section, it must specify contiguous storage.
- If multiple list items are explicitly mapped on the same construct and have the same containing array or have base pointers that share original storage, and if any of the list items do not have corresponding list items that are present in the device data environment prior to a task encountering the construct, then the list items must refer to the same array elements of either the containing array or the implicit array of the base pointers.
- If any part of the original storage of a list item with an explicit data-mapping attribute has corresponding storage in the device data environment prior to a task encountering the construct associated with the **map** clause, all of the original storage must have corresponding storage in the device data environment prior to the task encountering the construct.
- If a list item is an element of a structure, and a different element of the structure has a corresponding list item in the device data environment prior to a task encountering the construct associated with the **map** clause, then the list item must also have a corresponding list item in the device data environment prior to the task encountering the construct.
- A list item must have a mappable type.
- **threadprivate** variables cannot appear in a **map** clause.
- If a **mapper** map-type-modifier is specified, its type must match the type of the list-items passed to that map clause.
- Memory spaces and memory allocators cannot appear as a list item in a **map** clause.

C++

- If the type of a list item is a reference to a type *T* then the reference in the device data environment is initialized to refer to the object in the device data environment that corresponds to the object referenced by the list item. If mapping occurs, it occurs as though the object were mapped through a pointer with an array section of type *T* and length one.
- No type mapped through a reference can contain a reference to its own type, or any references to types that could produce a cycle of references.
- If the list item is a *lambda*, any pointers and references captured by the *lambda* must have the corresponding list item in the device data environment prior to the task encountering the construct.

C++

C / C++

- A list item cannot be a variable that is a member of a structure with a union type.
- A bit-field cannot appear in a **map** clause.
- A pointer that has a corresponding attached pointer must not be modified for the duration of the lifetime of the list item to which the corresponding pointer is attached in the device data environment.

C / C++

Fortran

- List items of the **map** clauses on the same construct must not share original storage unless they are the same variable or array section.
- A pointer that has a corresponding attached pointer and is associated with a given pointer target must not become associated with a different pointer target for the duration of the lifetime of the list item to which the corresponding pointer is attached in the device data environment.
- If the allocation status of a list item or any subobject of the list item with the **ALLOCATABLE** attribute is unallocated upon entry to a **target** region, the list item or any subobject of the corresponding list item must be unallocated upon exit from the region.
- If the allocation status of a list item or any subobject of the list item with the **ALLOCATABLE** attribute is allocated upon entry to a **target** region, the allocation status of the corresponding list item or any subobject of the corresponding list item must not be changed and must not be reshaped in the region.
- If an array section is mapped and the size of the section is smaller than that of the whole array, the behavior of referencing the whole array in the **target** region is unspecified.
- A list item must not be a whole array of an assumed-size array.
- If the association status of a list item with the **POINTER** attribute is associated upon entry to a **target** region, the list item must be associated with the same pointer target upon exit from the region.
- If the association status of a list item with the **POINTER** attribute is disassociated upon entry to a **target** region, the list item must be disassociated upon exit from the region.
- If the association status of a list item with the **POINTER** attribute is undefined upon entry to a **target** region, the list item must be undefined upon exit from the region.
- If the association status of a list item with the **POINTER** attribute is disassociated or undefined on entry and if the list item is associated with a pointer target inside a **target** region, then the pointer association status must become disassociated before the end of the region.

Fortran

Cross References

- `ompt_callback_target_data_op_t`, see Section [4.5.2.25](#) on page [488](#).
- `ompt_callback_target_map_t`, see Section [4.5.2.27](#) on page [492](#).

2.19.7.2 defaultmap Clause

Summary

The **defaultmap** clause explicitly determines the data-mapping attributes of variables that are referenced in a **target** construct for which the data-mapping attributes would otherwise be implicitly determined (see Section [2.19.7](#) on page [314](#)).

Syntax

The syntax of the **defaultmap** clause is as follows:

```
defaultmap (implicit-behavior[:variable-category])
```

Where *implicit-behavior* is one of:

```
alloc  
to  
from  
tofrom  
firstprivate  
none  
default
```

↔ C / C++ ↔

and *variable-category* is one of:

```
scalar  
aggregate  
pointer
```

↔ C / C++ ↔

and *variable-category* is one of:

```

scalar
aggregate
allocatable
pointer

```

Description

The **defaultmap** clause sets the implicit data-mapping attribute for all variables referenced in the construct. If *variable-category* is specified, the effect of the **defaultmap** clause is as follows:

- If *variable-category* is **scalar**, all scalar variables of non-pointer type or all non-pointer non-allocatable scalar variables that have an implicitly determined data-mapping or data-sharing attribute will have a data-mapping or data-sharing attribute specified by *implicit-behavior*.
- If *variable-category* is **aggregate** or **allocatable**, all aggregate or allocatable variables that have an implicitly determined data-mapping or data-sharing attribute will have a data-mapping or data-sharing attribute specified by *implicit-behavior*.
- If *variable-category* is **pointer**, all variables of pointer type or with the **POINTER** attribute that have implicitly determined data-mapping or data-sharing attributes will have a data-mapping or data-sharing attribute specified by *implicit-behavior*. The zero-length array section and attachment that are otherwise applied to an implicitly mapped pointer are only provided for the **default** behavior.

If no *variable-category* is specified in the clause then *implicit-behavior* specifies the implicitly determined data-mapping or data-sharing attribute for all variables referenced in the construct. If *implicit-behavior* is **none**, each variable referenced in the construct that does not have a predetermined data-sharing attribute and does not appear in a **to** or **link** clause on a **declare target** directive must be listed in a data-mapping attribute clause, a data-sharing attribute clause (including a data-sharing attribute clause on a combined construct where **target** is one of the constituent constructs), or an **is_device_ptr** clause. If *implicit-behavior* is **default**, then the clause has no effect for the variables in the category specified by *variable-category*.

1 2.19.7.3 declare mapper Directive

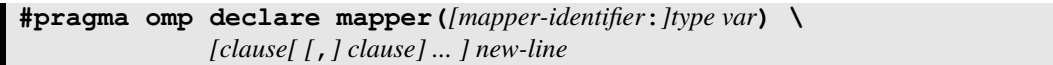
2 Summary

3 The **declare mapper** directive declares a user-defined mapper for a given type, and may define
4 a *mapper-identifier* that can be used in a **map** clause. The **declare mapper** directive is a
5 declarative directive.

6 Syntax

7  C / C++

8 The syntax of the **declare mapper** directive is as follows:

9 
#pragma omp declare mapper ([*mapper-identifier*:] *type* *var*) \
[*clause* [,] *clause*] ...] *new-line*

10  C / C++

11  Fortran

12 The syntax of the **declare mapper** directive is as follows:

13 
!\$omp declare mapper ([*mapper-identifier*:] *type* :: *var*) &
[*clause* [,] *clause*] ...]

14  Fortran

15 where:

- 16 • *mapper-identifier* is a base-language identifier or **default**
- 17 • *type* is a valid type in scope
- 18 • *var* is a valid base-language identifier
- 19 • *clause* is **map** ([[*map-type-modifier* [,] [*map-type-modifier* [,] ...]] *map-type*:] *list*) , where
20 *map-type* is one of the following:

- 21 – **alloc**
- 22 – **to**
- 23 – **from**
- 24 – **tofrom**

25 and where *map-type-modifier* is one of the following:

- **always**
- **close**

Description

User-defined mappers can be defined using the **declare mapper** directive. The type and the *mapper-identifier* uniquely identify the mapper for use in a **map** clause later in the program. If the *mapper-identifier* is not specified, then **default** is used. The visibility and accessibility of this declaration are the same as those of a variable declared at the same point in the program.

The variable declared by *var* is available for use in all **map** clauses on the directive, and no part of the variable to be mapped is mapped by default.

The default mapper for all types *T*, designated by the pre-defined *mapper-identifier* **default**, is as follows unless a user-defined mapper is specified for that type.

```
declare mapper (T v) map (tofrom: v)
```

Using the **default** *mapper-identifier* overrides the pre-defined default mapper for the given type, making it the default for all variables of *type*. All **map** clauses with this construct in scope that map a list item of *type* will use this mapper unless another is explicitly specified.

All **map** clauses on the directive are expanded into corresponding **map** clauses wherever this mapper is invoked, either by matching type or by being explicitly named in a **map** clause. A **map** clause with list item *var* maps *var* as though no mapper were specified.

▼ C++ ▼

The **declare mapper** directive can also appear at points in the program at which a static data member could be declared. In this case, the visibility and accessibility of the declaration are the same as those of a static data member declared at the same point in the program.

▲ C++ ▲

Restrictions

The restrictions to the **declare mapper** directive are as follows:

- No instance of *type* can be mapped as part of the mapper, either directly or indirectly through another type, except the instance passed as the list item. If a set of **declare mapper** directives results in a cyclic definition then the behavior is unspecified.
- The *type* must be of struct, union or class type in C and C++ or a non-intrinsic type in Fortran.
- The *type* must not declare a new type.
- At least one **map** clause that maps *var* or at least one element of *var* is required.
- List-items in **map** clauses on this construct may only refer to the declared variable *var* and entities that could be referenced by a procedure defined at the same location.
- Each *map-type-modifier* can appear at most once on the **map** clause.

- A *mapper-identifier* may not be redeclared in the current scope for the same type or for a type that is compatible according to the base language rules.

Fortran

- *type* must not be an abstract type.

Fortran

2.20 Nesting of Regions

This section describes a set of restrictions on the nesting of regions. The restrictions on nesting are as follows:

- A worksharing region may not be closely nested inside a worksharing, **loop**, **task**, **taskloop**, **critical**, **ordered**, **atomic**, or **master** region.
- A **barrier** region may not be closely nested inside a worksharing, **loop**, **task**, **taskloop**, **critical**, **ordered**, **atomic**, or **master** region.
- A **master** region may not be closely nested inside a worksharing, **loop**, **atomic**, **task**, or **taskloop** region.
- An **ordered** region corresponding to an **ordered** construct without any clause or with the **threads** or **depend** clause may not be closely nested inside a **critical**, **ordered**, **loop**, **atomic**, **task**, or **taskloop** region.
- An **ordered** region corresponding to an **ordered** construct without the **simd** clause specified must be closely nested inside a worksharing-loop region.
- An **ordered** region corresponding to an **ordered** construct with the **simd** clause specified must be closely nested inside a **simd** or worksharing-loop SIMD region.
- An **ordered** region corresponding to an **ordered** construct with both the **simd** and **threads** clauses must be closely nested inside a worksharing-loop SIMD region or closely nested inside a worksharing-loop and **simd** region.
- A **critical** region may not be nested (closely or otherwise) inside a **critical** region with the same name. This restriction is not sufficient to prevent deadlock.
- OpenMP constructs may not be encountered during execution of an **atomic** region.
- The only OpenMP constructs that can be encountered during execution of a **simd** (or worksharing-loop SIMD) region are the **atomic** construct, the **loop** construct, the **simd** construct and the **ordered** construct with the **simd** clause.

- If a **target update**, **target data**, **target enter data**, or **target exit data** construct is encountered during execution of a **target** region, the behavior is unspecified.
- If a **target** construct is encountered during execution of a **target** region and a **device** clause in which the **ancestor device-modifier** appears is not present on the construct, the behavior is unspecified.
- A **teams** region can only be strictly nested within the implicit parallel region or a **target** region. If a **teams** construct is nested within a **target** construct, that **target** construct must contain no statements, declarations or directives outside of the **teams** construct.
- **distribute**, **distribute simd**, distribute parallel worksharing-loop, distribute parallel worksharing-loop SIMD, **loop**, **parallel** regions, including any **parallel** regions arising from combined constructs, **omp_get_num_teams()** regions, and **omp_get_team_num()** regions are the only OpenMP regions that may be strictly nested inside the **teams** region.
- The region corresponding to the **distribute** construct must be strictly nested inside a **teams** region.
- If *construct-type-clause* is **taskgroup**, the **cancel** construct must be closely nested inside a **task** construct and the **cancel** region must be closely nested inside a **taskgroup** region. If *construct-type-clause* is **sections**, the **cancel** construct must be closely nested inside a **sections** or **section** construct. Otherwise, the **cancel** construct must be closely nested inside an OpenMP construct that matches the type specified in *construct-type-clause* of the **cancel** construct.
- A **cancellation point** construct for which *construct-type-clause* is **taskgroup** must be closely nested inside a **task** construct, and the **cancellation point** region must be closely nested inside a **taskgroup** region. A **cancellation point** construct for which *construct-type-clause* is **sections** must be closely nested inside a **sections** or **section** construct. Otherwise, a **cancellation point** construct must be closely nested inside an OpenMP construct that matches the type specified in *construct-type-clause*.
- The only constructs that may be nested inside a **loop** region are the **loop** construct, the **parallel** construct, the **simd** construct, and combined constructs for which the first construct is a **parallel** construct.
- A **loop** region may not contain calls to procedures that contain OpenMP directives or calls to the OpenMP Runtime API.

This page intentionally left blank

CHAPTER 3

Runtime Library Routines

This chapter describes the OpenMP API runtime library routines and queryable runtime states. In this chapter, *true* and *false* are used as generic terms to simplify the description of the routines.

C / C++

true means a nonzero integer value and *false* means an integer value of zero.

C / C++

Fortran

true means a logical value of **.TRUE.** and *false* means a logical value of **.FALSE.**

Fortran

Fortran

Restrictions

The following restriction applies to all OpenMP runtime library routines:

- OpenMP runtime library routines may not be called from **PURE** or **ELEMENTAL** procedures.

Fortran

1 3.1 Runtime Library Definitions

2 For each base language, a compliant implementation must supply a set of definitions for the
3 OpenMP API runtime library routines and the special data types of their parameters. The set of
4 definitions must contain a declaration for each OpenMP API runtime library routine and variable
5 and a definition of each required data type listed below. In addition, each set of definitions may
6 specify other implementation specific values.

▼ C / C++ ▼

7 The library routines are external functions with “C” linkage.

8 Prototypes for the C/C++ runtime library routines described in this chapter shall be provided in a
9 header file named **omp.h**. This file also defines the following:

- 10 • The type **omp_lock_t**;
- 11 • The type **omp_nest_lock_t**;
- 12 • The type **omp_sync_hint_t**;
- 13 • The type **omp_lock_hint_t** (deprecated);
- 14 • The type **omp_sched_t**;
- 15 • The type **omp_proc_bind_t**;
- 16 • The type **omp_control_tool_t**;
- 17 • The type **omp_control_tool_result_t**;
- 18 • The type **omp_depend_t**;
- 19 • The type **omp_memspace_handle_t**, which must be an implementation-defined enum type
20 with an enumerator for at least each predefined memory space in Table 2.8 on page 152;
- 21 • The type **omp_allocator_handle_t**, which must be an implementation-defined enum type
22 with at least the **omp_null_allocator** enumerator with the value zero and an enumerator
23 for each predefined memory allocator in Table 2.10 on page 155;
- 24 • The type **omp_uintptr_t**, which is an unsigned integer type capable of holding a pointer on
25 any device;
- 26 • The type **omp_pause_resource_t**; and
- 27 • The type **omp_event_handle_t**, which must be an implementation-defined enum type.

▲ C / C++ ▲

C++

The `omp.h` header file also defines a class template that models the **Allocator** concept in the `omp::allocator` namespace for each predefined memory allocator in Table 2.10 on page 155 for which the name includes neither the `omp_` prefix nor the `_alloc` suffix.

C++

Fortran

The OpenMP Fortran API runtime library routines are external procedures. The return values of these routines are of default kind, unless otherwise specified.

Interface declarations for the OpenMP Fortran runtime library routines described in this chapter shall be provided in the form of a Fortran **include** file named `omp_lib.h` or a Fortran 90 **module** named `omp_lib`. It is implementation defined whether the **include** file or the **module** file (or both) is provided.

These files also define the following:

- The **integer parameter** `omp_lock_kind`;
- The **integer parameter** `omp_nest_lock_kind`;
- The **integer parameter** `omp_sync_hint_kind`;
- The **integer parameter** `omp_lock_hint_kind` (deprecated);
- The **integer parameter** `omp_sched_kind`;
- The **integer parameter** `omp_proc_bind_kind`;
- The **integer parameter** `omp_control_tool_kind`;
- The **integer parameter** `omp_control_tool_result_kind`;
- The **integer parameter** `omp_depend_kind`;
- The **integer parameter** `omp_memspace_handle_kind`;
- The **integer parameter** `omp_allocator_handle_kind`;
- The **integer parameter** `omp_alloctrail_key_kind`;
- The **integer parameter** `omp_alloctrail_val_kind`;
- An **integer parameter** of kind `omp_memspace_handle_kind` for each predefined memory space in Table 2.8 on page 152;
- An **integer parameter** of kind `omp_allocator_handle_kind` for each predefined memory allocator in Table 2.10 on page 155;
- The **integer parameter** `omp_pause_resource_kind`;
- The **integer parameter** `omp_event_handle_kind`; and

- The **integer parameter `openmp_version`** with a value `yyyymm` where `yyyy` and `mm` are the year and month designations of the version of the OpenMP Fortran API that the implementation supports; this value matches that of the C preprocessor macro `__OPENMP`, when a macro preprocessor is supported (see Section 2.2 on page 49).

It is implementation defined whether any of the OpenMP runtime library routines that take an argument are extended with a generic interface so arguments of different **KIND** type can be accommodated.

Fortran

3.2 Execution Environment Routines

This section describes routines that affect and monitor threads, processors, and the parallel environment.

3.2.1 `omp_set_num_threads`

Summary

The **`omp_set_num_threads`** routine affects the number of threads to be used for subsequent parallel regions that do not specify a **`num_threads`** clause, by setting the value of the first element of the *nthreads-var* ICV of the current task.

Format

C / C++
`void omp_set_num_threads(int num_threads);`

C / C++
 Fortran
`subroutine omp_set_num_threads(num_threads)`
`integer num_threads`

Fortran

Constraints on Arguments

The value of the argument passed to this routine must evaluate to a positive integer, or else the behavior of this routine is implementation defined.

Binding

The binding task set for an **omp_set_num_threads** region is the generating task.

Effect

The effect of this routine is to set the value of the first element of the *nthreads-var* ICV of the current task to the value specified in the argument.

Cross References

- *nthreads-var* ICV, see Section 2.5 on page 63.
- **parallel** construct and **num_threads** clause, see Section 2.6 on page 74.
- Determining the number of threads for a **parallel** region, see Section 2.6.1 on page 78.
- **omp_get_num_threads** routine, see Section 3.2.2 on page 335.
- **omp_get_max_threads** routine, see Section 3.2.3 on page 336.
- **OMP_NUM_THREADS** environment variable, see Section 6.2 on page 602.

3.2.2 omp_get_num_threads

Summary

The **omp_get_num_threads** routine returns the number of threads in the current team.

Format

		C / C++	
	int omp_get_num_threads(void);		
		C / C++	
		Fortran	
	integer function omp_get_num_threads()		
		Fortran	

Binding

The binding region for an **omp_get_num_threads** region is the innermost enclosing **parallel** region.

Effect

The `omp_get_num_threads` routine returns the number of threads in the team that is executing the `parallel` region to which the routine region binds. If called from the sequential part of a program, this routine returns 1.

Cross References

- *nthreads-var* ICV, see Section 2.5 on page 63.
- `parallel` construct and `num_threads` clause, see Section 2.6 on page 74.
- Determining the number of threads for a `parallel` region, see Section 2.6.1 on page 78.
- `omp_set_num_threads` routine, see Section 3.2.1 on page 334.
- `OMP_NUM_THREADS` environment variable, see Section 6.2 on page 602.

3.2.3 `omp_get_max_threads`

Summary

The `omp_get_max_threads` routine returns an upper bound on the number of threads that could be used to form a new team if a `parallel` construct without a `num_threads` clause were encountered after execution returns from this routine.

Format

▼	C / C++	▼
<code>int omp_get_max_threads(void);</code>		
▲	C / C++	▲
▼	Fortran	▼
<code>integer function omp_get_max_threads()</code>		
▲	Fortran	▲

Binding

The binding task set for an `omp_get_max_threads` region is the generating task.

Effect

The value returned by `omp_get_max_threads` is the value of the first element of the *nthreads-var* ICV of the current task. This value is also an upper bound on the number of threads that could be used to form a new team if a parallel region without a `num_threads` clause were encountered after execution returns from this routine.

Note – The return value of the `omp_get_max_threads` routine can be used to allocate sufficient storage dynamically for all threads in the team formed at the subsequent active `parallel` region.

Cross References

- *nthreads-var* ICV, see Section 2.5 on page 63.
- `parallel` construct and `num_threads` clause, see Section 2.6 on page 74.
- Determining the number of threads for a `parallel` region, see Section 2.6.1 on page 78.
- `omp_set_num_threads` routine, see Section 3.2.1 on page 334.
- `omp_get_num_threads` routine, see Section 3.2.2 on page 335.
- `omp_get_thread_num` routine, see Section 3.2.4 on page 337.
- `OMP_NUM_THREADS` environment variable, see Section 6.2 on page 602.

3.2.4 `omp_get_thread_num`

Summary

The `omp_get_thread_num` routine returns the thread number, within the current team, of the calling thread.

Format

C / C++

`int omp_get_thread_num(void);`

C / C++

Fortran

`integer function omp_get_thread_num()`

Fortran

1 **Binding**

2 The binding thread set for an **omp_get_thread_num** region is the current team. The binding
3 region for an **omp_get_thread_num** region is the innermost enclosing **parallel** region.

4 **Effect**

5 The **omp_get_thread_num** routine returns the thread number of the calling thread, within the
6 team that is executing the **parallel** region to which the routine region binds. The thread number
7 is an integer between 0 and one less than the value returned by **omp_get_num_threads**,
8 inclusive. The thread number of the master thread of the team is 0. The routine returns 0 if it is
9 called from the sequential part of a program.

10 ▼
11 Note – The thread number may change during the execution of an untied task. The value returned
12 by **omp_get_thread_num** is not generally useful during the execution of such a task region.
13 ▲

14 **Cross References**

- 15 • *nthreads-var* ICV, see Section 2.5 on page 63.
- 16 • **parallel** construct and **num_threads** clause, see Section 2.6 on page 74.
- 17 • Determining the number of threads for a **parallel** region, see Section 2.6.1 on page 78.
- 18 • **omp_set_num_threads** routine, see Section 3.2.1 on page 334.
- 19 • **omp_get_num_threads** routine, see Section 3.2.2 on page 335.
- 20 • **OMP_NUM_THREADS** environment variable, see Section 6.2 on page 602.

21 **3.2.5 omp_get_num_procs**

22 **Summary**

23 The **omp_get_num_procs** routine returns the number of processors available to the device.

24 **Format**

25 ▼ C / C++ ▼
25 | **int omp_get_num_procs(void);** |
26 ▲ C / C++ ▲
26 ▼ Fortran ▼
26 | **integer function omp_get_num_procs()** |
26 ▲ Fortran ▲

Binding

The binding thread set for an `omp_get_num_procs` region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Effect

The `omp_get_num_procs` routine returns the number of processors that are available to the device at the time the routine is called. This value may change between the time that it is determined by the `omp_get_num_procs` routine and the time that it is read in the calling context due to system actions outside the control of the OpenMP implementation.

Cross References

- `omp_get_num_places` routine, see Section 3.2.24 on page 358.
- `omp_get_place_num_procs` routine, see Section 3.2.25 on page 359.
- `omp_get_place_proc_ids` routine, see Section 3.2.26 on page 360.
- `omp_get_place_num` routine, see Section 3.2.27 on page 362.

3.2.6 `omp_in_parallel`

Summary

The `omp_in_parallel` routine returns *true* if the *active-levels-var* ICV is greater than zero; otherwise, it returns *false*.

Format

		C / C++	
	<code>int omp_in_parallel(void);</code>		
		C / C++	
		Fortran	
	<code>logical function omp_in_parallel()</code>		
		Fortran	

Binding

The binding task set for an `omp_in_parallel` region is the generating task.

Effect

The effect of the `omp_in_parallel` routine is to return *true* if the current task is enclosed by an active `parallel` region, and the `parallel` region is enclosed by the outermost initial task region on the device; otherwise it returns *false*.

Cross References

- *active-levels-var*, see Section 2.5 on page 63.
- `parallel` construct, see Section 2.6 on page 74.
- `omp_get_num_threads` routine, see Section 3.2.2 on page 335.
- `omp_get_active_level` routine, see Section 3.2.21 on page 355.

3.2.7 `omp_set_dynamic`

Summary

The `omp_set_dynamic` routine enables or disables dynamic adjustment of the number of threads available for the execution of subsequent `parallel` regions by setting the value of the *dyn-var* ICV.

Format

		C / C++	
	<code>void omp_set_dynamic(int <i>dynamic_threads</i>);</code>		
		C / C++	
		Fortran	
	<code>subroutine omp_set_dynamic(<i>dynamic_threads</i>)</code> <code>logical <i>dynamic_threads</i></code>		
		Fortran	

Binding

The binding task set for an `omp_set_dynamic` region is the generating task.

Effect

For implementations that support dynamic adjustment of the number of threads, if the argument to `omp_set_dynamic` evaluates to *true*, dynamic adjustment is enabled for the current task; otherwise, dynamic adjustment is disabled for the current task. For implementations that do not support dynamic adjustment of the number of threads, this routine has no effect: the value of *dyn-var* remains *false*.

Cross References

- *dyn-var* ICV, see Section 2.5 on page 63.
- Determining the number of threads for a `parallel` region, see Section 2.6.1 on page 78.
- `omp_get_num_threads` routine, see Section 3.2.2 on page 335.
- `omp_get_dynamic` routine, see Section 3.2.8 on page 341.
- `OMP_DYNAMIC` environment variable, see Section 6.3 on page 603.

3.2.8 omp_get_dynamic

Summary

The `omp_get_dynamic` routine returns the value of the *dyn-var* ICV, which determines whether dynamic adjustment of the number of threads is enabled or disabled.

Format

		C / C++	
	<code>int omp_get_dynamic(void);</code>		
		C / C++	
		Fortran	
	<code>logical function omp_get_dynamic()</code>		
		Fortran	

Binding

The binding task set for an `omp_get_dynamic` region is the generating task.

Effect

This routine returns *true* if dynamic adjustment of the number of threads is enabled for the current task; it returns *false*, otherwise. If an implementation does not support dynamic adjustment of the number of threads, then this routine always returns *false*.

Cross References

- *dyn-var* ICV, see Section 2.5 on page 63.
- Determining the number of threads for a **parallel** region, see Section 2.6.1 on page 78.
- **omp_set_dynamic** routine, see Section 3.2.7 on page 340.
- **OMP_DYNAMIC** environment variable, see Section 6.3 on page 603.

3.2.9 omp_get_cancellation

Summary

The **omp_get_cancellation** routine returns the value of the *cancel-var* ICV, which determines if cancellation is enabled or disabled.

Format

		C / C++	
	int	omp_get_cancellation(void);	
		C / C++	
		Fortran	
	logical function	omp_get_cancellation()	
		Fortran	

Binding

The binding task set for an **omp_get_cancellation** region is the whole program.

Effect

This routine returns *true* if cancellation is enabled. It returns *false* otherwise.

Cross References

- *cancel-var* ICV, see Section 2.5.1 on page 64.
- **cancel** construct, see Section 2.18.1 on page 263.
- **OMP_CANCELLATION** environment variable, see Section 6.11 on page 610.

3.2.10 omp_set_nested

Summary

The deprecated **omp_set_nested** routine enables or disables nested parallelism by setting the *max-active-levels-var* ICV.

Format

C / C++

void omp_set_nested(int nested);

C / C++

Fortran

subroutine omp_set_nested(nested)

logical nested

Fortran

Binding

The binding task set for an **omp_set_nested** region is the generating task.

Effect

If the argument to **omp_set_nested** evaluates to *true*, the value of the *max-active-levels-var* ICV is set to the number of active levels of parallelism that the implementation supports; otherwise, if the value of *max-active-levels-var* is greater than 1 then it is set to 1. This routine has been deprecated.

Cross References

- *max-active-levels-var* ICV, see Section 2.5 on page 63.
- Determining the number of threads for a **parallel** region, see Section 2.6.1 on page 78.
- **omp_get_nested** routine, see Section 3.2.11 on page 344.
- **omp_set_max_active_levels** routine, see Section 3.2.16 on page 350.
- **omp_get_max_active_levels** routine, see Section 3.2.17 on page 351.
- **OMP_NESTED** environment variable, see Section 6.9 on page 609.

3.2.11 omp_get_nested

Summary

The deprecated **omp_get_nested** routine returns whether nested parallelism is enabled or disabled, according to the value of the *max-active-levels-var* ICV.

Format

		C / C++	
	int omp_get_nested(void);		
		C / C++	
		Fortran	
	logical function omp_get_nested()		
		Fortran	

Binding

The binding task set for an **omp_get_nested** region is the generating task.

Effect

This routine returns *true* if *max-active-levels-var* is greater than 1 for the current task; it returns *false*, otherwise. If an implementation does not support nested parallelism, this routine always returns *false*. This routine has been deprecated.

Cross References

- *max-active-levels-var* ICV, see Section 2.5 on page 63.
- Determining the number of threads for a **parallel** region, see Section 2.6.1 on page 78.
- **omp_set_nested** routine, see Section 3.2.10 on page 343.
- **omp_set_max_active_levels** routine, see Section 3.2.16 on page 350.
- **omp_get_max_active_levels** routine, see Section 3.2.17 on page 351.
- **OMP_NESTED** environment variable, see Section 6.9 on page 609.

3.2.12 omp_set_schedule

Summary

The **omp_set_schedule** routine affects the schedule that is applied when **runtime** is used as schedule kind, by setting the value of the *run-sched-var* ICV.

Format

C / C++

void omp_set_schedule(omp_sched_t kind, int chunk_size);

C / C++

Fortran

subroutine omp_set_schedule(kind, chunk_size)
integer (kind=omp_sched_kind) kind
integer chunk_size

Fortran

Constraints on Arguments

The first argument passed to this routine can be one of the valid OpenMP schedule kinds (except for **runtime**) or any implementation specific schedule. The C/C++ header file (**omp.h**) and the Fortran include file (**omp_lib.h**) and/or Fortran 90 module file (**omp_lib**) define the valid constants. The valid constants must include the following, which can be extended with implementation specific values:

C / C++

```
typedef enum omp_sched_t {  
    // schedule kinds  
    omp_sched_static = 0x1,  
    omp_sched_dynamic = 0x2,  
    omp_sched_guided = 0x3,  
    omp_sched_auto = 0x4,  
  
    // schedule modifier  
    omp_sched_monotonic = 0x80000000u  
} omp_sched_t;
```

C / C++
Fortran

```
! schedule kinds  
integer(kind=omp_sched_kind), &  
    parameter :: omp_sched_static = &  
        int(Z'1', kind=omp_sched_kind)  
integer(kind=omp_sched_kind), &  
    parameter :: omp_sched_dynamic = &  
        int(Z'2', kind=omp_sched_kind)  
integer(kind=omp_sched_kind), &  
    parameter :: omp_sched_guided = &  
        int(Z'3', kind=omp_sched_kind)  
integer(kind=omp_sched_kind), &  
    parameter :: omp_sched_auto = &  
        int(Z'4', kind=omp_sched_kind)  
  
! schedule modifier  
integer(kind=omp_sched_kind), &  
    parameter :: omp_sched_monotonic = &  
        int(Z'80000000', kind=omp_sched_kind)
```

Fortran

Binding

The binding task set for an **omp_set_schedule** region is the generating task.

Effect

The effect of this routine is to set the value of the *run-sched-var* ICV of the current task to the values specified in the two arguments. The schedule is set to the schedule kind that is specified by the first argument *kind*. It can be any of the standard schedule kinds or any other implementation specific one. For the schedule kinds **static**, **dynamic**, and **guided** the *chunk_size* is set to the value of the second argument, or to the default *chunk_size* if the value of the second argument is less than 1; for the schedule kind **auto** the second argument has no meaning; for implementation specific schedule kinds, the values and associated meanings of the second argument are implementation defined.

Each of the schedule kinds can be combined with the **omp_sched_monotonic** modifier by using the + or | operators in C/C++ or the + operator in Fortran. If the schedule kind is combined with the **omp_sched_monotonic** modifier, the schedule is modified as if the **monotonic** schedule modifier was specified. Otherwise, the schedule modifier is **nonmonotonic**.

Cross References

- *run-sched-var* ICV, see Section 2.5 on page 63.
- Determining the schedule of a worksharing-loop, see Section 2.9.2.1 on page 109.
- **omp_set_schedule** routine, see Section 3.2.12 on page 345.
- **omp_get_schedule** routine, see Section 3.2.13 on page 347.
- **OMP_SCHEDULE** environment variable, see Section 6.1 on page 601.

3.2.13 omp_get_schedule

Summary

The **omp_get_schedule** routine returns the schedule that is applied when the runtime schedule is used.

Format

```

C / C++
void omp_get_schedule(omp_sched_t *kind, int *chunk_size);

C / C++
Fortran
subroutine omp_get_schedule(kind, chunk_size)
integer (kind=omp_sched_kind) kind
integer chunk_size

Fortran
```

Binding

The binding task set for an **omp_get_schedule** region is the generating task.

Effect

This routine returns the *run-sched-var* ICV in the task to which the routine binds. The first argument *kind* returns the schedule to be used. It can be any of the standard schedule kinds as defined in Section 3.2.12 on page 345, or any implementation specific schedule kind. The second argument *chunk_size* returns the chunk size to be used, or a value less than 1 if the default chunk size is to be used, if the returned schedule kind is **static**, **dynamic**, or **guided**. The value returned by the second argument is implementation defined for any other schedule kinds.

Cross References

- *run-sched-var* ICV, see Section 2.5 on page 63.
- Determining the schedule of a worksharing-loop, see Section 2.9.2.1 on page 109.
- **omp_set_schedule** routine, see Section 3.2.12 on page 345.
- **OMP_SCHEDULE** environment variable, see Section 6.1 on page 601.

3.2.14 omp_get_thread_limit

Summary

The **omp_get_thread_limit** routine returns the maximum number of OpenMP threads available to participate in the current contention group.

Format

		C / C++	
	int	omp_get_thread_limit	(void);
		C / C++	
		Fortran	
	integer	function	omp_get_thread_limit
			()
		Fortran	

Binding

The binding thread set for an `omp_get_thread_limit` region is all threads on the device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Effect

The `omp_get_thread_limit` routine returns the value of the *thread-limit-var* ICV.

Cross References









- *thread-limit-var* ICV, see Section 2.5 on page 63.
- `omp_get_num_threads` routine, see Section 3.2.2 on page 335.
- `OMP_THREAD_LIMIT` environment variable, see Section 6.10 on page 610.
- `OMP_NUM_THREADS` environment variable, see Section 6.2 on page 602.

3.2.15 `omp_get_supported_active_levels`

Summary

The `omp_get_supported_active_levels` routine returns the number of active levels of parallelism supported by the implementation.

Format

			C / C++		
					
			Fortran		
					

```
int omp_get_supported_active_levels(void);  
  
integer function omp_get_supported_active_levels()
```

Binding

The binding task set for an `omp_get_supported_active_levels` region is the generating task.

1 **Effect**

2 The `omp_get_supported_active_levels` routine returns the number of active levels of
3 parallelism supported by the implementation. The *max-active-levels-var* ICV may not have a value
4 that is greater than this number. The value returned by the
5 **`omp_get_supported_active_levels`** routine is implementation defined, but it must be
6 greater than 0.

7 **Cross References**

- 8 • *max-active-levels-var* ICV, see Section 2.5 on page 63.
- 9 • `omp_get_max_active_levels` routine, see Section 3.2.17 on page 351.
- 10 • `omp_set_max_active_levels` routine, see Section 3.2.16 on page 350.

11 **3.2.16 `omp_set_max_active_levels`**

12 **Summary**

13 The `omp_set_max_active_levels` routine limits the number of nested active parallel
14 regions on the device, by setting the *max-active-levels-var* ICV

15 **Format**

16

C / C++

`void omp_set_max_active_levels(int max_levels);`

17

C / C++

`subroutine omp_set_max_active_levels(max_levels)`

18

Fortran

`integer max_levels`

19 **Constraints on Arguments**

20 The value of the argument passed to this routine must evaluate to a non-negative integer, otherwise
21 the behavior of this routine is implementation defined.

1 **Binding**

2 When called from a sequential part of the program, the binding thread set for an
3 **omp_set_max_active_levels** region is the encountering thread. When called from within
4 any **parallel** or **teams** region, the binding thread set (and binding region, if required) for the
5 **omp_set_max_active_levels** region is implementation defined.

6 **Effect**

7 The effect of this routine is to set the value of the *max-active-levels-var* ICV to the value specified
8 in the argument.

9 If the number of active levels requested exceeds the number of active levels of parallelism
10 supported by the implementation, the value of the *max-active-levels-var* ICV will be set to the
11 number of active levels supported by the implementation.

12 This routine has the described effect only when called from a sequential part of the program. When
13 called from within a **parallel** or **teams** region, the effect of this routine is implementation
14 defined.

15 **Cross References**

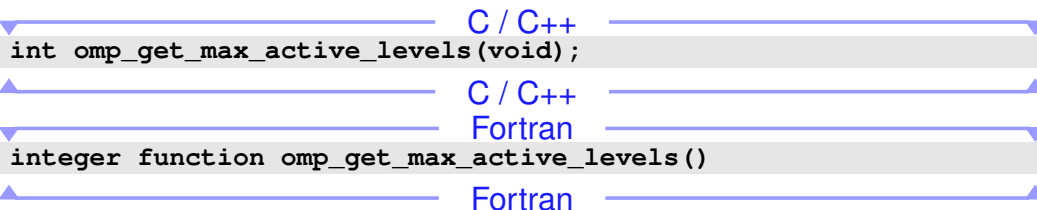
- 16 • *max-active-levels-var* ICV, see Section 2.5 on page 63.
- 17 • **parallel** construct, see Section 2.6 on page 74.
- 18 • **omp_get_supported_active_levels** routine, see Section 3.2.15 on page 349.
- 19 • **omp_get_max_active_levels** routine, see Section 3.2.17 on page 351.
- 20 • **OMP_MAX_ACTIVE_LEVELS** environment variable, see Section 6.8 on page 608.

21 **3.2.17 omp_get_max_active_levels**

22 **Summary**

23 The **omp_get_max_active_levels** routine returns the value of the *max-active-levels-var*
24 ICV, which determines the maximum number of nested active parallel regions on the device.

25 **Format**

26 
 C / C++
| int omp_get_max_active_levels(void);
 C / C++
 Fortran
| integer function omp_get_max_active_levels()
 Fortran

Binding

When called from a sequential part of the program, the binding thread set for an **omp_get_max_active_levels** region is the encountering thread. When called from within any **parallel** or **teams** region, the binding thread set (and binding region, if required) for the **omp_get_max_active_levels** region is implementation defined.

Effect

The **omp_get_max_active_levels** routine returns the value of the *max-active-levels-var* ICV, which determines the maximum number of nested active parallel regions on the device.

Cross References

- *max-active-levels-var* ICV, see Section 2.5 on page 63.
- **parallel** construct, see Section 2.6 on page 74.
- **omp_get_supported_active_levels** routine, see Section 3.2.15 on page 349.
- **omp_set_max_active_levels** routine, see Section 3.2.16 on page 350.
- **OMP_MAX_ACTIVE_LEVELS** environment variable, see Section 6.8 on page 608.

3.2.18 omp_get_level

Summary

The **omp_get_level** routine returns the value of the *levels-var* ICV.

Format

		C / C++	
	int omp_get_level(void);		
		C / C++	
		Fortran	
	integer function omp_get_level()		
		Fortran	

Binding

The binding task set for an **omp_get_level** region is the generating task.

Effect

The effect of the `omp_get_level` routine is to return the number of nested `parallel` regions (whether active or inactive) that enclose the current task such that all of the `parallel` regions are enclosed by the outermost initial task region on the current device.

Cross References

- *levels-var* ICV, see Section 2.5 on page 63.
- `parallel` construct, see Section 2.6 on page 74.
- `omp_get_active_level` routine, see Section 3.2.21 on page 355.
- `OMP_MAX_ACTIVE_LEVELS` environment variable, see Section 6.8 on page 608.

3.2.19 `omp_get_ancestor_thread_num`

Summary

The `omp_get_ancestor_thread_num` routine returns, for a given nested level of the current thread, the thread number of the ancestor of the current thread.

Format

C / C++

int omp_get_ancestor_thread_num(int level);

C / C++

Fortran

integer function omp_get_ancestor_thread_num(level)
integer level

Fortran

Binding

The binding thread set for an `omp_get_ancestor_thread_num` region is the encountering thread. The binding region for an `omp_get_ancestor_thread_num` region is the innermost enclosing `parallel` region.

Effect

The `omp_get_ancestor_thread_num` routine returns the thread number of the ancestor at a given nest level of the current thread or the thread number of the current thread. If the requested nest level is outside the range of 0 and the nest level of the current thread, as returned by the `omp_get_level` routine, the routine returns -1.

Note – When the `omp_get_ancestor_thread_num` routine is called with a value of `level=0`, the routine always returns 0. If `level=omp_get_level()`, the routine has the same effect as the `omp_get_thread_num` routine.

Cross References

- `parallel` construct, see Section 2.6 on page 74.
- `omp_get_num_threads` routine, see Section 3.2.2 on page 335.
- `omp_get_thread_num` routine, see Section 3.2.4 on page 337.
- `omp_get_level` routine, see Section 3.2.18 on page 352.
- `omp_get_team_size` routine, see Section 3.2.20 on page 354.

3.2.20 `omp_get_team_size`

Summary

The `omp_get_team_size` routine returns, for a given nested level of the current thread, the size of the thread team to which the ancestor or the current thread belongs.

Format

		C / C++	
	<code>int omp_get_team_size(int level);</code>		
		C / C++	
		Fortran	
	<code>integer function omp_get_team_size(level)</code>		
	<code>integer level</code>		
		Fortran	

1 **Binding**

2 The binding thread set for an `omp_get_team_size` region is the encountering thread. The
3 binding region for an `omp_get_team_size` region is the innermost enclosing **parallel**
4 region.

5 **Effect**

6 The `omp_get_team_size` routine returns the size of the thread team to which the ancestor or
7 the current thread belongs. If the requested nested level is outside the range of 0 and the nested
8 level of the current thread, as returned by the `omp_get_level` routine, the routine returns -1.
9 Inactive parallel regions are regarded like active parallel regions executed with one thread.

10 ▼
11 Note – When the `omp_get_team_size` routine is called with a value of `level=0`, the routine
12 always returns 1. If `level=omp_get_level()`, the routine has the same effect as the
13 `omp_get_num_threads` routine.
14 ▲

15 **Cross References**

- 16 • `omp_get_num_threads` routine, see Section 3.2.2 on page 335.
17 • `omp_get_level` routine, see Section 3.2.18 on page 352.
18 • `omp_get_ancestor_thread_num` routine, see Section 3.2.19 on page 353.

19 **3.2.21 omp_get_active_level**

20 **Summary**

21 The `omp_get_active_level` routine returns the value of the *active-level-vars* ICV..

22 **Format**

23 ▼ C / C++ ▼
24 | `int omp_get_active_level(void);`
25 ▲ C / C++ ▲
26 ▼ Fortran ▼
27 | `integer function omp_get_active_level()`
28 ▲ Fortran ▲

Binding

The binding task set for the an **omp_get_active_level** region is the generating task.

Effect

The effect of the **omp_get_active_level** routine is to return the number of nested active **parallel** regions enclosing the current task such that all of the **parallel** regions are enclosed by the outermost initial task region on the current device.

Cross References

- *active-levels-var* ICV, see Section 2.5 on page 63.
- **omp_get_level** routine, see Section 3.2.18 on page 352.
- **omp_set_max_active_levels** routine, see Section 3.2.16 on page 350.
- **omp_get_max_active_levels** routine, see Section 3.2.17 on page 351.
- **OMP_MAX_ACTIVE_LEVELS** environment variable, see Section 6.8 on page 608.

3.2.22 omp_in_final

Summary

The **omp_in_final** routine returns *true* if the routine is executed in a final task region; otherwise, it returns *false*.

Format

		C / C++	
	int omp_in_final(void);		
		C / C++	
		Fortran	
	logical function omp_in_final()		
		Fortran	

Binding

The binding task set for an **omp_in_final** region is the generating task.

Effect

`omp_in_final` returns *true* if the enclosing task region is final. Otherwise, it returns *false*.

Cross References

- `task` construct, see Section 2.10.1 on page 135.

3.2.23 `omp_get_proc_bind`

Summary

The `omp_get_proc_bind` routine returns the thread affinity policy to be used for the subsequent nested `parallel` regions that do not specify a `proc_bind` clause.

Format

		C / C++	
	<code>omp_proc_bind_t</code>	<code>omp_get_proc_bind(void);</code>	
		C / C++	
		Fortran	
	<code>integer (kind=omp_proc_bind_kind)</code>	<code>function omp_get_proc_bind()</code>	
		Fortran	

Constraints on Arguments

The value returned by this routine must be one of the valid affinity policy kinds. The C/C++ header file (`omp.h`) and the Fortran include file (`omp_lib.h`) and/or Fortran 90 module file (`omp_lib`) define the valid constants. The valid constants must include the following:

		C / C++	
	<code>typedef enum omp_proc_bind_t {</code>		
	<code>omp_proc_bind_false = 0,</code>		
	<code>omp_proc_bind_true = 1,</code>		
	<code>omp_proc_bind_master = 2,</code>		
	<code>omp_proc_bind_close = 3,</code>		
	<code>omp_proc_bind_spread = 4</code>		
	<code>} omp_proc_bind_t;</code>		
		C / C++	

Fortran

```
1 integer (kind=omp_proc_bind_kind), &  
2     parameter :: omp_proc_bind_false = 0  
3 integer (kind=omp_proc_bind_kind), &  
4     parameter :: omp_proc_bind_true = 1  
5 integer (kind=omp_proc_bind_kind), &  
6     parameter :: omp_proc_bind_master = 2  
7 integer (kind=omp_proc_bind_kind), &  
8     parameter :: omp_proc_bind_close = 3  
9 integer (kind=omp_proc_bind_kind), &  
10    parameter :: omp_proc_bind_spread = 4
```

Fortran

Binding

The binding task set for an **omp_get_proc_bind** region is the generating task.

Effect

The effect of this routine is to return the value of the first element of the *bind-var* ICV of the current task. See Section 2.6.2 on page 80 for the rules that govern the thread affinity policy.

Cross References

- *bind-var* ICV, see Section 2.5 on page 63.
- Controlling OpenMP thread affinity, see Section 2.6.2 on page 80.
- **omp_get_num_places** routine, see Section 3.2.24 on page 358.
- **OMP_PROC_BIND** environment variable, see Section 6.4 on page 604.
- **OMP_PLACES** environment variable, see Section 6.5 on page 605.

3.2.24 omp_get_num_places

Summary

The **omp_get_num_places** routine returns the number of places available to the execution environment in the place list.

Format

Binding

The binding thread set for an **omp_get_num_places** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Effect

The **omp_get_num_places** routine returns the number of places in the place list. This value is equivalent to the number of places in the *place-partition-var* ICV in the execution environment of the initial task.

Cross References

- *place-partition-var* ICV, see Section 2.5 on page 63.
- Controlling OpenMP thread affinity, see Section 2.6.2 on page 80.
- **omp_get_place_num** routine, see Section 3.2.27 on page 362.
- **OMP_PLACES** environment variable, see Section 6.5 on page 605.

3.2.25 omp_get_place_num_procs

Summary

The **omp_get_place_num_procs** routine returns the number of processors available to the execution environment in the specified place.

Format

C / C++

```
int omp_get_place_num_procs(int place_num);
```

C / C++

Fortran

```
integer function omp_get_place_num_procs(place_num)  
integer place_num
```

Fortran

Binding

The binding thread set for an **omp_get_place_num_procs** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Effect

The **omp_get_place_num_procs** routine returns the number of processors associated with the place numbered *place_num*. The routine returns zero when *place_num* is negative, or is greater than or equal to the value returned by **omp_get_num_places()**.

Cross References

- *place-partition-var* ICV, see Section 2.5 on page 63.
- Controlling OpenMP thread affinity, see Section 2.6.2 on page 80.
- **omp_get_num_places** routine, see Section 3.2.24 on page 358.
- **omp_get_place_proc_ids** routine, see Section 3.2.26 on page 360.
- **OMP_PLACES** environment variable, see Section 6.5 on page 605.

3.2.26 omp_get_place_proc_ids

Summary

The **omp_get_place_proc_ids** routine returns the numerical identifiers of the processors available to the execution environment in the specified place.

Format

C / C++
void omp_get_place_proc_ids(int *place_num*, int **ids*);

C / C++
Fortran
subroutine omp_get_place_proc_ids(*place_num*, *ids*)
integer *place_num*
integer *ids* (*)

Binding

The binding thread set for an **omp_get_place_proc_ids** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Effect

The **omp_get_place_proc_ids** routine returns the numerical identifiers of each processor associated with the place numbered *place_num*. The numerical identifiers are non-negative, and their meaning is implementation defined. The numerical identifiers are returned in the array *ids* and their order in the array is implementation defined. The array must be sufficiently large to contain **omp_get_place_num_procs(*place_num*)** integers; otherwise, the behavior is unspecified. The routine has no effect when *place_num* has a negative value, or a value greater than or equal to **omp_get_num_places()**.

Cross References

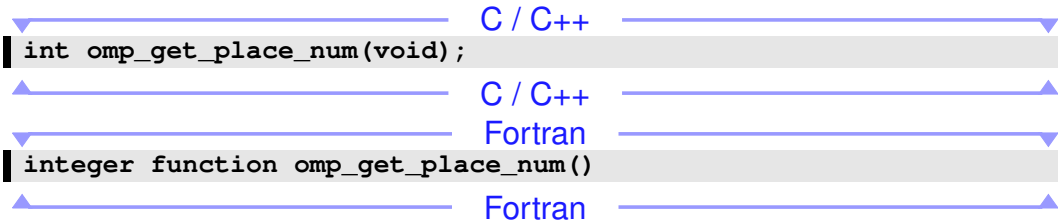
- *place-partition-var* ICV, see Section 2.5 on page 63.
- Controlling OpenMP thread affinity, see Section 2.6.2 on page 80.
- **omp_get_num_places** routine, see Section 3.2.24 on page 358.
- **omp_get_place_num_procs** routine, see Section 3.2.25 on page 359.
- **OMP_PLACES** environment variable, see Section 6.5 on page 605.

1 3.2.27 `omp_get_place_num`

2 Summary

3 The `omp_get_place_num` routine returns the place number of the place to which the
4 encountering thread is bound.

5 Format

6 
7 `int omp_get_place_num(void);`
8 `integer function omp_get_place_num()`

8 Binding

9 The binding thread set for an `omp_get_place_num` region is the encountering thread.

10 Effect

11 When the encountering thread is bound to a place, the `omp_get_place_num` routine returns the
12 place number associated with the thread. The returned value is between 0 and one less than the
13 value returned by `omp_get_num_places()`, inclusive. When the encountering thread is not
14 bound to a place, the routine returns -1.

15 Cross References









- 16 • *place-partition-var* ICV, see Section 2.5 on page 63.
17 • Controlling OpenMP thread affinity, see Section 2.6.2 on page 80.
18 • `omp_get_num_places` routine, see Section 3.2.24 on page 358.
19 • `OMP_PLACES` environment variable, see Section 6.5 on page 605.

20 3.2.28 `omp_get_partition_num_places`

21 Summary

22 The `omp_get_partition_num_places` routine returns the number of places in the place
23 partition of the innermost implicit task.

Format

		
<code>int omp_get_partition_num_places(void);</code>		
		
		
<code>integer function omp_get_partition_num_places()</code>		
		

Binding

The binding task set for an `omp_get_partition_num_places` region is the encountering implicit task.

Effect

The `omp_get_partition_num_places` routine returns the number of places in the *place-partition-var* ICV.

Cross References

- *place-partition-var* ICV, see Section [2.5](#) on page [63](#).
- Controlling OpenMP thread affinity, see Section [2.6.2](#) on page [80](#).
- `omp_get_num_places` routine, see Section [3.2.24](#) on page [358](#).
- `OMP_PLACES` environment variable, see Section [6.5](#) on page [605](#).

3.2.29 `omp_get_partition_place_nums`

Summary

The `omp_get_partition_place_nums` routine returns the list of place numbers corresponding to the places in the *place-partition-var* ICV of the innermost implicit task.

Format

C / C++

```
void omp_get_partition_place_nums(int *place_nums);
```

C / C++

Fortran

```
subroutine omp_get_partition_place_nums(place_nums)  
integer place_nums(*)
```

Fortran

Binding

The binding task set for an **omp_get_partition_place_nums** region is the encountering implicit task.

Effect

The **omp_get_partition_place_nums** routine returns the list of place numbers that correspond to the places in the *place-partition-var* ICV of the innermost implicit task. The array must be sufficiently large to contain **omp_get_partition_num_places()** integers; otherwise, the behavior is unspecified.

Cross References

- *place-partition-var* ICV, see Section 2.5 on page 63.
- Controlling OpenMP thread affinity, see Section 2.6.2 on page 80.
- **omp_get_partition_num_places** routine, see Section 3.2.28 on page 362.
- **OMP_PLACES** environment variable, see Section 6.5 on page 605.

3.2.30 omp_set_affinity_format

Summary

The **omp_set_affinity_format** routine sets the affinity format to be used on the device by setting the value of the *affinity-format-var* ICV.

Format

		C / C++	
	<code>void omp_set_affinity_format(const char *format);</code>		
		C / C++	
		Fortran	
	<code>subroutine omp_set_affinity_format (format)</code>		
	<code>character (len=*) , intent (in) :: format</code>		
		Fortran	

Binding

When called from a sequential part of the program, the binding thread set for an `omp_set_affinity_format` region is the encountering thread. When called from within any `parallel` or `teams` region, the binding thread set (and binding region, if required) for the `omp_set_affinity_format` region is implementation defined.

Effect

The effect of `omp_set_affinity_format` routine is to copy the character string specified by the *format* argument into the *affinity-format-var* ICV on the current device.

This routine has the described effect only when called from a sequential part of the program. When called from within a `parallel` or `teams` region, the effect of this routine is implementation defined.

Cross References

- Controlling OpenMP thread affinity, see Section [2.6.2](#) on page [80](#).
- `omp_get_affinity_format` routine, see Section [3.2.31](#) on page [366](#).
- `omp_display_affinity` routine, see Section [3.2.32](#) on page [367](#).
- `omp_capture_affinity` routine, see Section [3.2.33](#) on page [368](#).
- `OMP_DISPLAY_AFFINITY` environment variable, see Section [6.13](#) on page [612](#).
- `OMP_AFFINITY_FORMAT` environment variable, see Section [6.14](#) on page [613](#).

1 **3.2.31 omp_get_affinity_format**

2 **Summary**

3 The **omp_get_affinity_format** routine returns the value of the *affinity-format-var* ICV on
4 the device.

5 **Format**

C / C++

6 | **size_t omp_get_affinity_format(char *buffer, size_t size);**

C / C++

Fortran

7 | **integer function omp_get_affinity_format(buffer)**
8 | **character(len=*) ,intent(out) :: buffer**

Fortran

9 **Binding**

10 When called from a sequential part of the program, the binding thread set for an
11 **omp_get_affinity_format** region is the encountering thread. When called from within any
12 **parallel** or **teams** region, the binding thread set (and binding region, if required) for the
13 **omp_get_affinity_format** region is implementation defined.

14 **Effect**

C / C++

15 | The **omp_get_affinity_format** routine returns the number of characters in the
16 | *affinity-format-var* ICV on the current device, excluding the terminating null byte (' \0 ') and if
17 | *size* is non-zero, writes the value of the *affinity-format-var* ICV on the current device to *buffer*
18 | followed by a null byte. If the return value is larger or equal to *size*, the affinity format specification
19 | is truncated, with the terminating null byte stored to *buffer[size-1]*. If *size* is zero, nothing is
20 | stored and *buffer* may be **NULL**.

C / C++

Fortran

21 The **omp_get_affinity_format** routine returns the number of characters that are required to
22 hold the *affinity-format-var* ICV on the current device and writes the value of the
23 *affinity-format-var* ICV on the current device to *buffer*. If the return value is larger than
24 **len(buffer)**, the affinity format specification is truncated.

Fortran

25 | If the *buffer* argument does not conform to the specified format then the result is implementation
26 | defined.

1 **Cross References**

- 2 • Controlling OpenMP thread affinity, see Section 2.6.2 on page 80.
- 3 • `omp_set_affinity_format` routine, see Section 3.2.30 on page 364.
- 4 • `omp_display_affinity` routine, see Section 3.2.32 on page 367.
- 5 • `omp_capture_affinity` routine, see Section 3.2.33 on page 368.
- 6 • `OMP_DISPLAY_AFFINITY` environment variable, see Section 6.13 on page 612.
- 7 • `OMP_AFFINITY_FORMAT` environment variable, see Section 6.14 on page 613.

8 **3.2.32 omp_display_affinity**

9 **Summary**

10 The `omp_display_affinity` routine prints the OpenMP thread affinity information using the

11 format specification provided.

12 **Format**

13

C / C++

```
void omp_display_affinity(const char *format);
```

14

C / C++

15

Fortran

```
subroutine omp_display_affinity(format)
character(len=*) , intent(in) :: format
```

16

Fortran

16 **Binding**

17 The binding thread set for an `omp_display_affinity` region is the encountering thread.

18 **Effect**

19 The `omp_display_affinity` routine prints the thread affinity information of the current

20 thread in the format specified by the *format* argument, followed by a *new-line*. If the *format* is

21 **NULL** (for C/C++) or a zero-length string (for Fortran and C/C++), the value of the

22 *affinity-format-var* ICV is used. If the *format* argument does not conform to the specified format

23 then the result is implementation defined.

Cross References

- Controlling OpenMP thread affinity, see Section 2.6.2 on page 80.
- `omp_set_affinity_format` routine, see Section 3.2.30 on page 364.
- `omp_get_affinity_format` routine, see Section 3.2.31 on page 366.
- `omp_capture_affinity` routine, see Section 3.2.33 on page 368.
- `OMP_DISPLAY_AFFINITY` environment variable, see Section 6.13 on page 612.
- `OMP_AFFINITY_FORMAT` environment variable, see Section 6.14 on page 613.

3.2.33 `omp_capture_affinity`

Summary

The `omp_capture_affinity` routine prints the OpenMP thread affinity information into a buffer using the format specification provided.

Format

C / C++

```
size_t omp_capture_affinity(  
    char *buffer,  
    size_t size,  
    const char *format  
);
```

C / C++

Fortran

```
integer function omp_capture_affinity(buffer,format)  
character(len=*) ,intent(out) :: buffer  
character(len=*) ,intent(in)  :: format
```

Fortran

Binding

The binding thread set for an `omp_capture_affinity` region is the encountering thread.

Effect

C / C++

The **omp_capture_affinity** routine returns the number of characters in the entire thread affinity information string excluding the terminating null byte ('`\0`') and if *size* is non-zero, writes the thread affinity information of the current thread in the format specified by the *format* argument into the character string **buffer** followed by a null byte. If the return value is larger or equal to *size*, the thread affinity information string is truncated, with the terminating null byte stored to **buffer[size-1]**. If *size* is zero, nothing is stored and *buffer* may be **NULL**. If the *format* is **NULL** or a zero-length string, the value of the *affinity-format-var* ICV is used.

C / C++

Fortran

The **omp_capture_affinity** routine returns the number of characters required to hold the entire thread affinity information string and prints the thread affinity information of the current thread into the character string **buffer** with the size of **len(buffer)** in the format specified by the *format* argument. If the *format* is a zero-length string, the value of the *affinity-format-var* ICV is used. If the return value is larger than **len(buffer)**, the thread affinity information string is truncated. If the *format* is a zero-length string, the value of the *affinity-format-var* ICV is used.

Fortran

If the *format* argument does not conform to the specified format then the result is implementation defined.

Cross References

- Controlling OpenMP thread affinity, see Section 2.6.2 on page 80.
- **omp_set_affinity_format** routine, see Section 3.2.30 on page 364.
- **omp_get_affinity_format** routine, see Section 3.2.31 on page 366.
- **omp_display_affinity** routine, see Section 3.2.32 on page 367.
- **OMP_DISPLAY_AFFINITY** environment variable, see Section 6.13 on page 612.
- **OMP_AFFINITY_FORMAT** environment variable, see Section 6.14 on page 613.

3.2.34 omp_set_default_device

Summary

The **omp_set_default_device** routine controls the default target device by assigning the value of the *default-device-var* ICV.

Format

C / C++

```
void omp_set_default_device(int device_num);
```

C / C++

Fortran

```
subroutine omp_set_default_device(device_num)  
integer device_num
```

Fortran

Binding

The binding task set for an **omp_set_default_device** region is the generating task.

Effect

The effect of this routine is to set the value of the *default-device-var* ICV of the current task to the value specified in the argument. When called from within a **target** region the effect of this routine is unspecified.

Cross References

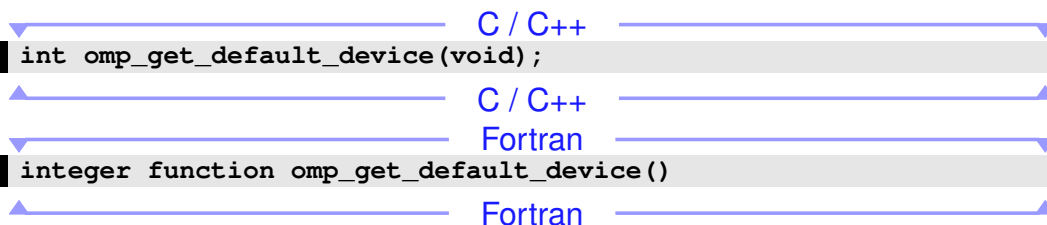
- *default-device-var*, see Section 2.5 on page 63.
- **target** construct, see Section 2.12.5 on page 170
- **omp_get_default_device**, see Section 3.2.35 on page 370.
- **OMP_DEFAULT_DEVICE** environment variable, see Section 6.15 on page 615

3.2.35 omp_get_default_device

Summary

The **omp_get_default_device** routine returns the default target device.

Format



Binding

The binding task set for an **omp_get_default_device** region is the generating task.

Effect

The **omp_get_default_device** routine returns the value of the *default-device-var* ICV of the current task. When called from within a **target** region the effect of this routine is unspecified.

Cross References

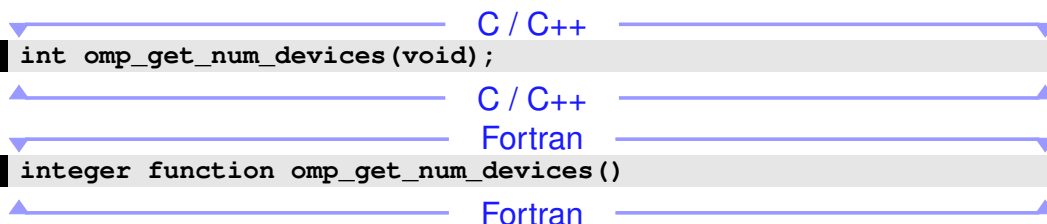
- *default-device-var*, see Section 2.5 on page 63.
- **target** construct, see Section 2.12.5 on page 170
- **omp_set_default_device**, see Section 3.2.34 on page 369.
- **OMP_DEFAULT_DEVICE** environment variable, see Section 6.15 on page 615.

3.2.36 omp_get_num_devices

Summary

The **omp_get_num_devices** routine returns the number of target devices.

Format



Binding

The binding task set for an `omp_get_num_devices` region is the generating task.

Effect

The `omp_get_num_devices` routine returns the number of available target devices. When called from within a `target` region the effect of this routine is unspecified.

Cross References

- `target` construct, see Section 2.12.5 on page 170
- `omp_get_default_device`, see Section 3.2.35 on page 370.
- `omp_get_device_num`, see Section 3.2.37 on page 372.

3.2.37 `omp_get_device_num`

Summary

The `omp_get_device_num` routine returns the device number of the device on which the calling thread is executing.

Format

	C / C++
<code>int omp_get_device_num(void);</code>	
	C / C++
	Fortran
<code>integer function omp_get_device_num()</code>	
	Fortran

Binding

The binding task set for an `omp_get_devices_num` region is the generating task.

Effect

The `omp_get_device_num` routine returns the device number of the device on which the calling thread is executing. When called on the host device, it will return the same value as the `omp_get_initial_device` routine.

Cross References

- **target** construct, see Section 2.12.5 on page 170
- **omp_get_default_device**, see Section 3.2.35 on page 370.
- **omp_get_num_devices**, see Section 3.2.36 on page 371.
- **omp_get_initial_device** routine, see Section 3.2.41 on page 376.

3.2.38 omp_get_num_teams

Summary

The **omp_get_num_teams** routine returns the number of initial teams in the current **teams** region.

Format

		C / C++	
	<code>int omp_get_num_teams(void);</code>		
		C / C++	
		Fortran	
	<code>integer function omp_get_num_teams()</code>		
		Fortran	

Binding

The binding task set for an **omp_get_num_teams** region is the generating task

Effect

The effect of this routine is to return the number of initial teams in the current **teams** region. The routine returns 1 if it is called from outside of a **teams** region.

Cross References

- **teams** construct, see Section 2.7 on page 82.
- **target** construct, see Section 2.12.5 on page 170.
- **omp_get_team_num** routine, see Section 3.2.39 on page 374.

1 **3.2.39 omp_get_team_num**

2 **Summary**

3 The `omp_get_team_num` routine returns the initial team number of the calling thread.

4 **Format**

5	<code>int omp_get_team_num(void);</code>	C / C++
6	<code>integer function omp_get_team_num()</code>	Fortran

7 **Binding**

8 The binding task set for an `omp_get_team_num` region is the generating task.

9 **Effect**

10 The `omp_get_team_num` routine returns the initial team number of the calling thread. The
11 initial team number is an integer between 0 and one less than the value returned by
12 `omp_get_num_teams()`, inclusive. The routine returns 0 if it is called outside of a **teams**
13 region.

14 **Cross References**

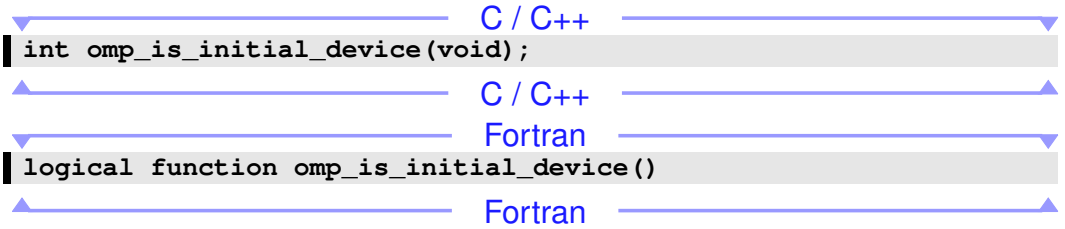
- 15 • **teams** construct, see Section 2.7 on page 82.
- 16 • **target** construct, see Section 2.12.5 on page 170
- 17 • `omp_get_num_teams` routine, see Section 3.2.38 on page 373.

1 3.2.40 `omp_is_initial_device`

2 Summary

3 The `omp_is_initial_device` routine returns *true* if the current task is executing on the host
4 device; otherwise, it returns *false*.

5 Format

6 
7 `int omp_is_initial_device(void);`
`logical function omp_is_initial_device()`

8 Binding

9 The binding task set for an `omp_is_initial_device` region is the generating task.

10 Effect

11 The effect of this routine is to return *true* if the current task is executing on the host device;
12 otherwise, it returns *false*.

13 Cross References

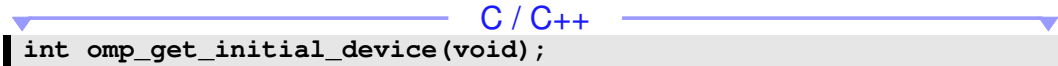
- 14 • `omp_get_get_initial_device` routine, see Section [3.2.41](#) on page [376](#).
15 • Device memory routines, see Section [3.6](#) on page [397](#).

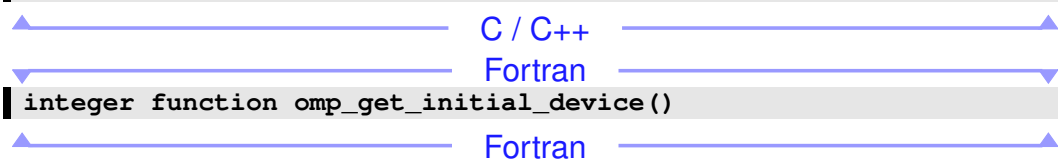
1 **3.2.41 omp_get_initial_device**

2 **Summary**

3 The `omp_get_initial_device` routine returns a device number that represents the host
4 device.

5 **Format**

6  `int omp_get_initial_device(void);`

7  `integer function omp_get_initial_device()`

8 **Binding**

9 The binding task set for an `omp_get_initial_device` region is the generating task.

10 **Effect**

11 The effect of this routine is to return the device number of the host device. The value of the device
12 number is implementation defined. When called from within a **target** region the effect of this
13 routine is unspecified.

14 **Cross References**

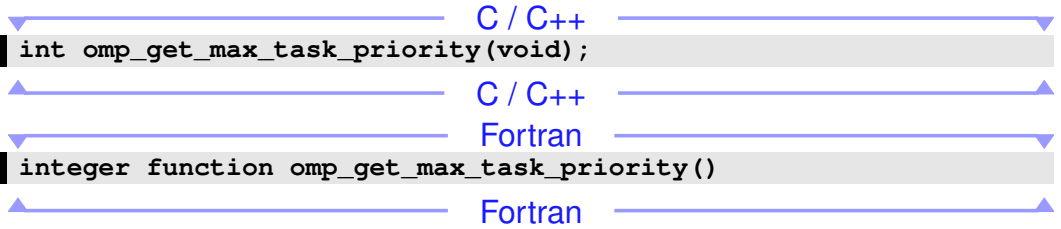







- 15 • **target** construct, see Section [2.12.5](#) on page [170](#).
- 16 • `omp_is_initial_device` routine, see Section [3.2.40](#) on page [375](#).
- 17 • Device memory routines, see Section [3.6](#) on page [397](#).

1 3.2.42 omp_get_max_task_priority

2 Summary

3 The `omp_get_max_task_priority` routine returns the maximum value that can be specified
4 in the `priority` clause.

5 Format

		C / C++		
6	<code>int omp_get_max_task_priority(void);</code>			
		C / C++		
		Fortran		
7	<code>integer function omp_get_max_task_priority()</code>			
		Fortran		

8 Binding

9 The binding thread set for an `omp_get_max_task_priority` region is all threads on the
10 device. The effect of executing this routine is not related to any specific region that corresponds to
11 any construct or API routine.

12 Effect

13 The `omp_get_max_task_priority` routine returns the value of the *max-task-priority-var*
14 ICV, which determines the maximum value that can be specified in the `priority` clause.

15 Cross References


- 16 • *max-task-priority-var*, see Section 2.5 on page 63.
- 17 • `task` construct, see Section 2.10.1 on page 135.


1 3.2.43 omp_pause_resource

2 Summary

3 The **omp_pause_resource** routine allows the runtime to relinquish resources used by OpenMP
4 on the specified device.

5 Format


6  C / C++
7 `int omp_pause_resource(
8 omp_pause_resource_t kind,
9 int device_num
10);`


11  Fortran
12 `integer function omp_pause_resource(kind, device_num)
13 integer (kind=omp_pause_resource_kind) kind
14 integer device_num`

13 Constraints on Arguments

14 The first argument passed to this routine can be one of the valid OpenMP pause kind, or any
15 implementation specific pause kind. The C/C++ header file (**omp.h**) and the Fortran include file
16 (**omp_lib.h**) and/or Fortran 90 module file (**omp_lib**) define the valid constants. The valid
17 constants must include the following, which can be extended with implementation specific values:

18 Format

19  C / C++
20 `typedef enum omp_pause_resource_t {
21 omp_pause_soft = 1,
22 omp_pause_hard = 2
23 } omp_pause_resource_t;`

24  Fortran
25 `integer (kind=omp_pause_resource_kind), parameter :: &
26 omp_pause_soft = 1
27 integer (kind=omp_pause_resource_kind), parameter :: &
28 omp_pause_hard = 2`

The second argument passed to this routine indicates the device that will be paused. The **device_num** parameter must be greater than or equal to zero and less than the result of **omp_get_num_devices()** or equal to the result of a call to **omp_get_initial_device()**.

Binding

The binding task set for an **omp_pause_resource** region is the whole program.

Effect

The **omp_pause_resource** routine allows the runtime to relinquish resources used by OpenMP on the specified device.

If successful, the **omp_pause_hard** value results in a hard pause for which the OpenMP state is not guaranteed to persist across the **omp_pause_resource** call. A hard pause may relinquish any data allocated by OpenMP on a given device, including data allocated by memory routines for that device as well as data present on the device as a result of a **declare target** or **target data** construct. A hard pause may also relinquish any data associated with a **threadprivate** directive. When relinquished and when applicable, base language appropriate deallocation/finalization is performed. When relinquished and when applicable, mapped data on a device will not be copied back from the device to the host.

If successful, the **omp_pause_soft** value results in a soft pause for which the OpenMP state is guaranteed to persist across the call, with the exception of any data associated with a **threadprivate** directive, which may be relinquished across the call. When relinquished and when applicable, base language appropriate deallocation/finalization is performed.

▼

Note – A hard pause may relinquish more resources, but may resume processing OpenMP regions more slowly. A soft pause allows OpenMP regions to restart more quickly, but may relinquish fewer resources. An OpenMP implementation will reclaim resources as needed for OpenMP regions encountered after the **omp_pause_resource** region. Since a hard pause may unmap data on the specified device, appropriate data mapping is required before using data on the specified device after the **omp_pause_region** region.

▲

The routine returns zero in case of success, and nonzero otherwise.

Tool Callbacks

If the tool is not allowed to interact with the specified device after encountering this call, then the runtime must call the tool finalizer for that device.

Restrictions

The `omp_pause_resource` routine has the following restrictions:

- The `omp_pause_resource` region may not be nested in any explicit OpenMP region.
- The routine may only be called when all explicit tasks have finalized execution. Calling the routine in any other circumstances may result in unspecified behavior.

Cross References

- `target` construct, see Section 2.12.5 on page 170
- `declare target` directive, see Section 2.12.7 on page 180
- `threadprivate` directives, see Section 2.19.2 on page 274.
- `omp_get_num_devices`, see Section 3.2.36 on page 371.
- `omp_get_get_initial_device` routine, see Section 3.2.41 on page 376.
- To pause resources on all devices at once, see Section 3.2.44 on page 380.

3.2.44 `omp_pause_resource_all`

Summary

The `omp_pause_resource_all` routine allows the runtime to relinquish resources used by OpenMP on all devices.

Format

	C / C++	
<code>int omp_pause_resource_all(omp_pause_resource_t kind);</code>		
	C / C++	
	Fortran	
<code>integer function omp_pause_resource_all(kind)</code>		
<code>integer (kind=omp_pause_resource_kind) kind</code>		
	Fortran	

Binding

The binding task set for an `omp_pause_resource_all` region is the whole program.

Effect

The `omp_pause_resource_all` routine allows the runtime to relinquish resources used by OpenMP on all devices. It is equivalent to calling the `omp_pause_resource` routine once for each available device, including the host device.

The argument `kind` passed to this routine can be one of the valid OpenMP pause kind as defined in Section 3.2.43 on page 378, or any implementation specific pause kind.

Tool Callbacks

If the tool is not allowed to interact with a given device after encountering this call, then the runtime must call the tool finalizer for that device.

Restrictions

The `omp_pause_resource_all` routine has the following restrictions:

- The `omp_pause_resource_all` region may not be nested in any explicit OpenMP region.
- The routine may only be called when all explicit tasks have finalized execution. Calling the routine in any other circumstances may result in unspecified behavior.

Cross References

- `target` construct, see Section 2.12.5 on page 170
- `declare target` directive, see Section 2.12.7 on page 180
- `omp_get_num_devices`, see Section 3.2.36 on page 371.
- `omp_get_get_initial_device` routine, see Section 3.2.41 on page 376.
- To pause resources on a specific device only, see Section 3.2.43 on page 378.

3.3 Lock Routines

The OpenMP runtime library includes a set of general-purpose lock routines that can be used for synchronization. These general-purpose lock routines operate on OpenMP locks that are represented by OpenMP lock variables. OpenMP lock variables must be accessed only through the routines described in this section; programs that otherwise access OpenMP lock variables are non-conforming.

An OpenMP lock can be in one of the following states: *uninitialized*; *unlocked*; or *locked*. If a lock is in the *unlocked* state, a task can *set* the lock, which changes its state to *locked*. The task that sets the lock is then said to *own* the lock. A task that owns a lock can *unset* that lock, returning it to the *unlocked* state. A program in which a task unsets a lock that is owned by another task is non-conforming.

Two types of locks are supported: *simple locks* and *nestable locks*. A *nestable lock* can be set multiple times by the same task before being unset; a *simple lock* cannot be set if it is already owned by the task trying to set it. *Simple lock* variables are associated with *simple locks* and can only be passed to *simple lock* routines. *Nestable lock* variables are associated with *nestable locks* and can only be passed to *nestable lock* routines.

Each type of lock can also have a *synchronization hint* that contains information about the intended usage of the lock by the application code. The effect of the hint is implementation defined. An OpenMP implementation can use this hint to select a usage-specific lock, but hints do not change the mutual exclusion semantics of locks. A conforming implementation can safely ignore the hint.

Constraints on the state and ownership of the lock accessed by each of the lock routines are described with the routine. If these constraints are not met, the behavior of the routine is unspecified.

The OpenMP lock routines access a lock variable such that they always read and update the most current value of the lock variable. It is not necessary for an OpenMP program to include explicit **flush** directives to ensure that the lock variable's value is consistent among different tasks.

Binding

The binding thread set for all lock routine regions is all threads in the contention group. As a consequence, for each OpenMP lock, the lock routine effects relate to all tasks that call the routines, without regard to which teams the threads in the contention group that are executing the tasks belong.

Simple Lock Routines

C / C++

The type **omp_lock_t** represents a simple lock. For the following routines, a simple lock variable must be of **omp_lock_t** type. All simple lock routines require an argument that is a pointer to a variable of type **omp_lock_t**.

C / C++

Fortran

For the following routines, a simple lock variable must be an integer variable of **kind=omp_lock_kind**.

Fortran

The simple lock routines are as follows:

- The **omp_init_lock** routine initializes a simple lock;
- The **omp_init_lock_with_hint** routine initializes a simple lock and attaches a hint to it;
- The **omp_destroy_lock** routine uninitializes a simple lock;
- The **omp_set_lock** routine waits until a simple lock is available and then sets it;
- The **omp_unset_lock** routine unsets a simple lock; and
- The **omp_test_lock** routine tests a simple lock and sets it if it is available.

Nestable Lock Routines

C / C++

The type **omp_nest_lock_t** represents a nestable lock. For the following routines, a nestable lock variable must be of **omp_nest_lock_t** type. All nestable lock routines require an argument that is a pointer to a variable of type **omp_nest_lock_t**.

C / C++

Fortran

For the following routines, a nestable lock variable must be an integer variable of **kind=omp_nest_lock_kind**.

Fortran

The nestable lock routines are as follows:

- The **omp_init_nest_lock** routine initializes a nestable lock;
- The **omp_init_nest_lock_with_hint** routine initializes a nestable lock and attaches a hint to it;
- The **omp_destroy_nest_lock** routine uninitializes a nestable lock;
- The **omp_set_nest_lock** routine waits until a nestable lock is available and then sets it;
- The **omp_unset_nest_lock** routine unsets a nestable lock; and
- The **omp_test_nest_lock** routine tests a nestable lock and sets it if it is available.

Restrictions

OpenMP lock routines have the following restriction:

- The use of the same OpenMP lock in different contention groups results in unspecified behavior.

3.3.1 `omp_init_lock` and `omp_init_nest_lock`

Summary

These routines initialize an OpenMP lock without a hint.

Format

C / C++

```
void omp_init_lock(omp_lock_t *lock);  
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

C / C++
Fortran

```
subroutine omp_init_lock(svar)  
integer (kind=omp_lock_kind) svar  
  
subroutine omp_init_nest_lock(nvar)  
integer (kind=omp_nest_lock_kind) nvar
```

Constraints on Arguments

A program that accesses a lock that is not in the uninitialized state through either routine is non-conforming.

Effect

The effect of these routines is to initialize the lock to the unlocked state; that is, no task owns the lock. In addition, the nesting count for a nestable lock is set to zero.

Execution Model Events

The *lock-init* event occurs in a thread that executes an `omp_init_lock` region after initialization of the lock, but before it finishes the region. The *nest-lock-init* event occurs in a thread that executes an `omp_init_nest_lock` region after initialization of the lock, but before it finishes the region.

Tool Callbacks

A thread dispatches a registered `ompt_callback_lock_init` callback with `omp_sync_hint_none` as the *hint* argument and `ompt_mutex_lock` as the *kind* argument for each occurrence of a *lock-init* event in that thread. Similarly, a thread dispatches a registered `ompt_callback_lock_init` callback with `omp_sync_hint_none` as the *hint* argument and `ompt_mutex_nest_lock` as the *kind* argument for each occurrence of a *nest-lock-init* event in that thread. These callbacks have the type signature `ompt_callback_mutex_acquire_t` and occur in the task that encounters the routine.

Cross References

- `ompt_callback_mutex_acquire_t`, see Section 4.5.2.14 on page 476.

3.3.2 `omp_init_lock_with_hint` and `omp_init_nest_lock_with_hint`

Summary

These routines initialize an OpenMP lock with a hint. The effect of the hint is implementation-defined. The OpenMP implementation can ignore the hint without changing program semantics.

Format

C / C++

```
void omp_init_lock_with_hint(  
    omp_lock_t *lock,  
    omp_sync_hint_t hint  
);  
void omp_init_nest_lock_with_hint(  
    omp_nest_lock_t *lock,  
    omp_sync_hint_t hint  
);
```

C / C++

Fortran

```
1  subroutine omp_init_lock_with_hint(svar, hint)
2  integer (kind=omp_lock_kind) svar
3  integer (kind=omp_sync_hint_kind) hint
4
5  subroutine omp_init_nest_lock_with_hint(nvar, hint)
6  integer (kind=omp_nest_lock_kind) nvar
7  integer (kind=omp_sync_hint_kind) hint
```

Fortran

Constraints on Arguments

A program that accesses a lock that is not in the uninitialized state through either routine is non-conforming.

The second argument passed to these routines (*hint*) is a hint as described in Section 2.17.12 on page 260.

Effect

The effect of these routines is to initialize the lock to the unlocked state and, optionally, to choose a specific lock implementation based on the hint. After initialization no task owns the lock. In addition, the nesting count for a nestable lock is set to zero.

Execution Model Events

The *lock-init* event occurs in a thread that executes an **omp_init_lock_with_hint** region after initialization of the lock, but before it finishes the region. The *nest-lock-init_with_hint* event occurs in a thread that executes an **omp_init_nest_lock** region after initialization of the lock, but before it finishes the region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_lock_init** callback with the same value for its *hint* argument as the *hint* argument of the call to **omp_init_lock_with_hint** and **ompt_mutex_lock** as the *kind* argument for each occurrence of a *lock-init* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_lock_init** callback with the same value for its *hint* argument as the *hint* argument of the call to **omp_init_nest_lock_with_hint** and **ompt_mutex_nest_lock** as the *kind* argument for each occurrence of a *nest-lock-init* event in that thread. These callbacks have the type signature **ompt_callback_mutex_acquire_t** and occur in the task that encounters the routine.

Cross References

- Synchronization Hints, see Section 2.17.12 on page 260.
- `ompt_callback_mutex_acquire_t`, see Section 4.5.2.14 on page 476.

3.3.3 `omp_destroy_lock` and `omp_destroy_nest_lock`

Summary

These routines ensure that the OpenMP lock is uninitialized.

Format

C / C++

```
void omp_destroy_lock(omp_lock_t *lock);  
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
```

C / C++

Fortran

```
subroutine omp_destroy_lock(svar)  
integer (kind=omp_lock_kind) svar  
  
subroutine omp_destroy_nest_lock(nvar)  
integer (kind=omp_nest_lock_kind) nvar
```

Fortran

Constraints on Arguments

A program that accesses a lock that is not in the unlocked state through either routine is non-conforming.

Effect

The effect of these routines is to change the state of the lock to uninitialized.

Execution Model Events

The *lock-destroy* event occurs in a thread that executes an `omp_destroy_lock` region before it finishes the region. The *nest-lock-destroy_with_hint* event occurs in a thread that executes an `omp_destroy_nest_lock` region before it finishes the region.

Tool Callbacks

A thread dispatches a registered `ompt_callback_lock_destroy` callback with `ompt_mutex_lock` as the *kind* argument for each occurrence of a *lock-destroy* event in that thread. Similarly, a thread dispatches a registered `ompt_callback_lock_destroy` callback with `ompt_mutex_nest_lock` as the *kind* argument for each occurrence of a *nest-lock-destroy* event in that thread. These callbacks have the type signature `ompt_callback_mutex_acquire_t` and occur in the task that encounters the routine.

Cross References

- `ompt_callback_mutex_t`, see Section 4.5.2.15 on page 477.

3.3.4 `omp_set_lock` and `omp_set_nest_lock`

Summary

These routines provide a means of setting an OpenMP lock. The calling task region behaves as if it was suspended until the lock can be set by this task.

Format

C / C++

```
void omp_set_lock(omp_lock_t *lock);  
void omp_set_nest_lock(omp_nest_lock_t *lock);
```

C / C++

Fortran

```
subroutine omp_set_lock(svar)  
integer (kind=omp_lock_kind) svar  
  
subroutine omp_set_nest_lock(nvar)  
integer (kind=omp_nest_lock_kind) nvar
```

Fortran

Constraints on Arguments

A program that accesses a lock that is in the uninitialized state through either routine is non-conforming. A simple lock accessed by `omp_set_lock` that is in the locked state must not be owned by the task that contains the call or deadlock will result.

Effect

Each of these routines has an effect equivalent to suspension of the task that is executing the routine until the specified lock is available.

Note – The semantics of these routines is specified *as if* they serialize execution of the region guarded by the lock. However, implementations may implement them in other ways provided that the isolation properties are respected so that the actual execution delivers a result that could arise from some serialization.

A simple lock is available if it is unlocked. Ownership of the lock is granted to the task that executes the routine.

A nestable lock is available if it is unlocked or if it is already owned by the task that executes the routine. The task that executes the routine is granted, or retains, ownership of the lock, and the nesting count for the lock is incremented.

Execution Model Events

The *lock-acquire* event occurs in a thread that executes an **omp_set_lock** region before the associated lock is requested. The *nest-lock-acquire* event occurs in a thread that executes an **omp_set_nest_lock** region before the associated lock is requested.

The *lock-acquired* event occurs in a thread that executes an **omp_set_lock** region after it acquires the associated lock but before it finishes the region. The *nest-lock-acquired* event occurs in a thread that executes an **omp_set_nest_lock** region if the thread did not already own the lock, after it acquires the associated lock but before it finishes the region.

The *nest-lock-owned* event occurs in a thread when it already owns the lock and executes an **omp_set_nest_lock** region. The event occurs after the nesting count is incremented but before the thread finishes the region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_mutex_acquire** callback for each occurrence of a *lock-acquire* or *nest-lock-acquire* event in that thread. This callback has the type signature **ompt_callback_mutex_acquire_t**.

A thread dispatches a registered **ompt_callback_mutex_acquired** callback for each occurrence of a *lock-acquired* or *nest-lock-acquired* event in that thread. This callback has the type signature **ompt_callback_mutex_t**.

A thread dispatches a registered **ompt_callback_nest_lock** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *nest-lock-owned* event in that thread. This callback has the type signature **ompt_callback_nest_lock_t**.

The above callbacks occur in the task that encounters the lock function. The *kind* argument of these callbacks is `ompt_mutex_lock` when the events arise from an `omp_set_lock` region while it is `ompt_mutex_nest_lock` when the events arise from an `omp_set_nest_lock` region.

Cross References









- `ompt_callback_mutex_acquire_t`, see Section 4.5.2.14 on page 476.
- `ompt_callback_mutex_t`, see Section 4.5.2.15 on page 477.
- `ompt_callback_nest_lock_t`, see Section 4.5.2.16 on page 479.

3.3.5 `omp_unset_lock` and `omp_unset_nest_lock`

Summary

These routines provide the means of unsetting an OpenMP lock.

Format

		C / C++	
	<pre>void omp_unset_lock(omp_lock_t *lock); void omp_unset_nest_lock(omp_nest_lock_t *lock);</pre>		
		C / C++	
		Fortran	
	<pre>subroutine omp_unset_lock(svar) integer (kind=omp_lock_kind) svar subroutine omp_unset_nest_lock(nvar) integer (kind=omp_nest_lock_kind) nvar</pre>		
		Fortran	

Constraints on Arguments

A program that accesses a lock that is not in the locked state or that is not owned by the task that contains the call through either routine is non-conforming.

Effect

For a simple lock, the **omp_unset_lock** routine causes the lock to become unlocked.

For a nestable lock, the **omp_unset_nest_lock** routine decrements the nesting count, and causes the lock to become unlocked if the resulting nesting count is zero.

For either routine, if the lock becomes unlocked, and if one or more task regions were effectively suspended because the lock was unavailable, the effect is that one task is chosen and given ownership of the lock.

Execution Model Events

The *lock-release* event occurs in a thread that executes an **omp_unset_lock** region after it releases the associated lock but before it finishes the region. The *nest-lock-release* event occurs in a thread that executes an **omp_unset_nest_lock** region after it releases the associated lock but before it finishes the region.

The *nest-lock-held* event occurs in a thread that executes an **omp_unset_nest_lock** region before it finishes the region when the thread still owns the lock after the nesting count is decremented.

Tool Callbacks

A thread dispatches a registered **ompt_callback_mutex_released** callback with **ompt_mutex_lock** as the *kind* argument for each occurrence of a *lock-release* event in that thread. Similarly, a thread dispatches a registered **ompt_callback_mutex_released** callback with **ompt_mutex_nest_lock** as the *kind* argument for each occurrence of a *nest-lock-release* event in that thread. These callbacks have the type signature **ompt_callback_mutex_t** and occur in the task that encounters the routine.

A thread dispatches a registered **ompt_callback_nest_lock** callback with **ompt_scope_end** as its *endpoint* argument for each occurrence of a *nest-lock-held* event in that thread. This callback has the type signature **ompt_callback_nest_lock_t**.

Cross References

- **ompt_callback_mutex_t**, see Section [4.5.2.15](#) on page [477](#).
- **ompt_callback_nest_lock_t**, see Section [4.5.2.16](#) on page [479](#).

3.3.6 `omp_test_lock` and `omp_test_nest_lock`

Summary

These routines attempt to set an OpenMP lock but do not suspend execution of the task that executes the routine.

Format

	C / C++
<pre>int omp_test_lock(omp_lock_t *lock); int omp_test_nest_lock(omp_nest_lock_t *lock);</pre>	
	C / C++
	Fortran
<pre>logical function omp_test_lock(svar) integer (kind=omp_lock_kind) svar integer function omp_test_nest_lock(nvar) integer (kind=omp_nest_lock_kind) nvar</pre>	
	Fortran

Constraints on Arguments

A program that accesses a lock that is in the uninitialized state through either routine is non-conforming. The behavior is unspecified if a simple lock accessed by `omp_test_lock` is in the locked state and is owned by the task that contains the call.

Effect

These routines attempt to set a lock in the same manner as `omp_set_lock` and `omp_set_nest_lock`, except that they do not suspend execution of the task that executes the routine.

For a simple lock, the `omp_test_lock` routine returns *true* if the lock is successfully set; otherwise, it returns *false*.

For a nestable lock, the `omp_test_nest_lock` routine returns the new nesting count if the lock is successfully set; otherwise, it returns zero.

Execution Model Events

The *lock-test* event occurs in a thread that executes an **omp_test_lock** region before the associated lock is tested. The *nest-lock-test* event occurs in a thread that executes an **omp_test_nest_lock** region before the associated lock is tested.

The *lock-test-acquired* event occurs in a thread that executes an **omp_test_lock** region before it finishes the region if the associated lock was acquired. The *nest-lock-test-acquired* event occurs in a thread that executes an **omp_test_nest_lock** region before it finishes the region if the associated lock was acquired and the thread did not already own the lock.

The *nest-lock-owned* event occurs in a thread that executes an **omp_test_nest_lock** region before it finishes the region after the nesting count is incremented if the thread already owned the lock.

Tool Callbacks

A thread dispatches a registered **ompt_callback_mutex_acquire** callback for each occurrence of a *lock-test* or *nest-lock-test* event in that thread. This callback has the type signature **ompt_callback_mutex_acquire_t**.

A thread dispatches a registered **ompt_callback_mutex_acquired** callback for each occurrence of a *lock-test-acquired* or *nest-lock-test-acquired* event in that thread. This callback has the type signature **ompt_callback_mutex_t**.

A thread dispatches a registered **ompt_callback_nest_lock** callback with **ompt_scope_begin** as its *endpoint* argument for each occurrence of a *nest-lock-owned* event in that thread. This callback has the type signature **ompt_callback_nest_lock_t**.

The above callbacks occur in the task that encounters the lock function. The *kind* argument of these callbacks is **ompt_mutex_test_lock** when the events arise from an **omp_test_lock** region while it is **ompt_mutex_test_nest_lock** when the events arise from an **omp_test_nest_lock** region.

Cross References

- **ompt_callback_mutex_acquire_t**, see Section [4.5.2.14](#) on page [476](#).
- **ompt_callback_mutex_t**, see Section [4.5.2.15](#) on page [477](#).
- **ompt_callback_nest_lock_t**, see Section [4.5.2.16](#) on page [479](#).

1 **3.4 Timing Routines**

2 This section describes routines that support a portable wall clock timer.

3 **3.4.1 omp_get_wtime**

4 **Summary**

5 The **omp_get_wtime** routine returns elapsed wall clock time in seconds.

6 **Format**

7

▼

`double omp_get_wtime(void);`

▲

C / C++

8

▼

`double precision function omp_get_wtime()`

▲

Fortran

9 **Binding**

10 The binding thread set for an **omp_get_wtime** region is the encountering thread. The routine's
11 return value is not guaranteed to be consistent across any set of threads.

12 **Effect**

13 The **omp_get_wtime** routine returns a value equal to the elapsed wall clock time in seconds
14 since some *time-in-the-past*. The actual *time-in-the-past* is arbitrary, but it is guaranteed not to
15 change during the execution of the application program. The time returned is a *per-thread time*, so
16 it is not required to be globally consistent across all threads that participate in an application.

17

▼

18 **Note** – The routine is anticipated to be used to measure elapsed times as shown in the following
19 example:

		C / C++	
1	double start;		
2	double end;		
3	start = omp_get_wtime();		
4	... work to be timed ...		
5	end = omp_get_wtime();		
6	printf("Work took %f seconds\n", end - start);		
		C / C++	
		Fortran	
7	DOUBLE PRECISION START, END		
8	START = omp_get_wtime()		
9	... work to be timed ...		
10	END = omp_get_wtime()		
11	PRINT *, "Work took", END - START, "seconds"		
		Fortran	
12			

3.4.2 omp_get_wtick

Summary

The `omp_get_wtick` routine returns the precision of the timer used by `omp_get_wtime`.

Format

		C / C++	
17	double omp_get_wtick(void);		
		C / C++	
		Fortran	
18	double precision function omp_get_wtick()		
		Fortran	

Binding

The binding thread set for an `omp_get_wtick` region is the encountering thread. The routine's return value is not guaranteed to be consistent across any set of threads.

Effect

The `omp_get_wtick` routine returns a value equal to the number of seconds between successive clock ticks of the timer used by `omp_get_wtime`.

3.5 Event Routine

This section describes a routine that supports OpenMP event objects.

Binding

The binding thread set for all event routine regions is the encountering thread.

3.5.1 `omp_fulfill_event`

Summary

This routine fulfills and destroys an OpenMP event.

Format

C / C++

`void omp_fulfill_event(omp_event_handle_t event);`

C / C++

Fortran

`subroutine omp_fulfill_event(event)
integer (kind=omp_event_handle_kind) event`

Fortran

Constraints on Arguments

A program that calls this routine on an event that was already fulfilled is non-conforming. A program that calls this routine with an event handle that was not created by the `detach` clause is non-conforming.

Effect

The effect of this routine is to fulfill the event associated with the event handle argument. The effect of fulfilling the event will depend on how the event was created. The event is destroyed and cannot be accessed after calling this routine, and the event handle becomes unassociated with any event.

Execution Model Events

The *task-fulfill* event occurs in a thread that executes an **omp_fulfill_event** region before the event is fulfilled if the OpenMP event object was created by a **detach** clause on a task.

Tool Callbacks

A thread dispatches a registered **ompt_callback_task_schedule** callback with **NULL** as its *next_task_data* argument while the argument *prior_task_data* binds to the detached task for each occurrence of a *task-fulfill* event. If the *task-fulfill* event occurs before the detached task finished the execution of the associated *structured-block*, the callback has **ompt_task_early_fulfill** as its *prior_task_status* argument; otherwise the callback has **ompt_task_late_fulfill** as its *prior_task_status* argument. This callback has type signature **ompt_callback_task_schedule_t**.

Cross References

- **detach** clause, see Section 2.10.1 on page 135.
- **ompt_callback_task_schedule_t**, see Section 4.5.2.10 on page 470.

▼ C / C++ ▼

3.6 Device Memory Routines

This section describes routines that support allocation of memory and management of pointers in the data environments of target devices.

3.6.1 omp_target_alloc

Summary

The **omp_target_alloc** routine allocates memory in a device data environment.

Format

```
void* omp_target_alloc(size_t size, int device_num);
```

Effect

The **omp_target_alloc** routine returns the device address of a storage location of *size* bytes. The storage location is dynamically allocated in the device data environment of the device specified by *device_num*, which must be greater than or equal to zero and less than the result of **omp_get_num_devices()** or the result of a call to **omp_get_initial_device()**. When called from within a **target** region the effect of this routine is unspecified.

The **omp_target_alloc** routine returns **NULL** if it cannot dynamically allocate the memory in the device data environment.

The device address returned by **omp_target_alloc** can be used in an **is_device_ptr** clause, Section 2.12.5 on page 170.

Unless **unified_address** clause appears on a **requires** directive in the compilation unit, pointer arithmetic is not supported on the device address returned by **omp_target_alloc**.

Freeing the storage returned by **omp_target_alloc** with any routine other than **omp_target_free** results in unspecified behavior.

Execution Model Events

The *target-data-allocation* event occurs when a thread allocates data on a target device.

Tool Callbacks

A thread invokes a registered **ompt_callback_target_data_op** callback for each occurrence of a *target-data-allocation* event in that thread. The callback occurs in the context of the target task and has type signature **ompt_callback_target_data_op_t**.

Cross References

- **target** construct, see Section 2.12.5 on page 170
- **omp_get_num_devices** routine, see Section 3.2.36 on page 371
- **omp_get_initial_device** routine, see Section 3.2.41 on page 376
- **omp_target_free** routine, see Section 3.6.2 on page 399
- **ompt_callback_target_data_op_t**, see Section 4.5.2.25 on page 488.

3.6.2 `omp_target_free`

Summary

The `omp_target_free` routine frees the device memory allocated by the `omp_target_alloc` routine.

Format

```
void omp_target_free(void *device_ptr, int device_num);
```

Constraints on Arguments

A program that calls `omp_target_free` with a non-null pointer that does not have a value returned from `omp_target_alloc` is non-conforming. The `device_num` must be greater than or equal to zero and less than the result of `omp_get_num_devices()` or the result of a call to `omp_get_initial_device()`.

Effect

The `omp_target_free` routine frees the memory in the device data environment associated with `device_ptr`. If `device_ptr` is `NULL`, the operation is ignored.

Synchronization must be inserted to ensure that all accesses to `device_ptr` are completed before the call to `omp_target_free`.

When called from within a **target** region the effect of this routine is unspecified.

Execution Model Events

The *target-data-free* event occurs when a thread frees data on a target device.

Tool Callbacks

A thread invokes a registered `ompt_callback_target_data_op` callback for each occurrence of a *target-data-free* event in that thread. The callback occurs in the context of the target task and has type signature `ompt_callback_target_data_op_t`.

Cross References

- **target** construct, see Section [2.12.5](#) on page [170](#)
- `omp_get_num_devices` routine, see Section [3.2.36](#) on page [371](#)
- `omp_get_initial_device` routine, see Section [3.2.41](#) on page [376](#)
- `omp_target_alloc` routine, see Section [3.6.1](#) on page [397](#)
- `ompt_callback_target_data_op_t`, see Section [4.5.2.25](#) on page [488](#).

3.6.3 `omp_target_is_present`

Summary

The `omp_target_is_present` routine tests whether a host pointer has corresponding storage on a given device.

Format

```
int omp_target_is_present(const void *ptr, int device_num);
```

Constraints on Arguments

The value of *ptr* must be a valid host pointer or `NULL`. The *device_num* must be greater than or equal to zero and less than the result of `omp_get_num_devices()` or the result of a call to `omp_get_initial_device()`.

Effect

This routine returns non-zero if the specified pointer would be found present on device *device_num* by a `map` clause; otherwise, it returns zero.

When called from within a `target` region the effect of this routine is unspecified.

Cross References

- `target` construct, see Section [2.12.5](#) on page [170](#).
- `map` clause, see Section [2.19.7.1](#) on page [315](#).
- `omp_get_num_devices` routine, see Section [3.2.36](#) on page [371](#)
- `omp_get_initial_device` routine, see Section [3.2.41](#) on page [376](#)

3.6.4 `omp_target_memcpy`

Summary

The `omp_target_memcpy` routine copies memory between any combination of host and device pointers.

Format

```
int omp_target_memcpy(
    void *dst,
    const void *src,
    size_t length,
    size_t dst_offset,
    size_t src_offset,
    int dst_device_num,
    int src_device_num
);
```

Constraints on Arguments

Each device must be compatible with the device pointer specified on the same side of the copy. The *dst_device_num* and *src_device_num* must be greater than or equal to zero and less than the result of `omp_get_num_devices()` or equal to the result of a call to `omp_get_initial_device()`.

Effect

length bytes of memory at offset *src_offset* from *src* in the device data environment of device *src_device_num* are copied to *dst* starting at offset *dst_offset* in the device data environment of device *dst_device_num*. The return value is zero on success and non-zero on failure. The host device and host device data environment can be referenced with the device number returned by `omp_get_initial_device`. This routine contains a task scheduling point.

When called from within a **target** region the effect of this routine is unspecified.

Execution Model Events

The *target-data-op* event occurs when a thread transfers data on a target device.

Tool Callbacks

A thread invokes a registered `ompt_callback_target_data_op` callback for each occurrence of a *target-data-op* event in that thread. The callback occurs in the context of the target task and has type signature `ompt_callback_target_data_op_t`.

Cross References

- **target** construct, see Section 2.12.5 on page 170.
- `omp_get_initial_device` routine, see Section 3.2.41 on page 376
- `omp_target_alloc` routine, see Section 3.6.1 on page 397.
- `ompt_callback_target_data_op_t`, see Section 4.5.2.25 on page 488.

3.6.5 `omp_target_memcpy_rect`

Summary

The `omp_target_memcpy_rect` routine copies a rectangular subvolume from a multi-dimensional array to another multi-dimensional array. The copies can use any combination of host and device pointers.

Format

```
int omp_target_memcpy_rect (
    void *dst,
    const void *src,
    size_t element_size,
    int num_dims,
    const size_t *volume,
    const size_t *dst_offsets,
    const size_t *src_offsets,
    const size_t *dst_dimensions,
    const size_t *src_dimensions,
    int dst_device_num,
    int src_device_num
);
```

Constraints on Arguments

The length of the offset and dimension arrays must be at least the value of `num_dims`. The `dst_device_num` and `src_device_num` must be greater than or equal to zero and less than the result of `omp_get_num_devices()` or equal to the result of a call to `omp_get_initial_device()`.

The value of `num_dims` must be between 1 and the implementation-defined limit, which must be at least three.

Effect

This routine copies a rectangular subvolume of `src`, in the device data environment of device `src_device_num`, to `dst`, in the device data environment of device `dst_device_num`. The volume is specified in terms of the size of an element, number of dimensions, and constant arrays of length `num_dims`. The maximum number of dimensions supported is at least three, support for higher dimensionality is implementation defined. The volume array specifies the length, in number of elements, to copy in each dimension from `src` to `dst`. The `dst_offsets` (`src_offsets`) parameter specifies number of elements from the origin of `dst` (`src`) in elements. The `dst_dimensions` (`src_dimensions`) parameter specifies the length of each dimension of `dst` (`src`).

The routine returns zero if successful. If both *dst* and *src* are **NULL** pointers, the routine returns the number of dimensions supported by the implementation for the specified device numbers. The host device and host device data environment can be referenced with the device number returned by **omp_get_initial_device**. Otherwise, it returns a non-zero value. The routine contains a task scheduling point.

When called from within a **target** region the effect of this routine is unspecified.

Execution Model Events

The *target-data-op* event occurs when a thread transfers data on a target device.

Tool Callbacks

A thread invokes a registered **ompt_callback_target_data_op** callback for each occurrence of a *target-data-op* event in that thread. The callback occurs in the context of the target task and has type signature **ompt_callback_target_data_op_t**.

Cross References

- **target** construct, see Section 2.12.5 on page 170.
- **omp_get_initial_device** routine, see Section 3.2.41 on page 376
- **omp_target_alloc** routine, see Section 3.6.1 on page 397.
- **ompt_callback_target_data_op_t**, see Section 4.5.2.25 on page 488.

3.6.6 omp_target_associate_ptr

Summary

The **omp_target_associate_ptr** routine maps a device pointer, which may be returned from **omp_target_alloc** or implementation-defined runtime routines, to a host pointer.

Format

```

int omp_target_associate_ptr(
    const void *host_ptr,
    const void *device_ptr,
    size_t size,
    size_t device_offset,
    int device_num
);

```

Constraints on Arguments

The value of *device_ptr* value must be a valid pointer to device memory for the device denoted by the value of *device_num*. The *device_num* argument must be greater than or equal to zero and less than the result of `omp_get_num_devices()` or equal to the result of a call to `omp_get_initial_device()`.

Effect

The `omp_target_associate_ptr` routine associates a device pointer in the device data environment of device *device_num* with a host pointer such that when the host pointer appears in a subsequent `map` clause, the associated device pointer is used as the target for data motion associated with that host pointer. The *device_offset* parameter specifies the offset into *device_ptr* that is used as the base address for the device side of the mapping. The reference count of the resulting mapping will be infinite. After being successfully associated, the buffer to which the device pointer points is invalidated and accessing data directly through the device pointer results in unspecified behavior. The pointer can be retrieved for other uses by disassociating it. When called from within a **target** region the effect of this routine is unspecified.

The routine returns zero if successful. Otherwise it returns a non-zero value.

Only one device buffer can be associated with a given host pointer value and device number pair. Attempting to associate a second buffer will return non-zero. Associating the same pair of pointers on the same device with the same offset has no effect and returns zero. Associating pointers that share underlying storage will result in unspecified behavior. The `omp_target_is_present` function can be used to test whether a given host pointer has a corresponding variable in the device data environment.

Execution Model Events

The *target-data-associate* event occurs when a thread associates data on a target device.

Tool Callbacks

A thread invokes a registered `ompt_callback_target_data_op` callback for each occurrence of a *target-data-associate* event in that thread. The callback occurs in the context of the target task and has type signature `ompt_callback_target_data_op_t`.

Cross References

- **target** construct, see Section 2.12.5 on page 170.
- **map** clause, see Section 2.19.7.1 on page 315.
- **omp_target_alloc** routine, see Section 3.6.1 on page 397.
- **omp_target_disassociate_ptr** routine, see Section 3.6.6 on page 403
- **ompt_callback_target_data_op_t**, see Section 4.5.2.25 on page 488.

3.6.7 omp_target_disassociate_ptr

Summary

The **omp_target_disassociate_ptr** removes the associated pointer for a given device from a host pointer.

Format

```
int omp_target_disassociate_ptr(const void *ptr, int device_num);
```

Constraints on Arguments

The *device_num* must be greater than or equal to zero and less than the result of **omp_get_num_devices()** or equal to the result of a call to **omp_get_initial_device()**.

Effect

The **omp_target_disassociate_ptr** removes the associated device data on device *device_num* from the presence table for host pointer *ptr*. A call to this routine on a pointer that is not **NULL** and does not have associated data on the given device results in unspecified behavior. The reference count of the mapping is reduced to zero, regardless of its current value.

When called from within a **target** region the effect of this routine is unspecified.

The routine returns zero if successful. Otherwise it returns a non-zero value.

After a call to **omp_target_disassociate_ptr**, the contents of the device buffer are invalidated.

Execution Model Events

The *target-data-disassociate* event occurs when a thread disassociates data on a target device.

Tool Callbacks

A thread invokes a registered `ompt_callback_target_data_op` callback for each occurrence of a *target-data-disassociate* event in that thread. The callback occurs in the context of the target task and has type signature `ompt_callback_target_data_op_t`.

Cross References

- `target` construct, see Section 2.12.5 on page 170
- `omp_target_associate_ptr` routine, see Section 3.6.6 on page 403
- `ompt_callback_target_data_op_t`, see Section 4.5.2.25 on page 488.

▲ C / C++ ▲

3.7 Memory Management Routines

This section describes routines that support memory management on the current device.

Instances of memory management types must be accessed only through the routines described in this section; programs that otherwise access instances of these types are non-conforming.

3.7.1 Memory Management Types

The following type definitions are used by the memory management routines:

▲ C / C++ ▲

```
typedef enum omp_alloctrail_key_t {
    omp_atk_sync_hint = 1,
    omp_atk_alignment = 2,
    omp_atk_access = 3,
    omp_atk_pool_size = 4,
    omp_atk_fallback = 5,
    omp_atk_fb_data = 6,
    omp_atk_pinned = 7,
    omp_atk_partition = 8
} omp_alloctrail_key_t;

typedef enum omp_alloctrail_value_t {
```

```

1  omp_atv_false = 0,
2  omp_atv_true = 1,
3  omp_atv_default = 2,
4  omp_atv_contended = 3,
5  omp_atv_uncontended = 4,
6  omp_atv_sequential = 5,
7  omp_atv_private = 6,
8  omp_atv_all = 7,
9  omp_atv_thread = 8,
10 omp_atv_pteam = 9,
11 omp_atv_cgroup = 10,
12 omp_atv_default_mem_fb = 11,
13 omp_atv_null_fb = 12,
14 omp_atv_abort_fb = 13,
15 omp_atv_allocator_fb = 14,
16 omp_atv_environment = 15,
17 omp_atv_nearest = 16,
18 omp_atv_blocked = 17,
19 omp_atv_interleaved = 18
20 } omp_alloctrail_value_t;
21
22 typedef struct omp_alloctrail_t {
23     omp_alloctrail_key_t key;
24     omp_uintptr_t value;
25 } omp_alloctrail_t;

```

C / C++

Fortran

```

26
27 integer(kind=omp_alloctrail_key_kind), &
28     parameter :: omp_atk_sync_hint = 1
29 integer(kind=omp_alloctrail_key_kind), &
30     parameter :: omp_atk_alignment = 2
31 integer(kind=omp_alloctrail_key_kind), &
32     parameter :: omp_atk_access = 3
33 integer(kind=omp_alloctrail_key_kind), &
34     parameter :: omp_atk_pool_size = 4
35 integer(kind=omp_alloctrail_key_kind), &
36     parameter :: omp_atk_fallback = 5
37 integer(kind=omp_alloctrail_key_kind), &
38     parameter :: omp_atk_fb_data = 6
39 integer(kind=omp_alloctrail_key_kind), &
40     parameter :: omp_atk_pinned = 7
41 integer(kind=omp_alloctrail_key_kind), &

```

```

1      parameter :: omp_atk_partition = 8
2
3      integer(kind=omp_alloctratit_val_kind), &
4      parameter :: omp_atv_false = 0
5      integer(kind=omp_alloctratit_val_kind), &
6      parameter :: omp_atv_true = 1
7      integer(kind=omp_alloctratit_val_kind), &
8      parameter :: omp_atv_default = 2
9      integer(kind=omp_alloctratit_val_kind), &
10     parameter :: omp_atv_contended = 3
11     integer(kind=omp_alloctratit_val_kind), &
12     parameter :: omp_atv_uncontended = 4
13     integer(kind=omp_alloctratit_val_kind), &
14     parameter :: omp_atv_sequential = 5
15     integer(kind=omp_alloctratit_val_kind), &
16     parameter :: omp_atv_private = 6
17     integer(kind=omp_alloctratit_val_kind), &
18     parameter :: omp_atv_all = 7
19     integer(kind=omp_alloctratit_val_kind), &
20     parameter :: omp_atv_thread = 8
21     integer(kind=omp_alloctratit_val_kind), &
22     parameter :: omp_atv_pteam = 9
23     integer(kind=omp_alloctratit_val_kind), &
24     parameter :: omp_atv_cgroup = 10
25     integer(kind=omp_alloctratit_val_kind), &
26     parameter :: omp_atv_default_mem_fb = 11
27     integer(kind=omp_alloctratit_val_kind), &
28     parameter :: omp_atv_null_fb = 12
29     integer(kind=omp_alloctratit_val_kind), &
30     parameter :: omp_atv_abort_fb = 13
31     integer(kind=omp_alloctratit_val_kind), &
32     parameter :: omp_atv_allocator_fb = 14
33     integer(kind=omp_alloctratit_val_kind), &
34     parameter :: omp_atv_environment = 15
35     integer(kind=omp_alloctratit_val_kind), &
36     parameter :: omp_atv_nearest = 16
37     integer(kind=omp_alloctratit_val_kind), &
38     parameter :: omp_atv_blocked = 17
39     integer(kind=omp_alloctratit_val_kind), &
40     parameter :: omp_atv_interleaved = 18
41
42     type omp_alloctratit
43     integer(kind=omp_alloctratit_key_kind) key

```

```

1      integer(kind=omp_alloctrail_val_kind) value
2      end type omp_alloctrail
3
4      integer(kind=omp_allocator_handle_kind), &
5      parameter :: omp_null_allocator = 0

```

Fortran

3.7.2 omp_init_allocator

Summary

The **omp_init_allocator** routine initializes an allocator and associates it with a memory space.

Format

C / C++

```

omp_allocator_handle_t omp_init_allocator (
    omp_memspace_handle_t memspace,
    int ntraits,
    const omp_alloctrail_t traits[]
);

```

C / C++

Fortran

```

integer(kind=omp_allocator_handle_kind) &
function omp_init_allocator ( memspace, ntraits, traits )
integer(kind=omp_memspace_handle_kind),intent(in) :: memspace
integer,intent(in) :: ntraits
type(omp_alloctrail),intent(in) :: traits(*)

```

Fortran

Constraints on Arguments

The *memspace* argument must be one of the predefined memory spaces defined in Table 2.8.

If the *ntraits* argument is greater than zero then the *traits* argument must specify at least that many traits. If it specifies fewer than *ntraits* traits the behavior is unspecified.

Unless a **requires** directive with the **dynamic_allocators** clause is present in the same compilation unit, using this routine in a **target** region results in unspecified behavior.

Binding

The binding thread set for an **omp_init_allocator** region is all threads on a device. The effect of executing this routine is not related to any specific region that corresponds to any construct or API routine.

Effect

The **omp_init_allocator** routine creates a new allocator that is associated with the *memspace* memory space and returns a handle to it. All allocations through the created allocator will behave according to the allocator traits specified in the *traits* argument. The number of traits in the *traits* argument is specified by the *ntraits* argument. Specifying the same allocator trait more than once results in unspecified behavior. The routine returns a handle for the created allocator. If the special **omp_atv_default** value is used for a given trait, then its value will be the default value specified in Table 2.9 for that given trait.

If *memspace* is **omp_default_mem_space** and the **traits** argument is an empty set this routine will always return a handle to an allocator. Otherwise if an allocator based on the requirements cannot be created then the special **omp_null_allocator** handle is returned.

The use of an allocator returned by this routine on a device other than the one on which it was created results in unspecified behavior.

Cross References

- Memory Spaces, see Section 2.11.1 on page 152.
- Memory Allocators, see Section 2.11.2 on page 152.

3.7.3 **omp_destroy_allocator**

Summary

The **omp_destroy_allocator** routine releases all resources used by the allocator handle.

Format

		C / C++	
	void	omp_destroy_allocator	(omp_allocator_handle_t <i>allocator</i>);
		C / C++	
		Fortran	
	subroutine	omp_destroy_allocator	(<i>allocator</i>)
	integer(kind=omp_allocator_handle_kind),intent(in)	::	<i>allocator</i>
		Fortran	

1 **Constraints on Arguments**

2 The *allocator* argument must not represent a predefined memory allocator.

3 Unless a **requires** directive with the **dynamic_allocators** clause is present in the same

4 compilation unit, using this routine in a **target** region results in unspecified behavior.

5 **Binding**

6 The binding thread set for an **omp_destroy_allocator** region is all threads on a device. The

7 effect of executing this routine is not related to any specific region that corresponds to any construct

8 or API routine.

9 **Effect**

10 The **omp_destroy_allocator** routine releases all resources used to implement the *allocator*

11 handle. Accessing any memory allocated by the *allocator* after this call results in unspecified

12 behavior.

13 If *allocator* is **omp_null_allocator** then this routine will have no effect.

14 **Cross References**

- 15
 - Memory Allocators, see Section [2.11.2](#) on page [152](#).

16 **3.7.4 omp_set_default_allocator**

17 **Summary**

18 The **omp_set_default_allocator** routine sets the default memory allocator to be used by

19 allocation calls, **allocate** directives and **allocate** clauses that do not specify an allocator.

20 **Format**

21

C / C++

void omp_set_default_allocator (omp_allocator_handle_t allocator);

22

C / C++

Fortran

23

subroutine omp_set_default_allocator (allocator)

integer(kind=omp_allocator_handle_kind),intent(in) :: allocator

24

Fortran

Constraints on Arguments

The *allocator* argument must be a valid memory allocator handle.

Binding

The binding task set for an **omp_set_default_allocator** region is the binding implicit task.

Effect

The effect of this routine is to set the value of the *def-allocator-var* ICV of the binding implicit task to the value specified in the *allocator* argument.

Cross References

- *def-allocator-var* ICV, see Section 2.5 on page 63.
- Memory Allocators, see Section 2.11.2 on page 152.
- **omp_alloc** routine, see Section 3.7.6 on page 413.

3.7.5 omp_get_default_allocator

Summary

The **omp_get_default_allocator** routine returns a handle to the memory allocator to be used by allocation calls, **allocate** directives and **allocate** clauses that do not specify an allocator.

Format

	C / C++	
omp_allocator_handle_t	omp_get_default_allocator	(void);
	C / C++	
	Fortran	
integer(kind=omp_allocator_handle_kind) &	function omp_get_default_allocator	()
	Fortran	

Binding

The binding task set for an **omp_get_default_allocator** region is the binding implicit task.

Effect

The effect of this routine is to return the value of the *def-allocator-var* ICV of the binding implicit task.

Cross References

- *def-allocator-var* ICV, see Section 2.5 on page 63.
- Memory Allocators, see Section 2.11.2 on page 152.
- `omp_alloc` routine, see Section 3.7.6 on page 413.

C / C++

3.7.6 `omp_alloc`

Summary

The `omp_alloc` routine requests a memory allocation from a memory allocator.

Format

C
`void *omp_alloc (size_t size, omp_allocator_handle_t allocator);`

C++
`void *omp_alloc(
 size_t size,
 omp_allocator_handle_t allocator=omp_null_allocator
);`

C++

Constraints on Arguments

Unless `dynamic_allocators` appears on a `requires` directive in the same compilation unit, `omp_alloc` invocations that appear in `target` regions must not pass `omp_null_allocator` as the *allocator* argument, which must be a constant expression that evaluates to one of the predefined memory allocator values.

Effect

The **omp_alloc** routine requests a memory allocation of *size* bytes from the specified memory allocator. If the *allocator* argument is **omp_null_allocator** the memory allocator used by the routine will be the one specified by the *def-allocator-var* ICV of the binding implicit task. Upon success it returns a pointer to the allocated memory. Otherwise, the behavior specified by the **fallback** trait will be followed.

Allocated memory will be byte aligned to at least the alignment required by **malloc**.

Cross References

- Memory allocators, see Section 2.11.2 on page 152.

3.7.7 omp_free**Summary**

The **omp_free** routine deallocates previously allocated memory.

Format

C

```
void omp_free (void *ptr, omp_allocator_handle_t allocator);
```

C++

```
void omp_free(
    void *ptr,
    omp_allocator_handle_t allocator=omp_null_allocator
);
```

Effect

The **omp_free** routine deallocates the memory to which *ptr* points. The *ptr* argument must point to memory previously allocated with a memory allocator. If the *allocator* argument is specified it must be the memory allocator to which the allocation request was made. If the *allocator* argument is **omp_null_allocator** the implementation will determine that value automatically. Using **omp_free** on memory that was already deallocated or that was allocated by an allocator that has already been destroyed with **omp_destroy_allocator** results in unspecified behavior.

Cross References

- Memory allocators, see Section 2.11.2 on page 152.

C / C++

3.8 Tool Control Routine

Summary

The `omp_control_tool` routine enables a program to pass commands to an active tool.

Format

C / C++

```
int omp_control_tool(int command, int modifier, void *arg);
```

C / C++

Fortran

```
integer function omp_control_tool(command, modifier)  
integer (kind=omp_control_tool_kind) command  
integer modifier
```

Fortran

Description

An OpenMP program may use `omp_control_tool` to pass commands to a tool. An application can use `omp_control_tool` to request that a tool starts or restarts data collection when a code region of interest is encountered, that a tool pauses data collection when leaving the region of interest, that a tool flushes any data that it has collected so far, or that a tool ends data collection. Additionally, `omp_control_tool` can be used to pass tool-specific commands to a particular tool.

The following types correspond to return values from `omp_control_tool`:

C / C++

```
typedef enum omp_control_tool_result_t {  
    omp_control_tool_notool = -2,  
    omp_control_tool_nocallback = -1,  
    omp_control_tool_success = 0,  
    omp_control_tool_ignored = 1  
} omp_control_tool_result_t;
```

C / C++

Fortran

```
1 integer (kind=omp_control_tool_result_kind), &  
2     parameter :: omp_control_tool_notool = -2  
3 integer (kind=omp_control_tool_result_kind), &  
4     parameter :: omp_control_tool_nocallback = -1  
5 integer (kind=omp_control_tool_result_kind), &  
6     parameter :: omp_control_tool_success = 0  
7 integer (kind=omp_control_tool_result_kind), &  
8     parameter :: omp_control_tool_ignored = 1
```

Fortran

If the OMPT interface state is inactive, the OpenMP implementation returns **omp_control_tool_notool**. If the OMPT interface state is active, but no callback is registered for the *tool-control* event, the OpenMP implementation returns **omp_control_tool_nocallback**. An OpenMP implementation may return other implementation-defined negative values strictly smaller than -64; an application may assume that any negative return value indicates that a tool has not received the command. A return value of **omp_control_tool_success** indicates that the tool has performed the specified command. A return value of **omp_control_tool_ignored** indicates that the tool has ignored the specified command. A tool may return other positive values strictly greater than 64 that are tool-defined.

Constraints on Arguments

The following enumeration type defines four standard commands. Table 3.1 describes the actions that these commands request from a tool.

C / C++

```
21 typedef enum omp_control_tool_t {  
22     omp_control_tool_start = 1,  
23     omp_control_tool_pause = 2,  
24     omp_control_tool_flush = 3,  
25     omp_control_tool_end = 4  
26 } omp_control_tool_t;
```

C / C++

Fortran

```
27 integer (kind=omp_control_tool_kind), &  
28     parameter :: omp_control_tool_start = 1  
29 integer (kind=omp_control_tool_kind), &  
30     parameter :: omp_control_tool_pause = 2  
31 integer (kind=omp_control_tool_kind), &  
32     parameter :: omp_control_tool_flush = 3  
33 integer (kind=omp_control_tool_kind), &  
34     parameter :: omp_control_tool_end = 4
```

Fortran

Tool-specific values for *command* must be greater or equal to 64. Tools must ignore *command* values that they are not explicitly designed to handle. Other values accepted by a tool for *command*, and any values for *modifier* and *arg* are tool-defined.

TABLE 3.1: Standard Tool Control Commands

Command	Action
<code>omp_control_tool_start</code>	Start or restart monitoring if it is off. If monitoring is already on, this command is idempotent. If monitoring has already been turned off permanently, this command will have no effect.
<code>omp_control_tool_pause</code>	Temporarily turn monitoring off. If monitoring is already off, it is idempotent.
<code>omp_control_tool_flush</code>	Flush any data buffered by a tool. This command may be applied whether monitoring is on or off.
<code>omp_control_tool_end</code>	Turn monitoring off permanently; the tool finalizes itself and flushes all output.

Execution Model Events

The *tool-control* event occurs in the thread that encounters a call to `omp_control_tool` at a point inside its corresponding OpenMP region.

Tool Callbacks

A thread dispatches a registered `ompt_callback_control_tool` callback for each occurrence of a *tool-control* event. The callback executes in the context of the call that occurs in the user program and has type signature `ompt_callback_control_tool_t`. The callback may return any non-negative value, which will be returned to the application by the OpenMP implementation as the return value of the `omp_control_tool` call that triggered the callback.

Arguments passed to the callback are those passed by the user to `omp_control_tool`. If the call is made in Fortran, the tool will be passed `NULL` as the third argument to the callback. If any of the four standard commands is presented to a tool, the tool will ignore the *modifier* and *arg* argument values.

Cross References

- OMPT Interface, see Chapter 4 on page 419
- `ompt_callback_control_tool_t`, see Section 4.5.2.29 on page 495

This page intentionally left blank

CHAPTER 4

OMPT Interface

This chapter describes OMPT, which is an interface for *first-party* tools. *First-party* tools are linked or loaded directly into the OpenMP program. OMPT defines mechanisms to initialize a tool, to examine OpenMP state associated with an OpenMP thread, to interpret the call stack of an OpenMP thread, to receive notification about OpenMP *events*, to trace activity on OpenMP target devices, to assess implementation-dependent details of an OpenMP implementation (such as supported states and mutual exclusion implementations), and to control a tool from an OpenMP application.

4.1 OMPT Interfaces Definitions

C / C++

A compliant implementation must supply a set of definitions for the OMPT runtime entry points, OMPT callback signatures, and the special data types of their parameters and return values. These definitions, which are listed throughout this chapter, and their associated declarations shall be provided in a header file named **omp-tools.h**. In addition, the set of definitions may specify other implementation-specific values.

The **ompt_start_tool** function is an external function with C linkage.

C / C++

1 4.2 Activating a First-Party Tool

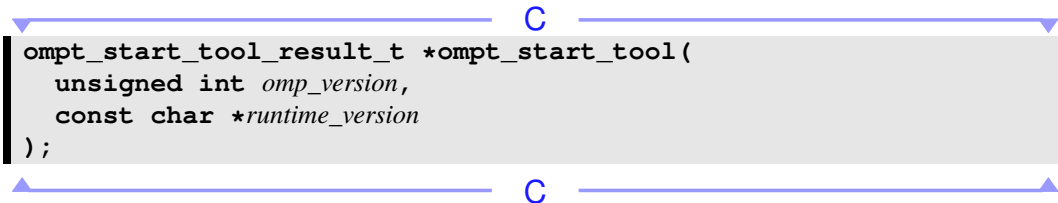
2 To activate a tool, an OpenMP implementation first determines whether the tool should be
3 initialized. If so, the OpenMP implementation invokes the initializer of the tool, which enables the
4 tool to prepare to monitor execution on the host. The tool may then also arrange to monitor
5 computation that executes on target devices. This section explains how the tool and an OpenMP
6 implementation interact to accomplish these tasks.

7 4.2.1 `ompt_start_tool`

8 Summary

9 In order to use the OMPT interface provided by an OpenMP implementation, a tool must implement
10 the `ompt_start_tool` function, through which the OpenMP implementation initializes the tool.

11 Format

```
12 
13 ompt_start_tool_result_t *ompt_start_tool(
14     unsigned int omp_version,
15     const char *runtime_version
16 );
```

16 Description

17 For a tool to use the OMPT interface that an OpenMP implementation provides, the tool must define
18 a globally-visible implementation of the function `ompt_start_tool`. The tool indicates that it
19 will use the OMPT interface that an OpenMP implementation provides by returning a non-null
20 pointer to an `ompt_start_tool_result_t` structure from the `ompt_start_tool`
21 implementation that it provides. The `ompt_start_tool_result_t` structure contains
22 pointers to tool initialization and finalization callbacks as well as a tool data word that an OpenMP
23 implementation must pass by reference to these callbacks. A tool may return `NULL` from
24 `ompt_start_tool` to indicate that it will not use the OMPT interface in a particular execution.

25 A tool may use the `omp_version` argument to determine if it is compatible with the OMPT interface
26 that the OpenMP implementation provides.

Description of Arguments

The argument *omp_version* is the value of the `_OPENMP` version macro associated with the OpenMP API implementation. This value identifies the OpenMP API version that an OpenMP implementation supports, which specifies the version of the OMPT interface that it supports.

The argument *runtime_version* is a version string that unambiguously identifies the OpenMP implementation.

Constraints on Arguments

The argument *runtime_version* must be an immutable string that is defined for the lifetime of a program execution.

Effect

If a tool returns a non-null pointer to an `ompt_start_tool_result_t` structure, an OpenMP implementation will call the tool initializer specified by the *initialize* field in this structure before beginning execution of any OpenMP construct or completing execution of any environment routine invocation; the OpenMP implementation will call the tool finalizer specified by the *finalize* field in this structure when the OpenMP implementation shuts down.

Cross References

- `ompt_start_tool_result_t`, see Section 4.4.1 on page 433.

4.2.2 Determining Whether a First-Party Tool Should be Initialized

An OpenMP implementation examines the *tool-var* ICV as one of its first initialization steps. If the value of *tool-var* is *disabled*, the initialization continues without a check for the presence of a tool and the functionality of the OMPT interface will be unavailable as the program executes. In this case, the OMPT interface state remains *inactive*.

Otherwise, the OMPT interface state changes to *pending* and the OpenMP implementation activates any first-party tool that it finds. A tool can provide a definition of `ompt_start_tool` to an OpenMP implementation in three ways:

- By statically-linking its definition of `ompt_start_tool` into an OpenMP application;
- By introducing a dynamically-linked library that includes its definition of `ompt_start_tool` into the application's address space; or

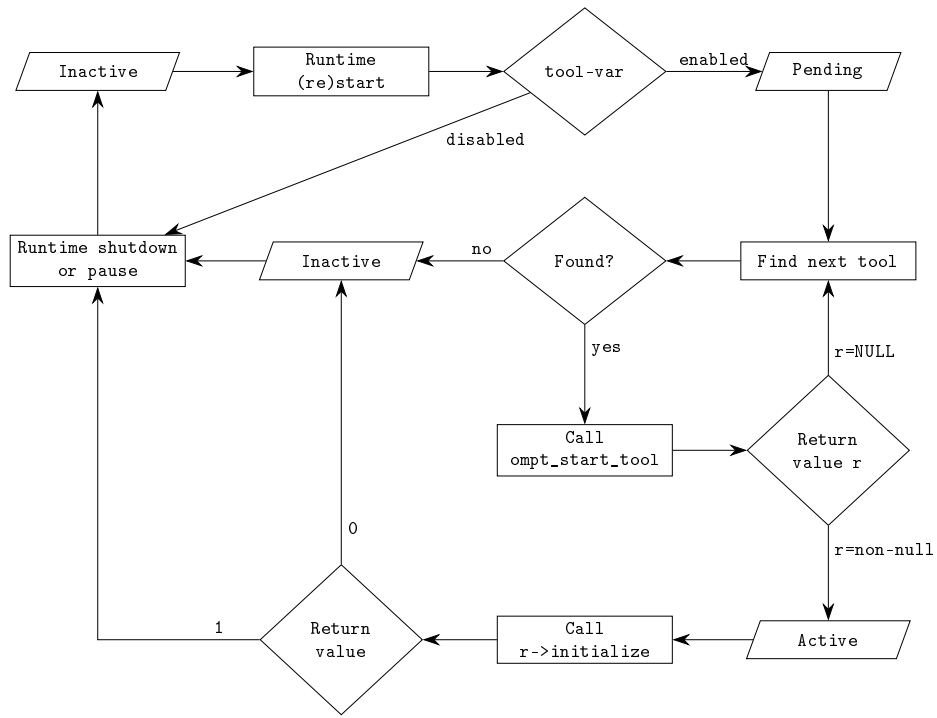


FIGURE 4.1: First-Party Tool Activation Flow Chart

- By providing, in the *tool-libraries-var* ICV, the name of a dynamically-linked library that is appropriate for the architecture and operating system used by the application and that includes a definition of `ompt_start_tool`.

If the value of *tool-var* is *enabled*, the OpenMP implementation must check if a tool has provided an implementation of `ompt_start_tool`. The OpenMP implementation first checks if a tool-provided implementation of `ompt_start_tool` is available in the address space, either statically-linked into the application or in a dynamically-linked library loaded in the address space. If multiple implementations of `ompt_start_tool` are available, the OpenMP implementation will use the first tool-provided implementation of `ompt_start_tool` that it finds.

If the implementation does not find a tool-provided implementation of `ompt_start_tool` in the address space, it consults the *tool-libraries-var* ICV, which contains a (possibly empty) list of dynamically-linked libraries. As described in detail in Section 6.19 on page 617, the libraries in *tool-libraries-var* are then searched for the first usable implementation of `ompt_start_tool` that one of the libraries in the list provides.

If the implementation finds a tool-provided definition of `ompt_start_tool`, it invokes that method; if a **NULL** pointer is returned, the OMPT interface state remains *pending* and the

implementation continues to look for implementations of `ompt_start_tool`; otherwise a non-null pointer to an `ompt_start_tool_result_t` structure is returned, the OMPT interface state changes to *active* and the OpenMP implementation makes the OMPT interface available as the program executes. In this case, as the OpenMP implementation completes its initialization, it initializes the OMPT interface.

If no tool can be found, the OMPT interface state changes to *inactive*.

Cross References

- *tool-libraries-var* ICV, see Section 2.5 on page 63.
- *tool-var* ICV, see Section 2.5 on page 63.
- `ompt_start_tool` function, see Section 4.2.1 on page 420.
- `ompt_start_tool_result_t` type, see Section 4.4.1 on page 433.

4.2.3 Initializing a First-Party Tool

To initialize the OMPT interface, the OpenMP implementation invokes the tool initializer that is specified in the `ompt_start_tool_result_t` structure that is indicated by the non-null pointer that `ompt_start_tool` returns. The initializer is invoked prior to the occurrence of any OpenMP *event*.

A tool initializer, described in Section 4.5.1.1 on page 457, uses the function specified in its *lookup* argument to look up pointers to OMPT interface runtime entry points that the OpenMP implementation provides; this process is described in Section 4.2.3.1 on page 424. Typically, a tool initializer obtains a pointer to the `ompt_set_callback` runtime entry point with type signature `ompt_set_callback_t` and then uses this runtime entry point to register tool callbacks for OpenMP events, as described in Section 4.2.4 on page 425.

A tool initializer may use the `ompt_enumerate_states` runtime entry point, which has type signature `ompt_enumerate_states_t`, to determine the thread states that an OpenMP implementation employs. Similarly, it may use the `ompt_enumerate_mutex_impls` runtime entry point, which has type signature `ompt_enumerate_mutex_impls_t`, to determine the mutual exclusion implementations that the OpenMP implementation employs.

If a tool initializer returns a non-zero value, the OMPT interface state remains *active* for the execution; otherwise, the OMPT interface state changes to *inactive*.

Cross References

- `ompt_start_tool` function, see Section 4.2.1 on page 420.
- `ompt_start_tool_result_t` type, see Section 4.4.1 on page 433.
- `ompt_initialize_t` type, see Section 4.5.1.1 on page 457.
- `ompt_callback_thread_begin_t` type, see Section 4.5.2.1 on page 459.
- `ompt_enumerate_states_t` type, see Section 4.6.1.1 on page 498.
- `ompt_enumerate_mutex_impls_t` type, see Section 4.6.1.2 on page 499.
- `ompt_set_callback_t` type, see Section 4.6.1.3 on page 500.
- `ompt_function_lookup_t` type, see Section 4.6.3 on page 531.

4.2.3.1 Binding Entry Points in the OMPT Callback Interface

Functions that an OpenMP implementation provides to support the OMPT interface are not defined as global function symbols. Instead, they are defined as runtime entry points that a tool can only identify through the *lookup* function that is provided as an argument with type signature `ompt_function_lookup_t` to the tool initializer. A tool can use this function to obtain a pointer to each of the runtime entry points that an OpenMP implementation provides to support the OMPT interface. Once a tool has obtained a *lookup* function, it may employ it at any point in the future.

For each runtime entry point in the OMPT interface for the host device, Table 4.1 provides the string name by which it is known and its associated type signature. Implementations can provide additional implementation-specific names and corresponding entry points. Any names that begin with `ompt_` are reserved names.

During initialization, a tool should look up each runtime entry point in the OMPT interface by name and bind a pointer maintained by the tool that can later be used to invoke the entry point. The entry points described in Table 4.1 enable a tool to assess the thread states and mutual exclusion implementations that an OpenMP implementation supports, to register tool callbacks, to inspect registered callbacks, to introspect OpenMP state associated with threads, and to use tracing to monitor computations that execute on target devices.

Detailed information about each runtime entry point listed in Table 4.1 is included as part of the description of its type signature.

Cross References

- `ompt_enumerate_states_t` type, see Section 4.6.1.1 on page 498.
- `ompt_enumerate_mutex_impls_t` type, see Section 4.6.1.2 on page 499.

- `ompt_set_callback_t` type, see Section 4.6.1.3 on page 500.
- `ompt_get_callback_t` type, see Section 4.6.1.4 on page 502.
- `ompt_get_thread_data_t` type, see Section 4.6.1.5 on page 503.
- `ompt_get_num_procs_t` type, see Section 4.6.1.6 on page 503.
- `ompt_get_num_places_t` type, see Section 4.6.1.7 on page 504.
- `ompt_get_place_proc_ids_t` type, see Section 4.6.1.8 on page 505.
- `ompt_get_place_num_t` type, see Section 4.6.1.9 on page 506.
- `ompt_get_partition_place_nums_t` type, see Section 4.6.1.10 on page 507.
- `ompt_get_proc_id_t` type, see Section 4.6.1.11 on page 508.
- `ompt_get_state_t` type, see Section 4.6.1.12 on page 508.
- `ompt_get_parallel_info_t` type, see Section 4.6.1.13 on page 510.
- `ompt_get_task_info_t` type, see Section 4.6.1.14 on page 512.
- `ompt_get_task_memory_t` type, see Section 4.6.1.15 on page 514.
- `ompt_get_target_info_t` type, see Section 4.6.1.16 on page 515.
- `ompt_get_num_devices_t` type, see Section 4.6.1.17 on page 516.
- `ompt_get_unique_id_t` type, see Section 4.6.1.18 on page 517.
- `ompt_finalize_tool_t` type, see Section 4.6.1.19 on page 517.
- `ompt_function_lookup_t` type, see Section 4.6.3 on page 531.

4.2.4 Monitoring Activity on the Host with OMPT

To monitor the execution of an OpenMP program on the host device, a tool initializer must register to receive notification of events that occur as an OpenMP program executes. A tool can use the `ompt_set_callback` runtime entry point to register callbacks for OpenMP events. The return codes for `ompt_set_callback` use the `ompt_set_result_t` enumeration type. If the `ompt_set_callback` runtime entry point is called outside a tool initializer, registration of supported callbacks may fail with a return value of `ompt_set_error`.

All callbacks registered with `ompt_set_callback` or returned by `ompt_get_callback` use the dummy type signature `ompt_callback_t`.

Table 4.2 shows the valid registration return codes of the `ompt_set_callback` runtime entry point with specific values of its *event* argument. For callbacks for which `ompt_set_always` is

TABLE 4.1: OMPT Callback Interface Runtime Entry Point Names and Their Type Signatures

Entry Point String Name	Type signature
<code>"ompt_enumerate_states"</code>	<code>ompt_enumerate_states_t</code>
<code>"ompt_enumerate_mutex_impls"</code>	<code>ompt_enumerate_mutex_impls_t</code>
<code>"ompt_set_callback"</code>	<code>ompt_set_callback_t</code>
<code>"ompt_get_callback"</code>	<code>ompt_get_callback_t</code>
<code>"ompt_get_thread_data"</code>	<code>ompt_get_thread_data_t</code>
<code>"ompt_get_num_places"</code>	<code>ompt_get_num_places_t</code>
<code>"ompt_get_place_proc_ids"</code>	<code>ompt_get_place_proc_ids_t</code>
<code>"ompt_get_place_num"</code>	<code>ompt_get_place_num_t</code>
<code>"ompt_get_partition_place_nums"</code>	<code>ompt_get_partition_place_nums_t</code>
<code>"ompt_get_proc_id"</code>	<code>ompt_get_proc_id_t</code>
<code>"ompt_get_state"</code>	<code>ompt_get_state_t</code>
<code>"ompt_get_parallel_info"</code>	<code>ompt_get_parallel_info_t</code>
<code>"ompt_get_task_info"</code>	<code>ompt_get_task_info_t</code>
<code>"ompt_get_task_memory"</code>	<code>ompt_get_task_memory_t</code>
<code>"ompt_get_num_devices"</code>	<code>ompt_get_num_devices_t</code>
<code>"ompt_get_num_procs"</code>	<code>ompt_get_num_procs_t</code>
<code>"ompt_get_target_info"</code>	<code>ompt_get_target_info_t</code>
<code>"ompt_get_unique_id"</code>	<code>ompt_get_unique_id_t</code>
<code>"ompt_finalize_tool"</code>	<code>ompt_finalize_tool_t</code>

the only registration return code that is allowed, an OpenMP implementation must guarantee that the callback will be invoked every time that a runtime event that is associated with it occurs. Support for such callbacks is required in a minimal implementation of the OMPT interface. For callbacks for which the `ompt_set_callback` runtime entry may return values other than `ompt_set_always`, whether an OpenMP implementation invokes a registered callback never, sometimes, or always is implementation-defined. If registration for a callback allows a return code of `ompt_set_never`, support for invoking such a callback may not be present in a minimal implementation of the OMPT interface. The return code from registering a callback indicates the implementation-defined level of support for the callback.

Two techniques reduce the size of the OMPT interface. First, in cases where events are naturally paired, for example, the beginning and end of a region, and the arguments needed by the callback at each endpoint are identical, a tool registers a single callback for the pair of events, with `ompt_scope_begin` or `ompt_scope_end` provided as an argument to identify for which endpoint the callback is invoked. Second, when a class of events is amenable to uniform treatment, OMPT provides a single callback for that class of events, for example, an `ompt_callback_sync_region_wait` callback is used for multiple kinds of synchronization regions, such as barrier, taskwait, and taskgroup regions. Some events, for example, `ompt_callback_sync_region_wait`, use both techniques.

Cross References

- `ompt_set_result_t` type, see Section 4.4.4.2 on page 438.
- `ompt_set_callback_t` type, see Section 4.6.1.3 on page 500.
- `ompt_get_callback_t` type, see Section 4.6.1.4 on page 502.

4.2.5 Tracing Activity on Target Devices with OMPT

A target device may or may not initialize a full OpenMP runtime system. Unless it does, it may not be possible to monitor activity on a device using a tool interface based on callbacks. To accommodate such cases, the OMPT interface defines a monitoring interface for tracing activity on target devices. Tracing activity on a target device involves the following steps:

- To prepare to trace activity on a target device, a tool must register for an `ompt_callback_device_initialize` callback. A tool may also register for an `ompt_callback_device_load` callback to be notified when code is loaded onto a target device or an `ompt_callback_device_unload` callback to be notified when code is unloaded from a target device. A tool may also optionally register an `ompt_callback_device_finalize` callback.

TABLE 4.2: Valid Return Codes of **ompt_set_callback** for Each Callback

Return code abbreviation	N	S/P	A
ompt_callback_thread_begin			*
ompt_callback_thread_end			*
ompt_callback_parallel_begin			*
ompt_callback_parallel_end			*
ompt_callback_task_create			*
ompt_callback_task_schedule			*
ompt_callback_implicit_task			*
ompt_callback_target			*
ompt_callback_target_data_op			*
ompt_callback_target_submit			*
ompt_callback_control_tool			*
ompt_callback_device_initialize			*
ompt_callback_device_finalize			*
ompt_callback_device_load			*
ompt_callback_device_unload			*
ompt_callback_sync_region_wait	*	*	*
ompt_callback_mutex_released	*	*	*
ompt_callback_dependences	*	*	*
ompt_callback_task_dependence	*	*	*
ompt_callback_work	*	*	*
ompt_callback_master	*	*	*
ompt_callback_target_map	*	*	*
ompt_callback_sync_region	*	*	*
ompt_callback_reduction	*	*	*
ompt_callback_lock_init	*	*	*
ompt_callback_lock_destroy	*	*	*
ompt_callback_mutex_acquire	*	*	*
ompt_callback_mutex_acquired	*	*	*
ompt_callback_nest_lock	*	*	*
ompt_callback_flush	*	*	*
ompt_callback_cancel	*	*	*
ompt_callback_dispatch	*	*	*

N = **ompt_set_never**S = **ompt_set_sometimes**P = **ompt_set_sometimes_paired**A = **ompt_set_always**

- When an OpenMP implementation initializes a target device, the OpenMP implementation dispatches the device initialization callback of the tool on the host device. If the OpenMP implementation or target device does not support tracing, the OpenMP implementation passes **NULL** to the device initializer of the tool for its *lookup* argument; otherwise, the OpenMP implementation passes a pointer to a device-specific runtime entry point with type signature **ompt_function_lookup_t** to the device initializer of the tool.
- If a non-null *lookup* pointer is provided to the device initializer of the tool, the tool may use it to determine the runtime entry points in the tracing interface that are available for the device and may bind the returned function pointers to tool variables. Table 4.3 indicates the names of runtime entry points that may be available for a device; an implementations may provide additional implementation-defined names and corresponding entry points. The driver for the device provides the runtime entry points that enable a tool to control the trace collection interface of the device. The *native* trace format that the interface uses may be device specific and the available kinds of trace records are implementation-defined. Some devices may allow a tool to collect traces of records in a standard format known as OMPT trace records. Each OMPT trace record serves as a substitute for an OMPT callback that cannot be made on the device. The fields in each trace record type are defined in the description of the callback that the record represents. If this type of record is provided then the *lookup* function returns values for the runtime entry points **ompt_set_trace_ompt** and **ompt_get_record_ompt**, which support collecting and decoding OMPT traces. If the native tracing format for a device is the OMPT format then tracing can be controlled using the runtime entry points for native or OMPT tracing.
- The tool uses the **ompt_set_trace_native** and/or the **ompt_set_trace_ompt** runtime entry point to specify what types of events or activities to monitor on the device. The return codes for **ompt_set_trace_ompt** and **ompt_set_trace_native** use the **ompt_set_result_t** enumeration type. If the **ompt_set_trace_native** /or the **ompt_set_trace_ompt** runtime entry point is called outside a device initializer, registration of supported callbacks may fail with a return code of **ompt_set_error**.
- The tool initiates tracing on the device by invoking **ompt_start_trace**. Arguments to **ompt_start_trace** include two tool callbacks through which the OpenMP implementation can manage traces associated with the device. One allocates a buffer in which the device can deposit trace events. The second callback processes a buffer of trace events from the device.
- If the device requires a trace buffer, the OpenMP implementation invokes the tool-supplied callback function on the host device to request a new buffer.
- The OpenMP implementation monitors the execution of OpenMP constructs on the device and records a trace of events or activities into a trace buffer. If possible, device trace records are marked with a *host_op_id*—an identifier that associates device activities with the target operation that the host initiated to cause these activities. To correlate activities on the host with activities on a device, a tool can register a **ompt_callback_target_submit** callback. Before the host initiates each distinct activity associated with a structured block for a **target** construct on a device, the OpenMP implementation dispatches the **ompt_callback_target_submit** callback on the host in the thread that is executing the task that encounters the **target** construct.

TABLE 4.3: OMPT Tracing Interface Runtime Entry Point Names and Their Type Signatures

Entry Point String Name	Type Signature
<code>"ompt_get_device_num_procs"</code>	<code>ompt_get_device_num_procs_t</code>
<code>"ompt_get_device_time"</code>	<code>ompt_get_device_time_t</code>
<code>"ompt_translate_time"</code>	<code>ompt_translate_time_t</code>
<code>"ompt_set_trace_ompt"</code>	<code>ompt_set_trace_ompt_t</code>
<code>"ompt_set_trace_native"</code>	<code>ompt_set_trace_native_t</code>
<code>"ompt_start_trace"</code>	<code>ompt_start_trace_t</code>
<code>"ompt_pause_trace"</code>	<code>ompt_pause_trace_t</code>
<code>"ompt_flush_trace"</code>	<code>ompt_flush_trace_t</code>
<code>"ompt_stop_trace"</code>	<code>ompt_stop_trace_t</code>
<code>"ompt_advance_buffer_cursor"</code>	<code>ompt_advance_buffer_cursor_t</code>
<code>"ompt_get_record_type"</code>	<code>ompt_get_record_type_t</code>
<code>"ompt_get_record_ompt"</code>	<code>ompt_get_record_ompt_t</code>
<code>"ompt_get_record_native"</code>	<code>ompt_get_record_native_t</code>
<code>"ompt_get_record_abstract"</code>	<code>ompt_get_record_abstract_t</code>

Examples of activities that could cause an **ompt_callback_target_submit** callback to be dispatched include an explicit data copy between a host and target device or execution of a computation. This callback provides the tool with a pair of identifiers: one that identifies the target region and a second that uniquely identifies an activity associated with that region. These identifiers help the tool correlate activities on the target device with their target region.

- When appropriate, for example, when a trace buffer fills or needs to be flushed, the OpenMP implementation invokes the tool-supplied buffer completion callback to process a non-empty sequence of records in a trace buffer that is associated with the device.
- The tool-supplied buffer completion callback may return immediately, ignoring records in the trace buffer, or it may iterate through them using the **ompt_advance_buffer_cursor** entry point to inspect each record. A tool may use the **ompt_get_record_type** runtime entry point to inspect the type of the record at the current cursor position. Three runtime entry points (**ompt_get_record_ompt**, **ompt_get_record_native**, and **ompt_get_record_abstract**) allow tools to inspect the contents of some or all records in a trace buffer. The **ompt_get_record_native** runtime entry point uses the native trace format of the device. The **ompt_get_record_abstract** runtime entry point decodes the contents of a native trace record and summarizes them as an **ompt_record_abstract_t** record. The **ompt_get_record_ompt** runtime entry point can only be used to retrieve records in OMPT format.
- Once tracing has been started on a device, a tool may pause or resume tracing on the device at any time by invoking **ompt_pause_trace** with an appropriate flag value as an argument.
- A tool may invoke the **ompt_flush_trace** runtime entry point for a device at any time between device initialization and finalization to cause the device to flush pending trace records.
- At any time, a tool may use the **ompt_start_trace** runtime entry point to start tracing or the **ompt_stop_trace** runtime entry point to stop tracing on a device. When tracing is stopped on a device, the OpenMP implementation eventually gathers all trace records already collected on the device and presents them to the tool using the buffer completion callback.
- An OpenMP implementation can be shut down while device tracing is in progress.
- When an OpenMP implementation is shut down, it finalizes each device. Device finalization occurs in three steps. First, the OpenMP implementation halts any tracing in progress for the device. Second, the OpenMP implementation flushes all trace records collected for the device and uses the buffer completion callback associated with that device to present them to the tool. Finally, the OpenMP implementation dispatches any **ompt_callback_device_finalize** callback registered for the device.

Restrictions

Tracing activity on devices has the following restriction:

- Implementation-defined names must not start with the prefix **ompt_**, which is reserved for the OpenMP specification.

Cross References

- `ompt_callback_device_initialize_t` callback type, see Section 4.5.2.19 on page 482.
- `ompt_callback_device_finalize_t` callback type, see Section 4.5.2.20 on page 484.
- `ompt_get_device_num_procs` runtime entry point, see Section 4.6.2.1 on page 518.
- `ompt_get_device_time` runtime entry point, see Section 4.6.2.2 on page 519.
- `ompt_translate_time` runtime entry point, see Section 4.6.2.3 on page 520.
- `ompt_set_trace_ompt` runtime entry point, see Section 4.6.2.4 on page 521.
- `ompt_set_trace_native` runtime entry point, see Section 4.6.2.5 on page 522.
- `ompt_start_trace` runtime entry point, see Section 4.6.2.6 on page 523.
- `ompt_pause_trace` runtime entry point, see Section 4.6.2.7 on page 524.
- `ompt_flush_trace` runtime entry point, see Section 4.6.2.8 on page 525.
- `ompt_stop_trace` runtime entry point, see Section 4.6.2.9 on page 526.
- `ompt_advance_buffer_cursor` runtime entry point, see Section 4.6.2.10 on page 527.
- `ompt_get_record_type` runtime entry point, see Section 4.6.2.11 on page 528.
- `ompt_get_record_ompt` runtime entry point, see Section 4.6.2.12 on page 529.
- `ompt_get_record_native` runtime entry point, see Section 4.6.2.13 on page 530.
- `ompt_get_record_abstract` runtime entry point, see Section 4.6.2.14 on page 531.

4.3 Finalizing a First-Party Tool

If the OMPT interface state is active, the tool finalizer, which has type signature `ompt_finalize_t` and is specified by the *finalize* field in the `ompt_start_tool_result_t` structure returned from the `ompt_start_tool` function, is called when the OpenMP implementation shuts down.

Cross References

- `ompt_finalize_t` callback type, see Section 4.5.1.2 on page 458

4.4 OMPT Data Types

The C/C++ header file (`omp-tools.h`) provides the definitions of the types that are specified throughout this subsection.

4.4.1 Tool Initialization and Finalization

Summary

A tool's implementation of `ompt_start_tool` returns a pointer to an `ompt_start_tool_result_t` structure, which contains pointers to the tool's initialization and finalization callbacks as well as an `ompt_data_t` object for use by the tool.

Format

C / C++

```
typedef struct ompt_start_tool_result_t {  
    ompt_initialize_t initialize;  
    ompt_finalize_t finalize;  
    ompt_data_t tool_data;  
} ompt_start_tool_result_t;
```

C / C++

Restrictions

The `ompt_start_tool_result_t` type has the following restriction:

- The *initialize* and *finalize* callback pointer values in an `ompt_start_tool_result_t` structure that `ompt_start_tool` returns must be non-null.

Cross References

- `ompt_start_tool` function, see Section 4.2.1 on page 420.
- `ompt_data_t` type, see Section 4.4.4.4 on page 440.
- `ompt_initialize_t` callback type, see Section 4.5.1.1 on page 457.
- `ompt_finalize_t` callback type, see Section 4.5.1.2 on page 458.

1 4.4.2 Callbacks

2 Summary

3 The `ompt_callbacks_t` enumeration type indicates the integer codes used to identify OpenMP
4 callbacks when registering or querying them.

5 Format

C / C++

```
6 typedef enum ompt_callbacks_t {  
7     ompt_callback_thread_begin           = 1,  
8     ompt_callback_thread_end            = 2,  
9     ompt_callback_parallel_begin        = 3,  
10    ompt_callback_parallel_end           = 4,  
11    ompt_callback_task_create            = 5,  
12    ompt_callback_task_schedule          = 6,  
13    ompt_callback_implicit_task          = 7,  
14    ompt_callback_target                 = 8,  
15    ompt_callback_target_data_op         = 9,  
16    ompt_callback_target_submit          = 10,  
17    ompt_callback_control_tool           = 11,  
18    ompt_callback_device_initialize       = 12,  
19    ompt_callback_device_finalize        = 13,  
20    ompt_callback_device_load            = 14,  
21    ompt_callback_device_unload          = 15,  
22    ompt_callback_sync_region_wait       = 16,  
23    ompt_callback_mutex_released         = 17,  
24    ompt_callback_dependences            = 18,  
25    ompt_callback_task_dependence        = 19,  
26    ompt_callback_work                   = 20,  
27    ompt_callback_master                  = 21,  
28    ompt_callback_target_map             = 22,  
29    ompt_callback_sync_region            = 23,  
30    ompt_callback_lock_init              = 24,  
31    ompt_callback_lock_destroy           = 25,  
32    ompt_callback_mutex_acquire          = 26,  
33    ompt_callback_mutex_acquired         = 27,  
34    ompt_callback_nest_lock              = 28,  
35    ompt_callback_flush                   = 29,  
36    ompt_callback_cancel                 = 30,  
37    ompt_callback_reduction              = 31,  
38    ompt_callback_dispatch                = 32  
39 } ompt_callbacks_t;
```

C / C++

1 4.4.3 Tracing



2 OpenMP provides type definitions that support tracing with OMPT.

3 4.4.3.1 Record Type



4 Summary

5 The `ompt_record_t` enumeration type indicates the integer codes used to identify OpenMP
6 trace record formats.

7 Format

8  

```
9 typedef enum ompt_record_t {  
10     ompt_record_ompt          = 1,  
11     ompt_record_native       = 2,  
12     ompt_record_invalid      = 3  
13 } ompt_record_t;
```



14  

13 4.4.3.2 Native Record Kind



14 Summary

15 The `ompt_record_native_t` enumeration type indicates the integer codes used to identify
16 OpenMP native trace record contents.

17 Format

18  

```
19 typedef enum ompt_record_native_t {  
20     ompt_record_native_info = 1,  
21     ompt_record_native_event = 2  
22 } ompt_record_native_t;
```

23  

1 4.4.3.3 Native Record Abstract Type

2 Summary

3 The **ompt_record_abstract_t** type provides an abstract trace record format that is used to
4 summarize native device trace records.

5 Format

C / C++

```
6 typedef struct ompt_record_abstract_t {  
7     ompt_record_native_t rclass;  
8     const char *type;  
9     ompt_device_time_t start_time;  
10    ompt_device_time_t end_time;  
11    ompt_hwid_t hwid;  
12 } ompt_record_abstract_t;
```

C / C++

13 Description

14 An **ompt_record_abstract_t** record contains information that a tool can use to process a
15 native record that it may not fully understand. The *rclass* field indicates that the record is
16 informational or that it represents an event; this information can help a tool determine how to
17 present the record. The record *type* field points to a statically-allocated, immutable character string
18 that provides a meaningful name that a tool can use to describe the event to a user. The *start_time*
19 and *end_time* fields are used to place an event in time. The times are relative to the device clock. If
20 an event does not have an associated *start_time* (*end_time*), the value of the *start_time* (*end_time*)
21 field is **ompt_time_none**. The hardware identifier field, *hwid*, indicates the location on the
22 device where the event occurred. A *hwid* may represent a hardware abstraction such as a core or a
23 hardware thread identifier. The meaning of a *hwid* value for a device is implementation defined. If
24 no hardware abstraction is associated with the record then the value of *hwid* is **ompt_hwid_none**.

25 4.4.3.4 Record Type

26 Summary

27 The **ompt_record_ompt_t** type provides an standard complete trace record format.

Format

C / C++

```
typedef struct ompt_record_ompt_t {
    ompt_callbacks_t type;
    ompt_device_time_t time;
    ompt_id_t thread_id;
    ompt_id_t target_id;
    union {
        ompt_record_thread_begin_t thread_begin;
        ompt_record_parallel_begin_t parallel_begin;
        ompt_record_parallel_end_t parallel_end;
        ompt_record_work_t work;
        ompt_record_dispatch_t dispatch;
        ompt_record_task_create_t task_create;
        ompt_record_dependences_t dependences;
        ompt_record_task_dependence_t task_dependence;
        ompt_record_task_schedule_t task_schedule;
        ompt_record_implicit_task_t implicit_task;
        ompt_record_master_t master;
        ompt_record_sync_region_t sync_region;
        ompt_record_mutex_acquire_t mutex_acquire;
        ompt_record_mutex_t mutex;
        ompt_record_nest_lock_t nest_lock;
        ompt_record_flush_t flush;
        ompt_record_cancel_t cancel;
        ompt_record_target_t target;
        ompt_record_target_data_op_t target_data_op;
        ompt_record_target_map_t target_map;
        ompt_record_target_kernel_t target_kernel;
        ompt_record_control_tool_t control_tool;
    } record;
} ompt_record_ompt_t;
```

C / C++

Description

The field *type* specifies the type of record provided by this structure. According to the type, event specific information is stored in the matching *record* entry.

Restrictions

The `ompt_record_ompt_t` type has the following restriction:

- If *type* is set to `ompt_callback_thread_end_t` then the value of *record* is undefined.

1 4.4.4 Miscellaneous Type Definitions

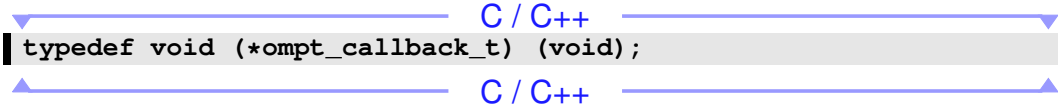
2 This section describes miscellaneous types and enumerations used by the tool interface.

3 4.4.4.1 `ompt_callback_t`

4 Summary

5 Pointers to tool callback functions with different type signatures are passed to the
6 **`ompt_set_callback`** runtime entry point and returned by the **`ompt_get_callback`**
7 runtime entry point. For convenience, these runtime entry points expect all type signatures to be
8 cast to a dummy type **`ompt_callback_t`**.

9 Format

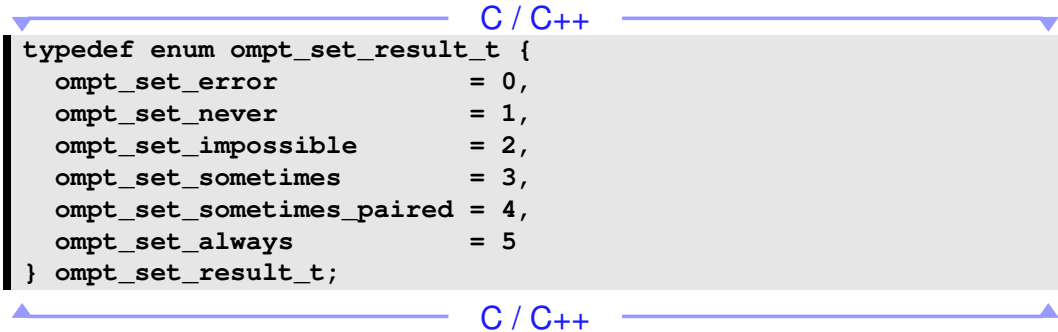
10  `typedef void (*ompt_callback_t) (void);`

11 4.4.4.2 `ompt_set_result_t`

12 Summary

13 The **`ompt_result_t`** enumeration type corresponds to values that the **`ompt_set_callback`**,
14 **`ompt_set_trace_ompt`** and **`ompt_set_trace_native`** runtime entry points return.

15 Format

16  `typedef enum ompt_set_result_t {
17 ompt_set_error = 0,
18 ompt_set_never = 1,
19 ompt_set_impossible = 2,
20 ompt_set_sometimes = 3,
21 ompt_set_sometimes_paired = 4,
22 ompt_set_always = 5
23 } ompt_set_result_t;`

1 **Description**

2 Values of `ompt_set_result_t`, may indicate several possible outcomes. The
3 **omp_set_error** value indicates that the associated call failed. Otherwise, the value indicates
4 when an event may occur and, when appropriate, *dispatching* a callback event leads to the
5 invocation of the callback. The **ompt_set_never** value indicates that the event will never occur
6 or that the callback will never be invoked at runtime. The **ompt_set_impossible** value
7 indicates that the event may occur but that tracing of it is not possible. The
8 **ompt_set_sometimes** value indicates that the event may occur and, for an
9 implementation-defined subset of associated event occurrences, will be traced or the callback will
10 be invoked at runtime. The **ompt_set_sometimes_paired** value indicates the same result as
11 **ompt_set_sometimes** and, in addition, that a callback with an *endpoint* value of
12 **ompt_scope_begin** will be invoked if and only if the same callback with an *endpoint* value of
13 **ompt_scope_end** will also be invoked sometime in the future. The **ompt_set_always** value
14 indicates that, whenever an associated event occurs, it will be traced or the callback will be invoked.

15 **Cross References**

- 16
 - Monitoring activity on the host with OMPT, see Section 4.2.4 on page 425.
 - Tracing activity on target devices with OMPT, see Section 4.2.5 on page 427.
 - **ompt_set_callback** runtime entry point, see Section 4.6.1.3 on page 500.
 - **ompt_set_trace_ompt** runtime entry point, see Section 4.6.2.4 on page 521.
 - **ompt_set_trace_native** runtime entry point, see Section 4.6.2.5 on page 522.

21 **4.4.4.3 ompt_id_t**

22 **Summary**

23 The `ompt_id_t` type is used to provide various identifiers to tools.

24 **Format**

25

```
| typedef uint64_t ompt_id_t;
```

 C / C++

Description

When tracing asynchronous activity on devices, identifiers enable tools to correlate target regions and operations that the host initiates with associated activities on a target device. In addition, OMPT provides identifiers to refer to parallel regions and tasks that execute on a device. These various identifiers are of type `ompt_id_t`.

`ompt_id_none` is defined as an instance of type `ompt_id_t` with the value 0.

Restrictions

The `ompt_id_t` type has the following restriction:

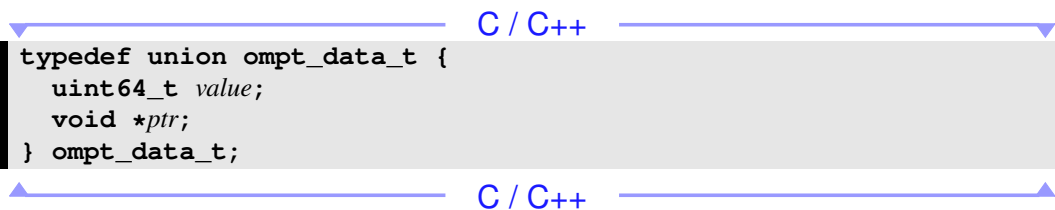
- Identifiers created on each device must be unique from the time an OpenMP implementation is initialized until it is shut down. Identifiers for each target region and target operation instance that the host device initiates must be unique over time on the host. Identifiers for parallel and task region instances that execute on a device must be unique over time within that device.

4.4.4.4 `ompt_data_t`

Summary

The `ompt_data_t` type represents data associated with threads and with parallel and task regions.

Format



```
typedef union ompt_data_t {  
    uint64_t value;  
    void *ptr;  
} ompt_data_t;
```

Description

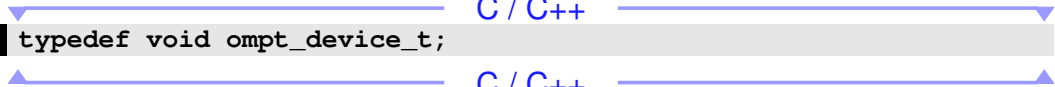
The `ompt_data_t` type represents data that is reserved for tool use and that is related to a thread or to a parallel or task region. When an OpenMP implementation creates a thread or an instance of a parallel or task region, it initializes the associated `ompt_data_t` object with the value `ompt_data_none`, which is an instance of the type with the data and pointer fields equal to 0.

1 4.4.4.5 ompt_device_t

2 Summary

3 The **ompt_device_t** opaque object type represents a device.

4 Format

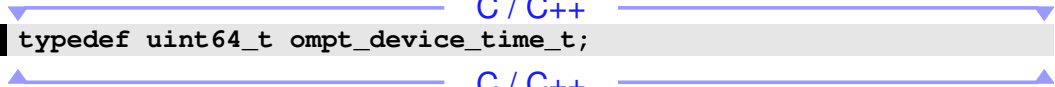
5  **typedef void ompt_device_t;**

6 4.4.4.6 ompt_device_time_t

7 Summary

8 The **ompt_device_time_t** type represents raw device time values.

9 Format

10  **typedef uint64_t ompt_device_time_t;**

11 Description

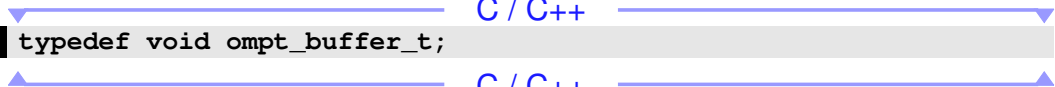
12 The **ompt_device_time_t** opaque object type represents raw device time values.
13 **ompt_time_none** refers to an unknown or unspecified time and is defined as an instance of type
14 **ompt_device_time_t** with the value 0.

15 4.4.4.7 ompt_buffer_t

16 Summary

17 The **ompt_buffer_t** opaque object type is a handle for a target buffer.

18 Format

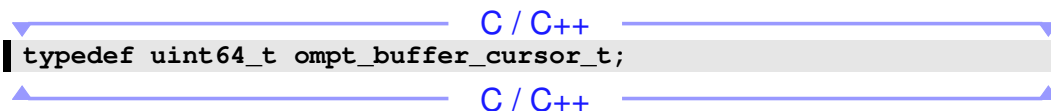
19  **typedef void ompt_buffer_t;**

1 4.4.4.8 ompt_buffer_cursor_t

2 Summary

3 The **ompt_buffer_cursor_t** opaque type is a handle for a position in a target buffer.

4 Format

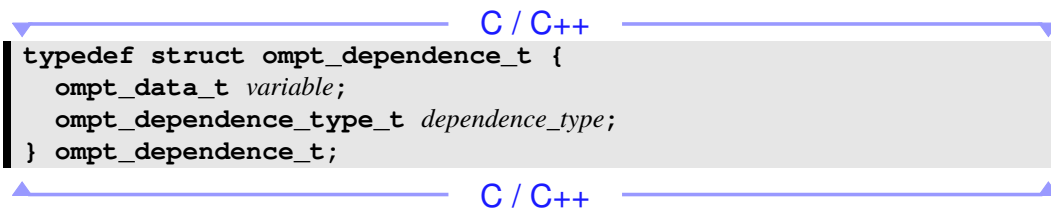
5  `typedef uint64_t ompt_buffer_cursor_t;`

6 4.4.4.9 ompt_dependence_t

7 Summary

8 The **ompt_dependence_t** type represents a task dependence.

9 Format

10  `typedef struct ompt_dependence_t {
11 ompt_data_t variable;
12 ompt_dependence_type_t dependence_type;
13 } ompt_dependence_t;`

14 Description

15 The **ompt_dependence_t** type is a structure that holds information about a depend clause. For
16 task dependences, the *variable* field points to the storage location of the dependence. For *doacross*
17 dependences, the *variable* field contains the value of a vector element that describes the
18 dependence. The *dependence_type* field indicates the type of the dependence.

19 Cross References



- 20 • **ompt_dependence_type_t** type, see Section [4.4.4.23](#) on page [450](#).

1 4.4.4.10 ompt_thread_t



2 Summary

3 The **ompt_thread_t** enumeration type defines the valid thread type values.

4 Format

5  

```
6 typedef enum ompt_thread_t {  
7     ompt_thread_initial          = 1,  
8     ompt_thread_worker          = 2,  
9     ompt_thread_other           = 3,  
10    ompt_thread_unknown         = 4  
11 } ompt_thread_t;
```

12  

11 Description



12 Any *initial thread* has thread type **ompt_thread_initial**. All *OpenMP threads* that are not
13 initial threads have thread type **ompt_thread_worker**. A thread that an OpenMP
14 implementation uses but that does not execute user code has thread type **ompt_thread_other**.
15 Any thread that is created outside an OpenMP implementation and that is not an *initial thread* has
16 thread type **ompt_thread_unknown**.

17 4.4.4.11 ompt_scope_endpoint_t



18 Summary

19 The **ompt_scope_endpoint_t** enumeration type defines valid scope endpoint values.

20 Format

21  

```
22 typedef enum ompt_scope_endpoint_t {  
23     ompt_scope_begin            = 1,  
24     ompt_scope_end             = 2  
25 } ompt_scope_endpoint_t;
```

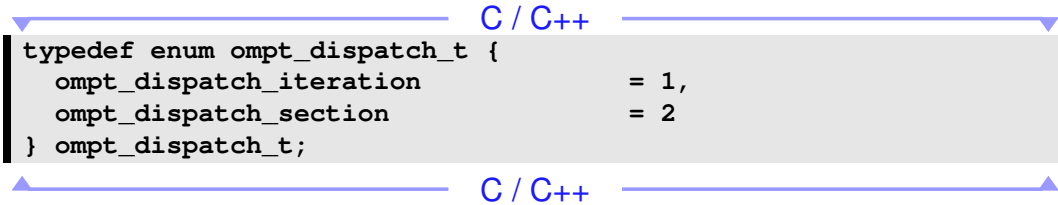
26  

1 4.4.4.12 ompt_dispatch_t

2 Summary

3 The **ompt_dispatch_t** enumeration type defines the valid dispatch kind values.

4 Format

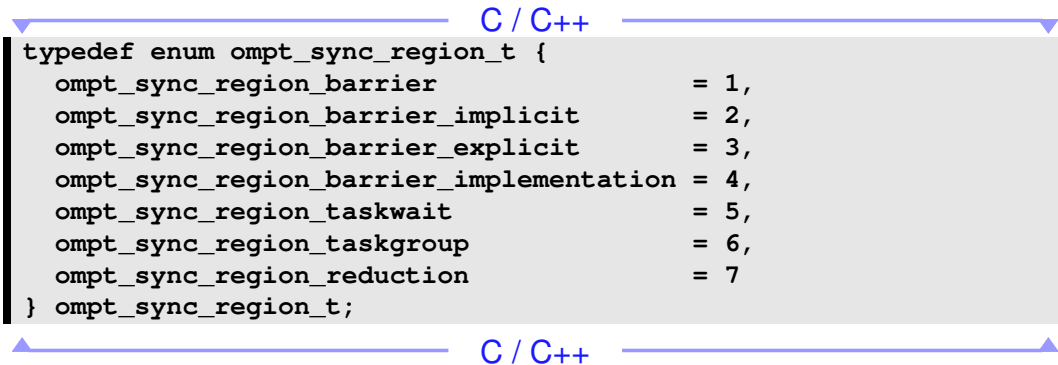
5  C / C++
6 `typedef enum ompt_dispatch_t {`
7 `ompt_dispatch_iteration = 1,`
8 `ompt_dispatch_section = 2`
`} ompt_dispatch_t;`

9 4.4.4.13 ompt_sync_region_t

10 Summary

11 The **ompt_sync_region_t** enumeration type defines the valid synchronization region kind
12 values.

13 Format

14  C / C++
15 `typedef enum ompt_sync_region_t {`
16 `ompt_sync_region_barrier = 1,`
17 `ompt_sync_region_barrier_implicit = 2,`
18 `ompt_sync_region_barrier_explicit = 3,`
19 `ompt_sync_region_barrier_implementation = 4,`
20 `ompt_sync_region_taskwait = 5,`
21 `ompt_sync_region_taskgroup = 6,`
22 `ompt_sync_region_reduction = 7`
`} ompt_sync_region_t;`

23 4.4.4.14 ompt_target_data_op_t

24 Summary

25 The **ompt_target_data_op_t** enumeration type defines the valid target data operation values.

Format

C / C++

```
typedef enum ompt_target_data_op_t {  
    ompt_target_data_alloc          = 1,  
    ompt_target_data_transfer_to_device = 2,  
    ompt_target_data_transfer_from_device = 3,  
    ompt_target_data_delete         = 4,  
    ompt_target_data_associate      = 5,  
    ompt_target_data_disassociate   = 6  
} ompt_target_data_op_t;
```

C / C++

4.4.4.15 ompt_work_t

Summary

The **ompt_work_t** enumeration type defines the valid work type values.

Format

C / C++

```
typedef enum ompt_work_t {  
    ompt_work_loop          = 1,  
    ompt_work_sections      = 2,  
    ompt_work_single_executor = 3,  
    ompt_work_single_other  = 4,  
    ompt_work_workshare     = 5,  
    ompt_work_distribute    = 6,  
    ompt_work_taskloop      = 7  
} ompt_work_t;
```

C / C++

4.4.4.16 ompt_mutex_t

Summary

The **ompt_mutex_t** enumeration type defines the valid mutex kind values.

Format

C / C++

```
typedef enum ompt_mutex_t {  
    ompt_mutex_lock           = 1,  
    ompt_mutex_test_lock     = 2,  
    ompt_mutex_nest_lock     = 3,  
    ompt_mutex_test_nest_lock = 4,  
    ompt_mutex_critical      = 5,  
    ompt_mutex_atomic        = 6,  
    ompt_mutex_ordered       = 7  
} ompt_mutex_t;
```

C / C++

4.4.4.17 ompt_native_mon_flag_t

Summary

The `ompt_native_mon_flag_t` enumeration type defines the valid native monitoring flag values.

Format

C / C++

```
typedef enum ompt_native_mon_flag_t {  
    ompt_native_data_motion_explicit = 0x01,  
    ompt_native_data_motion_implicit = 0x02,  
    ompt_native_kernel_invocation    = 0x04,  
    ompt_native_kernel_execution     = 0x08,  
    ompt_native_driver                = 0x10,  
    ompt_native_runtime               = 0x20,  
    ompt_native_overhead              = 0x40,  
    ompt_native_idleness              = 0x80  
} ompt_native_mon_flag_t;
```

C / C++

4.4.4.18 ompt_task_flag_t

Summary

The `ompt_task_flag_t` enumeration type defines valid task types.

Format

C / C++

```
typedef enum ompt_task_flag_t {  
    ompt_task_initial          = 0x00000001,  
    ompt_task_implicit        = 0x00000002,  
    ompt_task_explicit        = 0x00000004,  
    ompt_task_target          = 0x00000008,  
    ompt_task_undelayed       = 0x08000000,  
    ompt_task_untied          = 0x10000000,  
    ompt_task_final           = 0x20000000,  
    ompt_task_mergeable       = 0x40000000,  
    ompt_task_merged          = 0x80000000  
} ompt_task_flag_t;
```

C / C++

Description

The **ompt_task_flag_t** enumeration type defines valid task type values. The least significant byte provides information about the general classification of the task. The other bits represent properties of the task.

4.4.4.19 ompt_task_status_t

Summary

The **ompt_task_status_t** enumeration type indicates the reason that a task was switched when it reached a task scheduling point.

Format

C / C++

```
typedef enum ompt_task_status_t {  
    ompt_task_complete        = 1,  
    ompt_task_yield           = 2,  
    ompt_task_cancel          = 3,  
    ompt_task_detach          = 4,  
    ompt_task_early_fulfill    = 5,  
    ompt_task_late_fulfill     = 6,  
    ompt_task_switch           = 7  
} ompt_task_status_t;
```

C / C++

Description

The value **ompt_task_complete** of the **ompt_task_status_t** type indicates that the task that encountered the task scheduling point completed execution of the associated *structured-block* and an associated *allow-completion-event* was fulfilled. The value **ompt_task_yield** indicates that the task encountered a **taskyield** construct. The value **ompt_task_cancel** indicates that the task was canceled when it encountered an active cancellation point. The value **ompt_task_detach** indicates that a task with **detach** clause completed execution of the associated *structured-block* and is waiting for an *allow-completion-event* to be fulfilled. The value **ompt_task_early_fulfill** indicates that the *allow-completion-event* of the task is fulfilled before the task completed execution of the associated structured-block. The value **ompt_task_late_fulfill** indicates that the *allow-completion-event* of the task is fulfilled after the task completed execution of the associated structured-block. The value **ompt_task_switch** is used for all other cases that a task was switched.

4.4.4.20 ompt_target_t

Summary

The **ompt_target_t** enumeration type defines the valid target type values.

Format

```
typedef enum ompt_target_t {  
    ompt_target                = 1,  
    ompt_target_enter_data     = 2,  
    ompt_target_exit_data      = 3,  
    ompt_target_update         = 4  
} ompt_target_t;
```

4.4.4.21 ompt_parallel_flag_t

Summary

The **ompt_parallel_flag_t** enumeration type defines valid invoker values.

Format

C / C++

```
typedef enum ompt_parallel_flag_t {  
    ompt_parallel_invoker_program = 0x00000001,  
    ompt_parallel_invoker_runtime = 0x00000002,  
    ompt_parallel_league          = 0x40000000,  
    ompt_parallel_team            = 0x80000000  
} ompt_parallel_flag_t;
```

C / C++

Description

The **ompt_parallel_flag_t** enumeration type defines valid invoker values, which indicate how an outlined function is invoked.

The value **ompt_parallel_invoker_program** indicates that the outlined function associated with implicit tasks for the region is invoked directly by the application on the master thread for a parallel region.

The value **ompt_parallel_invoker_runtime** indicates that the outlined function associated with implicit tasks for the region is invoked by the runtime on the master thread for a parallel region.

The value **ompt_parallel_league** indicates that the callback is invoked due to the creation of a league of teams by a **teams** construct.

The value **ompt_parallel_team** indicates that the callback is invoked due to the creation of a team of threads by a **parallel** construct.

4.4.4.22 ompt_target_map_flag_t

Summary

The **ompt_target_map_flag_t** enumeration type defines the valid target map flag values.

Format

C / C++

```
typedef enum ompt_target_map_flag_t {  
    ompt_target_map_flag_to          = 0x01,  
    ompt_target_map_flag_from        = 0x02,  
    ompt_target_map_flag_alloc       = 0x04,  
    ompt_target_map_flag_release     = 0x08,  
    ompt_target_map_flag_delete      = 0x10,  
    ompt_target_map_flag_implicit    = 0x20  
} ompt_target_map_flag_t;
```

C / C++

4.4.4.23 ompt_dependence_type_t

Summary

The **ompt_dependence_type_t** enumeration type defines the valid task dependence type values.

Format

C / C++

```
typedef enum ompt_dependence_type_t {  
    ompt_dependence_type_in          = 1,  
    ompt_dependence_type_out         = 2,  
    ompt_dependence_type_inout       = 3,  
    ompt_dependence_type_mutexinoutset = 4,  
    ompt_dependence_type_source      = 5,  
    ompt_dependence_type_sink        = 6  
} ompt_dependence_type_t;
```

C / C++

4.4.4.24 ompt_cancel_flag_t

Summary

The **ompt_cancel_flag_t** enumeration type defines the valid cancel flag values.

Format

C / C++

```
typedef enum ompt_cancel_flag_t {  
    ompt_cancel_parallel      = 0x01,  
    ompt_cancel_sections     = 0x02,  
    ompt_cancel_loop         = 0x04,  
    ompt_cancel_taskgroup    = 0x08,  
    ompt_cancel_activated    = 0x10,  
    ompt_cancel_detected     = 0x20,  
    ompt_cancel_discarded_task = 0x40  
} ompt_cancel_flag_t;
```

C / C++

4.4.4.25 ompt_hwid_t

Summary

The **ompt_hwid_t** opaque type is a handle for a hardware identifier for a target device.

Format

C / C++

```
typedef uint64_t ompt_hwid_t;
```

C / C++

Description

The **ompt_hwid_t** opaque type is a handle for a hardware identifier for a target device.

ompt_hwid_none is an instance of the type that refers to an unknown or unspecified hardware identifier and that has the value 0. If no *hwid* is associated with an

ompt_record_abstract_t then the value of *hwid* is **ompt_hwid_none**.

Cross References

- **ompt_record_abstract_t** type, see Section [4.4.3.3](#) on page [436](#).

1 4.4.4.26 ompt_state_t

2 Summary

3 If the OMPT interface is in the *active* state then an OpenMP implementation must maintain *thread*
4 *state* information for each thread. The thread state maintained is an approximation of the
5 instantaneous state of a thread.

6 Format

C / C++

7 A thread state must be one of the values of the enumeration type **ompt_state_t** or an
8 implementation-defined state value of 512 or higher.

```
9 typedef enum ompt_state_t {  
10     ompt_state_work_serial          = 0x000,  
11     ompt_state_work_parallel        = 0x001,  
12     ompt_state_work_reduction       = 0x002,  
13  
14     ompt_state_wait_barrier         = 0x010,  
15     ompt_state_wait_barrier_implicit_parallel = 0x011,  
16     ompt_state_wait_barrier_implicit_workshare = 0x012,  
17     ompt_state_wait_barrier_implicit = 0x013,  
18     ompt_state_wait_barrier_explicit = 0x014,  
19  
20     ompt_state_wait_taskwait        = 0x020,  
21     ompt_state_wait_taskgroup       = 0x021,  
22  
23     ompt_state_wait_mutex           = 0x040,  
24     ompt_state_wait_lock            = 0x041,  
25     ompt_state_wait_critical        = 0x042,  
26     ompt_state_wait_atomic          = 0x043,  
27     ompt_state_wait_ordered         = 0x044,  
28  
29     ompt_state_wait_target          = 0x080,  
30     ompt_state_wait_target_map      = 0x081,  
31     ompt_state_wait_target_update   = 0x082,  
32  
33     ompt_state_idle                 = 0x100,  
34     ompt_state_overhead              = 0x101,  
35     ompt_state_undefined            = 0x102  
36 } ompt_state_t;
```

C / C++

Description

A tool can query the OpenMP state of a thread at any time. If a tool queries the state of a thread that is not associated with OpenMP then the implementation reports the state as **ompt_state_undefined**.

The value **ompt_state_work_serial** indicates that the thread is executing code outside all **parallel** regions.

The value **ompt_state_work_parallel** indicates that the thread is executing code within the scope of a **parallel** region.

The value **ompt_state_work_reduction** indicates that the thread is combining partial reduction results from threads in its team. An OpenMP implementation may never report a thread in this state; a thread that is combining partial reduction results may have its state reported as **ompt_state_work_parallel** or **ompt_state_overhead**.

The value **ompt_state_wait_barrier** indicates that the thread is waiting at either an implicit or explicit barrier. An implementation may never report a thread in this state; instead, a thread may have its state reported as **ompt_state_wait_barrier_implicit** or **ompt_state_wait_barrier_explicit**, as appropriate.

The value **ompt_state_wait_barrier_implicit** indicates that the thread is waiting at an implicit barrier in a **parallel** region. An OpenMP implementation may report **ompt_state_wait_barrier** for implicit barriers.

The value **ompt_state_wait_barrier_implicit_parallel** indicates that the thread is waiting at an implicit barrier at the end of a **parallel** region. An OpenMP implementation may report **ompt_state_wait_barrier** or **ompt_state_wait_barrier_implicit** for these barriers.

The value **ompt_state_wait_barrier_implicit_workshare** indicates that the thread is waiting at an implicit barrier at the end of a worksharing construct. An OpenMP implementation may report **ompt_state_wait_barrier** or **ompt_state_wait_barrier_implicit** for these barriers.

The value **ompt_state_wait_barrier_explicit** indicates that the thread is waiting in a **barrier** region. An OpenMP implementation may report **ompt_state_wait_barrier** for these barriers.

The value **ompt_state_wait_taskwait** indicates that the thread is waiting at a **taskwait** construct.

The value **ompt_state_wait_taskgroup** indicates that the thread is waiting at the end of a **taskgroup** construct.

The value **ompt_state_wait_mutex** indicates that the thread is waiting for a mutex of an unspecified type.

The value **ompt_state_wait_lock** indicates that the thread is waiting for a lock or nestable lock.

The value **ompt_state_wait_critical** indicates that the thread is waiting to enter a **critical** region.

The value **ompt_state_wait_atomic** indicates that the thread is waiting to enter an **atomic** region.

The value **ompt_state_wait_ordered** indicates that the thread is waiting to enter an **ordered** region.

The value **ompt_state_wait_target** indicates that the thread is waiting for a **target** region to complete.

The value **ompt_state_wait_target_map** indicates that the thread is waiting for a target data mapping operation to complete. An implementation may report **ompt_state_wait_target** for **target data** constructs.

The value **ompt_state_wait_target_update** indicates that the thread is waiting for a **target update** operation to complete. An implementation may report **ompt_state_wait_target** for **target update** constructs.

The value **ompt_state_idle** indicates that the thread is idle, that is, it is not part of an OpenMP team.

The value **ompt_state_overhead** indicates that the thread is in the overhead state at any point while executing within the OpenMP runtime, except while waiting at a synchronization point.

The value **ompt_state_undefined** indicates that the native thread is not created by the OpenMP implementation.

4.4.4.27 ompt_frame_t

Summary

The **ompt_frame_t** type describes procedure frame information for an OpenMP task.

Format

```

C / C++
typedef struct ompt_frame_t {
    ompt_data_t exit_frame;
    ompt_data_t enter_frame;
    int exit_frame_flags;
    int enter_frame_flags;
} ompt_frame_t;
C / C++

```

Description

Each **ompt_frame_t** object is associated with the task to which the procedure frames belong. Each non-merged initial, implicit, explicit, or target task with one or more frames on the stack of a native thread has an associated **ompt_frame_t** object.

The *exit_frame* field of an **ompt_frame_t** object contains information to identify the first procedure frame executing the task region. The *exit_frame* for the **ompt_frame_t** object associated with the *initial task* that is not nested inside any OpenMP construct is **NULL**.

The *enter_frame* field of an **ompt_frame_t** object contains information to identify the latest still active procedure frame executing the task region before entering the OpenMP runtime implementation or before executing a different task. If a task with frames on the stack has not been suspended, the value of *enter_frame* for the **ompt_frame_t** object associated with the task may contain **NULL**.

For *exit_frame*, the *exit_frame_flags* and, for *enter_frame*, the *enter_frame_flags* field indicates that the provided frame information points to a runtime or an application frame address. The same fields also specify the kind of information that is provided to identify the frame. These fields are a disjunction of values in the **ompt_frame_flag_t** enumeration type.

The lifetime of an **ompt_frame_t** object begins when a task is created and ends when the task is destroyed. Tools should not assume that a frame structure remains at a constant location in memory throughout the lifetime of the task. A pointer to an **ompt_frame_t** object is passed to some callbacks; a pointer to the **ompt_frame_t** object of a task can also be retrieved by a tool at any time, including in a signal handler, by invoking the **ompt_get_task_info** runtime entry point (described in Section 4.6.1.14). A pointer to an **ompt_frame_t** object that a tool retrieved is valid as long as the tool does not pass back control to the OpenMP implementation.

▼
Note – A monitoring tool that uses asynchronous sampling can observe values of *exit_frame* and *enter_frame* at inconvenient times. Tools must be prepared to handle **ompt_frame_t** objects observed just prior to when their field values will be set or cleared.
▲

4.4.4.28 ompt_frame_flag_t

Summary

The **ompt_frame_flag_t** enumeration type defines valid frame information flags.

Format

C / C++

```
typedef enum ompt_frame_flag_t {  
    ompt_frame_runtime      = 0x00,  
    ompt_frame_application  = 0x01,  
    ompt_frame_cfa          = 0x10,  
    ompt_frame_framepointer = 0x20,  
    ompt_frame_stackaddress = 0x30  
} ompt_frame_flag_t;
```

C / C++

Description

The value **ompt_frame_runtime** of the **ompt_frame_flag_t** type indicates that a frame address is a procedure frame in the OpenMP runtime implementation. The value **ompt_frame_application** of the **ompt_frame_flag_t** type indicates that an exit frame address is a procedure frame in the OpenMP application.

Higher order bits indicate the kind of provided information that is unique for the particular frame pointer. The value **ompt_frame_cfa** indicates that a frame address specifies a *canonical frame address*. The value **ompt_frame_framepointer** indicates that a frame address provides the value of the frame pointer register. The value **ompt_frame_stackaddress** indicates that a frame address specifies a pointer address that is contained in the current stack frame.

4.4.4.29 ompt_wait_id_t

Summary

The **ompt_wait_id_t** type describes wait identifiers for an OpenMP thread.

Format

C / C++

```
typedef uint64_t ompt_wait_id_t;
```

C / C++

Description

Each thread maintains a *wait identifier* of type `ompt_wait_id_t`. When a task that a thread executes is waiting for mutual exclusion, the wait identifier of the thread indicates the reason that the thread is waiting. A wait identifier may represent a critical section *name*, a lock, a program variable accessed in an atomic region, or a synchronization object that is internal to an OpenMP implementation. When a thread is not in a wait state then the value of the wait identifier of the thread is undefined.

`ompt_wait_id_none` is defined as an instance of type `ompt_wait_id_t` with the value 0.

4.5 OMPT Tool Callback Signatures and Trace Records

The C/C++ header file (`omp-tools.h`) provides the definitions of the types that are specified throughout this subsection.

Restrictions

- Tool callbacks may not use OpenMP directives or call any runtime library routines described in Section 3.

4.5.1 Initialization and Finalization Callback Signature

4.5.1.1 `ompt_initialize_t`

Summary

A callback with type signature `ompt_initialize_t` initializes use of the OMPT interface.

Format

```
C / C++
typedef int (*ompt_initialize_t) (
    ompt_function_lookup_t lookup,
    int initial_device_num,
    ompt_data_t *tool_data
);
```

C / C++

Description

To use the OMPT interface, an implementation of `ompt_start_tool` must return a non-null pointer to an `ompt_start_tool_result_t` structure that contains a non-null pointer to a tool initializer with type signature `ompt_initialize_t`. An OpenMP implementation will call the initializer after fully initializing itself but before beginning execution of any OpenMP construct or completing execution of any environment routine invocation.

The initializer returns a non-zero value if it succeeds.

Description of Arguments

The *lookup* argument is a callback to an OpenMP runtime routine that must be used to obtain a pointer to each runtime entry point in the OMPT interface. The *initial_device_num* argument provides the value of `omp_get_initial_device()`. The *tool_data* argument is a pointer to the *tool_data* field in the `ompt_start_tool_result_t` structure that `ompt_start_tool` returned. The expected actions of an initializer are described in Section 4.2.3.

Cross References

- `omp_get_initial_device` routine, see Section 3.2.41 on page 376.
- `ompt_start_tool` function, see Section 4.2.1 on page 420.
- `ompt_start_tool_result_t` type, see Section 4.4.1 on page 433.
- `ompt_data_t` type, see Section 4.4.4.4 on page 440.
- `ompt_function_lookup_t` type, see Section 4.6.3 on page 531.

4.5.1.2 `ompt_finalize_t`

Summary

A tool implements a finalizer with the type signature `ompt_finalize_t` to finalize the tool's use of the OMPT interface.

Format

```
typedef void (*ompt_finalize_t) (  
    ompt_data_t *tool_data  
);
```

C / C++

Description

To use the OMPT interface, an implementation of `ompt_start_tool` must return a non-null pointer to an `ompt_start_tool_result_t` structure that contains a non-null pointer to a tool finalizer with type signature `ompt_finalize_t`. An OpenMP implementation will call the tool finalizer after the last OMPT *event* as the OpenMP implementation shuts down.

Description of Arguments

The `tool_data` argument is a pointer to the `tool_data` field in the `ompt_start_tool_result_t` structure returned by `ompt_start_tool`.

Cross References

- `ompt_start_tool` function, see Section 4.2.1 on page 420.
- `ompt_start_tool_result_t` type, see Section 4.4.1 on page 433.
- `ompt_data_t` type, see Section 4.4.4.4 on page 440.

4.5.2 Event Callback Signatures and Trace Records

This section describes the signatures of tool callback functions that an OMPT tool may register and that are called during runtime of an OpenMP program. An implementation may also provide a trace of events per device. Along with the callbacks, the following defines standard trace records. For the trace records, tool data arguments are replaced by an ID, which must be initialized by the OpenMP implementation. Each of *parallel_id*, *task_id*, and *thread_id* must be unique per target region. Tool implementations of callbacks are not required to be *async signal safe*.

Cross References

- `ompt_id_t` type, see Section 4.4.4.3 on page 439.
- `ompt_data_t` type, see Section 4.4.4.4 on page 440.

4.5.2.1 `ompt_callback_thread_begin_t`

Summary

The `ompt_callback_thread_begin_t` type is used for callbacks that are dispatched when native threads are created.

Format

C / C++

```
typedef void (*ompt_callback_thread_begin_t) (  
    ompt_thread_t thread_type,  
    ompt_data_t *thread_data  
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_thread_begin_t {  
    ompt_thread_t thread_type;  
} ompt_record_thread_begin_t;
```

C / C++

Description of Arguments

The *thread_type* argument indicates the type of the new thread: initial, worker, or other. The binding of the *thread_data* argument is the new thread.

Cross References

- **parallel** construct, see Section 2.6 on page 74.
- **teams** construct, see Section 2.7 on page 82.
- Initial task, see Section 2.10.5 on page 148.
- **ompt_data_t** type, see Section 4.4.4.4 on page 440.
- **ompt_thread_t** type, see Section 4.4.4.10 on page 443.

4.5.2.2 ompt_callback_thread_end_t

Summary

The **ompt_callback_thread_end_t** type is used for callbacks that are dispatched when native threads are destroyed.

Format

```
typedef void (*ompt_callback_thread_end_t) (  
    ompt_data_t *thread_data  
);
```

Description of Arguments

The binding of the *thread_data* argument is the thread that will be destroyed.

Cross References

- **parallel** construct, see Section 2.6 on page 74.
- **teams** construct, see Section 2.7 on page 82.
- Initial task, see Section 2.10.5 on page 148.
- **ompt_record_ompt_t** type, see Section 4.4.3.4 on page 436.
- **ompt_data_t** type, see Section 4.4.4.4 on page 440.

4.5.2.3 ompt_callback_parallel_begin_t

Summary

The **ompt_callback_parallel_begin_t** type is used for callbacks that are dispatched when **parallel** and **teams** regions start.

Format

```
typedef void (*ompt_callback_parallel_begin_t) (  
    ompt_data_t *encountering_task_data,  
    const ompt_frame_t *encountering_task_frame,  
    ompt_data_t *parallel_data,  
    unsigned int requested_parallelism,  
    int flags,  
    const void *codeptr_ra  
);
```

Trace Record

C / C++

```
typedef struct ompt_record_parallel_begin_t {  
    ompt_id_t encountering_task_id;  
    ompt_id_t parallel_id;  
    unsigned int requested_parallelism;  
    int flags;  
    const void *codeptr_ra;  
} ompt_record_parallel_begin_t;
```

C / C++

Description of Arguments

The binding of the *encountering_task_data* argument is the encountering task.

The *encountering_task_frame* argument points to the frame object that is associated with the encountering task.

The binding of the *parallel_data* argument is the **parallel** or **teams** region that is beginning.

The *requested_parallelism* argument indicates the number of threads or teams that the user requested.

The *flags* argument indicates whether the code for the region is inlined into the application or invoked by the runtime and also whether the region is a **parallel** or **teams** region. Valid values for *flags* are a disjunction of elements in the enum **ompt_parallel_flag_t**.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_parallel_begin_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

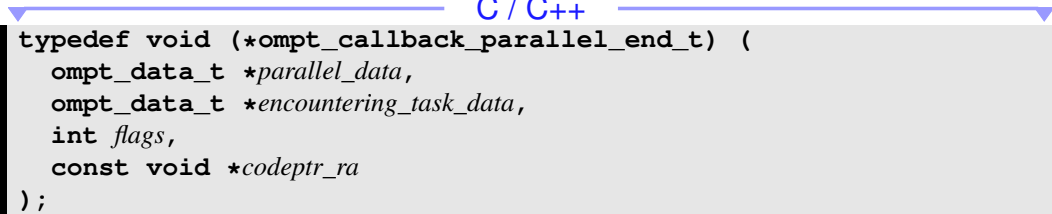
- **parallel** construct, see Section 2.6 on page 74.
- **teams** construct, see Section 2.7 on page 82.
- **ompt_data_t** type, see Section 4.4.4.4 on page 440.
- **ompt_parallel_flag_t** type, see Section 4.4.4.21 on page 448.
- **ompt_frame_t** type, see Section 4.4.4.27 on page 454.

1 4.5.2.4 ompt_callback_parallel_end_t

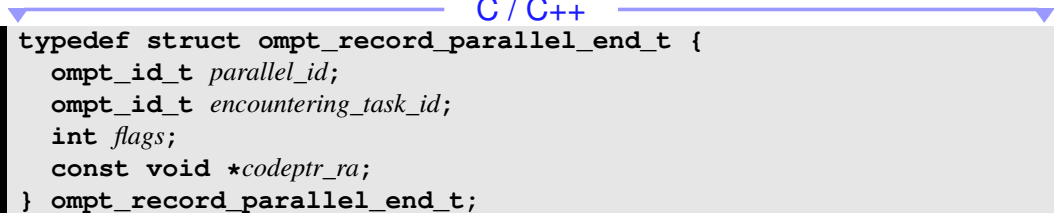
2 Summary

3 The **ompt_callback_parallel_end_t** type is used for callbacks that are dispatched when
4 **parallel** and **teams** regions ends.

5 Format

6  C / C++
7 `typedef void (*ompt_callback_parallel_end_t) (
8 ompt_data_t *parallel_data,
9 ompt_data_t *encountering_task_data,
10 int flags,
11 const void *codeptr_ra
12);`

12 Trace Record

13  C / C++
14 `typedef struct ompt_record_parallel_end_t {
15 ompt_id_t parallel_id;
16 ompt_id_t encountering_task_id;
17 int flags;
18 const void *codeptr_ra;
19 } ompt_record_parallel_end_t;`

19 Description of Arguments

20 The binding of the *parallel_data* argument is the **parallel** or **teams** region that is ending.

21 The binding of the *encountering_task_data* argument is the encountering task.

22 The *flags* argument indicates whether the execution of the region is inlined into the application or
23 invoked by the runtime and also whether it is a **parallel** or **teams** region. Values for *flags* are a
24 disjunction of elements in the enum **ompt_parallel_flag_t**.

25 The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a
26 runtime routine implements the region associated with a callback that has type signature
27 **ompt_callback_parallel_end_t** then *codeptr_ra* contains the return address of the call to
28 that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the
29 return address of the invocation of the callback. If attribution to source code is impossible or
30 inappropriate, *codeptr_ra* may be **NULL**.

Cross References

- **parallel** construct, see Section 2.6 on page 74.
- **teams** construct, see Section 2.7 on page 82.
- **ompt_data_t** type, see Section 4.4.4.4 on page 440.
- **ompt_parallel_flag_t** type, see Section 4.4.4.21 on page 448.

4.5.2.5 ompt_callback_work_t

Summary

The **ompt_callback_work_t** type is used for callbacks that are dispatched when worksharing regions, loop-related regions, and **taskloop** regions begin and end.

Format

C / C++

```
typedef void (*ompt_callback_work_t) (  
    ompt_work_t wstype,  
    ompt_scope_endpoint_t endpoint,  
    ompt_data_t *parallel_data,  
    ompt_data_t *task_data,  
    uint64_t count,  
    const void *codeptr_ra  
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_work_t {  
    ompt_work_t wstype;  
    ompt_scope_endpoint_t endpoint;  
    ompt_id_t parallel_id;  
    ompt_id_t task_id;  
    uint64_t count;  
    const void *codeptr_ra;  
} ompt_record_work_t;
```

C / C++

Description of Arguments

The *wstype* argument indicates the kind of region.

The *endpoint* argument indicates that the callback signals the beginning of a scope or the end of a scope.

The binding of the *parallel_data* argument is the current parallel region.

The binding of the *task_data* argument is the current task.

The *count* argument is a measure of the quantity of work involved in the construct. For a worksharing-loop construct, *count* represents the number of iterations of the loop. For a **taskloop** construct, *count* represents the number of iterations in the iteration space, which may be the result of collapsing several associated loops. For a **sections** construct, *count* represents the number of sections. For a **workshare** construct, *count* represents the units of work, as defined by the **workshare** construct. For a **single** construct, *count* is always 1. When the *endpoint* argument signals the end of a scope, a *count* value of 0 indicates that the actual *count* value is not available.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_work_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

- Worksharing constructs, see Section 2.8 on page 86 and Section 2.9.2 on page 101.
- Loop-related constructs, see Section 2.9 on page 95.
- **taskloop** construct, see Section 2.10.2 on page 140.
- **ompt_data_t** type, see Section 4.4.4.4 on page 440.
- **ompt_scope_endpoint_t** type, see Section 4.4.4.11 on page 443.
- **ompt_work_t** type, see Section 4.4.4.15 on page 445.

4.5.2.6 **ompt_callback_dispatch_t**

Summary

The **ompt_callback_dispatch_t** type is used for callbacks that are dispatched when a thread begins to execute a section or loop iteration.

Format

```
C / C++
typedef void (*ompt_callback_dispatch_t) (
    ompt_data_t *parallel_data,
    ompt_data_t *task_data,
    ompt_dispatch_t kind,
    ompt_data_t instance
);
```

Trace Record

```
C / C++
typedef struct ompt_record_dispatch_t {
    ompt_id_t parallel_id;
    ompt_id_t task_id;
    ompt_dispatch_t kind;
    ompt_data_t instance;
} ompt_record_dispatch_t;
```

Description of Arguments

The binding of the *parallel_data* argument is the current parallel region.

The binding of the *task_data* argument is the implicit task that executes the structured block of the parallel region.

The *kind* argument indicates whether a loop iteration or a section is being dispatched.

For a loop iteration, the *instance.value* argument contains the iteration variable value. For a structured block in the **sections** construct, *instance.ptr* contains a code address that identifies the structured block. In cases where a runtime routine implements the structured block associated with this callback, *instance.ptr* contains the return address of the call to the runtime routine. In cases where the implementation of the structured block is inlined, *instance.ptr* contains the return address of the invocation of this callback.

Cross References

- **sections** and **section** constructs, see Section 2.8.1 on page 86.
- Worksharing-loop construct, see Section 2.9.2 on page 101.
- **taskloop** construct, see Section 2.10.2 on page 140.
- **ompt_data_t** type, see Section 4.4.4.4 on page 440.
- **ompt_dispatch_t** type, see Section 4.4.4.12 on page 444.

4.5.2.7 ompt_callback_task_create_t

Summary

The **ompt_callback_task_create_t** type is used for callbacks that are dispatched when **task** regions or initial tasks are generated.

Format

C / C++

```
typedef void (*ompt_callback_task_create_t) (  
    ompt_data_t *encountering_task_data,  
    const ompt_frame_t *encountering_task_frame,  
    ompt_data_t *new_task_data,  
    int flags,  
    int has_dependences,  
    const void *codeptr_ra  
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_task_create_t {  
    ompt_id_t encountering_task_id;  
    ompt_id_t new_task_id;  
    int flags;  
    int has_dependences;  
    const void *codeptr_ra;  
} ompt_record_task_create_t;
```

C / C++

Description of Arguments

The binding of the *encountering_task_data* argument is the encountering task. This argument is **NULL** for an initial task.

The *encountering_task_frame* argument points to the frame object associated with the encountering task. This argument is **NULL** for an initial task.

The binding of the *new_task_data* argument is the generated task.

The *flags* argument indicates the kind of the task (initial, explicit, or target) that is generated. Values for *flags* are a disjunction of elements in the **ompt_task_flag_t** enumeration type.

The *has_dependences* argument is *true* if the generated task has dependences and *false* otherwise.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_task_create_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

- **task** construct, see Section 2.10.1 on page 135.
- Initial task, see Section 2.10.5 on page 148.
- **ompt_data_t** type, see Section 4.4.4.4 on page 440.
- **ompt_task_flag_t** type, see Section 4.4.4.18 on page 446.
- **ompt_frame_t** type, see Section 4.4.4.27 on page 454.

4.5.2.8 ompt_callback_dependences_t

Summary

The **ompt_callback_dependences_t** type is used for callbacks that are related to dependences and that are dispatched when new tasks are generated and when **ordered** constructs are encountered.

Format

```
typedef void (*ompt_callback_dependencies_t) (  
    ompt_data_t *task_data,  
    const ompt_dependence_t *deps,  
    int ndeps  
);
```

Trace Record

```
typedef struct ompt_record_dependencies_t {  
    ompt_id_t task_id;  
    ompt_dependence_t dep;  
    int ndeps;  
} ompt_record_dependencies_t;
```

Description of Arguments

The binding of the *task_data* argument is the generated task.

The *deps* argument lists dependences of the new task or the dependence vector of the ordered construct.

The *ndeps* argument specifies the length of the list passed by the *deps* argument. The memory for *deps* is owned by the caller; the tool cannot rely on the data after the callback returns.

The performance monitor interface for tracing activity on target devices provides one record per dependence.

Cross References


- **ordered** construct, see Section 2.17.9 on page 250.
- **depend** clause, see Section 2.17.11 on page 255.
- **ompt_data_t** type, see Section 4.4.4.4 on page 440.
- **ompt_dependence_t** type, see Section 4.4.4.9 on page 442.

1 4.5.2.9 `ompt_callback_task_dependence_t`


2 Summary

3 The `ompt_callback_task_dependence_t` type is used for callbacks that are dispatched
4 when unfulfilled task dependences are encountered.

5 Format

6 
7 `typedef void (*ompt_callback_task_dependence_t) (
8 ompt_data_t *src_task_data,
9 ompt_data_t *sink_task_data
10);`

10 Trace Record

11 
12 `typedef struct ompt_record_task_dependence_t {
13 ompt_id_t src_task_id;
14 ompt_id_t sink_task_id;
15 } ompt_record_task_dependence_t;`

15 Description of Arguments

16 The binding of the `src_task_data` argument is a running task with an outgoing dependence.

17 The binding of the `sink_task_data` argument is a task with an unsatisfied incoming dependence.

18 Cross References

- 19 • `depend` clause, see Section [2.17.11](#) on page [255](#).
- 20 • `ompt_data_t` type, see Section [4.4.4.4](#) on page [440](#).

21 4.5.2.10 `ompt_callback_task_schedule_t`

22 Summary

23 The `ompt_callback_task_schedule_t` type is used for callbacks that are dispatched when
24 task scheduling decisions are made.

Format

C / C++

```
typedef void (*ompt_callback_task_schedule_t) (  
    ompt_data_t *prior_task_data,  
    ompt_task_status_t prior_task_status,  
    ompt_data_t *next_task_data  
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_task_schedule_t {  
    ompt_id_t prior_task_id;  
    ompt_task_status_t prior_task_status;  
    ompt_id_t next_task_id;  
} ompt_record_task_schedule_t;
```

C / C++

Description of Arguments

The *prior_task_status* argument indicates the status of the task that arrived at a task scheduling point.

The binding of the *prior_task_data* argument is the task that arrived at the scheduling point.

The binding of the *next_task_data* argument is the task that is resumed at the scheduling point.

This argument is **NULL** if the callback is dispatched for a *task-fulfill* event.

Cross References

- Task scheduling, see Section 2.10.6 on page 149.
- **ompt_data_t** type, see Section 4.4.4.4 on page 440.
- **ompt_task_status_t** type, see Section 4.4.4.19 on page 447.

4.5.2.11 ompt_callback_implicit_task_t

Summary

The **ompt_callback_implicit_task_t** type is used for callbacks that are dispatched when initial tasks and implicit tasks are generated and completed.

Format

C / C++

```
typedef void (*ompt_callback_implicit_task_t) (  
    ompt_scope_endpoint_t endpoint,  
    ompt_data_t *parallel_data,  
    ompt_data_t *task_data,  
    unsigned int actual_parallelism,  
    unsigned int index,  
    int flags  
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_implicit_task_t {  
    ompt_scope_endpoint_t endpoint;  
    ompt_id_t parallel_id;  
    ompt_id_t task_id;  
    unsigned int actual_parallelism;  
    unsigned int index;  
    int flags;  
} ompt_record_implicit_task_t;
```

C / C++

Description of Arguments

The *endpoint* argument indicates that the callback signals the beginning of a scope or the end of a scope.

The binding of the *parallel_data* argument is the current parallel region. For the *implicit-task-end* event, this argument is **NULL**.

The binding of the *task_data* argument is the implicit task that executes the structured block of the parallel region.

The *actual_parallelism* argument indicates the number of threads in the **parallel** region or the number of teams in the **teams** region. For initial tasks, that are not closely nested in a **teams** construct, this argument is **1**. For the *implicit-task-end* and the *initial-task-end* events, this argument is **0**.

The *index* argument indicates the thread number or team number of the calling thread, within the team or league that is executing the parallel or **teams** region to which the implicit task region binds. For initial tasks, that are not created by a **teams** construct, this argument is **1**.

The *flags* argument indicates the kind of the task (initial or implicit).

Cross References

- **parallel** construct, see Section 2.6 on page 74.
- **teams** construct, see Section 2.7 on page 82.
- **ompt_data_t** type, see Section 4.4.4.4 on page 440.
- **ompt_scope_endpoint_t** enumeration type, see Section 4.4.4.11 on page 443.

4.5.2.12 ompt_callback_master_t

Summary

The **ompt_callback_master_t** type is used for callbacks that are dispatched when **master** regions start and end.

Format

C / C++

```
typedef void (*ompt_callback_master_t) (  
    ompt_scope_endpoint_t endpoint,  
    ompt_data_t *parallel_data,  
    ompt_data_t *task_data,  
    const void *codeptr_ra  
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_master_t {  
    ompt_scope_endpoint_t endpoint;  
    ompt_id_t parallel_id;  
    ompt_id_t task_id;  
    const void *codeptr_ra;  
} ompt_record_master_t;
```

C / C++

Description of Arguments

The *endpoint* argument indicates that the callback signals the beginning of a scope or the end of a scope.

The binding of the *parallel_data* argument is the current parallel region.

The binding of the *task_data* argument is the encountering task.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_master_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

- **master** construct, see Section 2.16 on page 221.
- **ompt_data_t** type, see Section 4.4.4.4 on page 440.
- **ompt_scope_endpoint_t** type, see Section 4.4.4.11 on page 443.

4.5.2.13 ompt_callback_sync_region_t

Summary

The **ompt_callback_sync_region_t** type is used for callbacks that are dispatched when barrier regions, **taskwait** regions, and **taskgroup** regions begin and end and when waiting begins and ends for them as well as for when reductions are performed.

Format

```
C / C++
typedef void (*ompt_callback_sync_region_t) (
    ompt_sync_region_t kind,
    ompt_scope_endpoint_t endpoint,
    ompt_data_t *parallel_data,
    ompt_data_t *task_data,
    const void *codeptr_ra
);
C / C++
```

Trace Record

C / C++

```
typedef struct ompt_record_sync_region_t {  
    ompt_sync_region_t kind;  
    ompt_scope_endpoint_t endpoint;  
    ompt_id_t parallel_id;  
    ompt_id_t task_id;  
    const void *codeptr_ra;  
} ompt_record_sync_region_t;
```

C / C++

Description of Arguments

The *kind* argument indicates the kind of synchronization.

The *endpoint* argument indicates that the callback signals the beginning of a scope or the end of a scope.

The binding of the *parallel_data* argument is the current parallel region. For the *barrier-end* event at the end of a parallel region this argument is **NULL**.

The binding of the *task_data* argument is the current task.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_sync_region_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

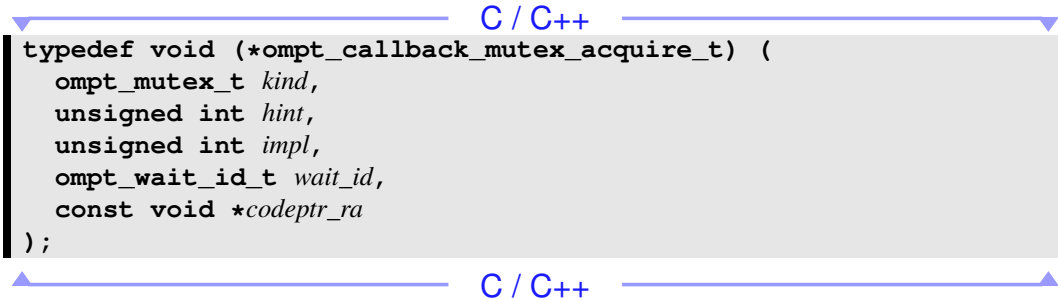
- **barrier** construct, see Section [2.17.2](#) on page [226](#).
- Implicit barriers, see Section [2.17.3](#) on page [228](#).
- **taskwait** construct, see Section [2.17.5](#) on page [230](#).
- **taskgroup** construct, see Section [2.17.6](#) on page [232](#).
- Properties common to all reduction clauses, see Section [2.19.5.1](#) on page [294](#).
- **ompt_data_t** type, see Section [4.4.4.4](#) on page [440](#).
- **ompt_scope_endpoint_t** type, see Section [4.4.4.11](#) on page [443](#).
- **ompt_sync_region_t** type, see Section [4.4.4.13](#) on page [444](#).

1 4.5.2.14 ompt_callback_mutex_acquire_t

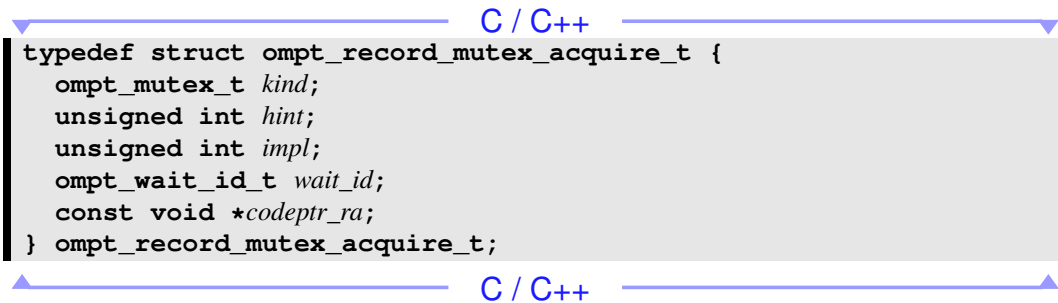
2 Summary

3 The **ompt_callback_mutex_acquire_t** type is used for callbacks that are dispatched when
4 locks are initialized, acquired and tested and when **critical** regions, **atomic** regions, and
5 **ordered** regions are begun.

6 Format

7  C / C++
8 `typedef void (*ompt_callback_mutex_acquire_t) (
9 ompt_mutex_t kind,
10 unsigned int hint,
11 unsigned int impl,
12 ompt_wait_id_t wait_id,
13 const void *codeptr_ra
14);`

14 Trace Record

15  C / C++
16 `typedef struct ompt_record_mutex_acquire_t {
17 ompt_mutex_t kind;
18 unsigned int hint;
19 unsigned int impl;
20 ompt_wait_id_t wait_id;
21 const void *codeptr_ra;
22 } ompt_record_mutex_acquire_t;`

22 Description of Arguments

23 The *kind* argument indicates the kind of the lock involved.

24 The *hint* argument indicates the hint that was provided when initializing an implementation of
25 mutual exclusion. If no hint is available when a thread initiates acquisition of mutual exclusion, the
26 runtime may supply **omp_sync_hint_none** as the value for *hint*.

27 The *impl* argument indicates the mechanism chosen by the runtime to implement the mutual
28 exclusion.

The *wait_id* argument indicates the object being awaited.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_mutex_acquire_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

- **critical** construct, see Section 2.17.1 on page 223.
- **atomic** construct, see Section 2.17.7 on page 234.
- **ordered** construct, see Section 2.17.9 on page 250.
- **omp_init_lock** and **omp_init_nest_lock** routines, see Section 3.3.1 on page 384.
- **ompt_mutex_t** type, see Section 4.4.4.16 on page 445.
- **ompt_wait_id_t** type, see Section 4.4.4.29 on page 456.

4.5.2.15 ompt_callback_mutex_t

Summary

The **ompt_callback_mutex_t** type is used for callbacks that indicate important synchronization events.

Format

```
typedef void (*ompt_callback_mutex_t) (  
    ompt_mutex_t kind,  
    ompt_wait_id_t wait_id,  
    const void *codeptr_ra  
);
```

Trace Record

C / C++

```
typedef struct ompt_record_mutex_t {  
    ompt_mutex_t kind;  
    ompt_wait_id_t wait_id;  
    const void *codeptr_ra;  
} ompt_record_mutex_t;
```

C / C++

Description of Arguments

The *kind* argument indicates the kind of mutual exclusion event.

The *wait_id* argument indicates the object being awaited.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_mutex_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

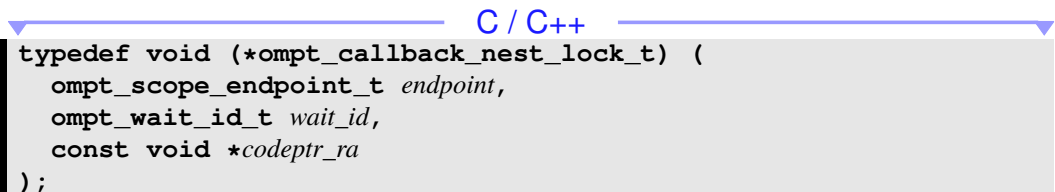
- **critical** construct, see Section 2.17.1 on page 223.
- **atomic** construct, see Section 2.17.7 on page 234.
- **ordered** construct, see Section 2.17.9 on page 250.
- **omp_destroy_lock** and **omp_destroy_nest_lock** routines, see Section 3.3.3 on page 387.
- **omp_set_lock** and **omp_set_nest_lock** routines, see Section 3.3.4 on page 388.
- **omp_unset_lock** and **omp_unset_nest_lock** routines, see Section 3.3.5 on page 390.
- **omp_test_lock** and **omp_test_nest_lock** routines, see Section 3.3.6 on page 392.
- **ompt_mutex_t** type, see Section 4.4.4.16 on page 445.
- **ompt_wait_id_t** type, see Section 4.4.4.29 on page 456.

1 4.5.2.16 ompt_callback_nest_lock_t

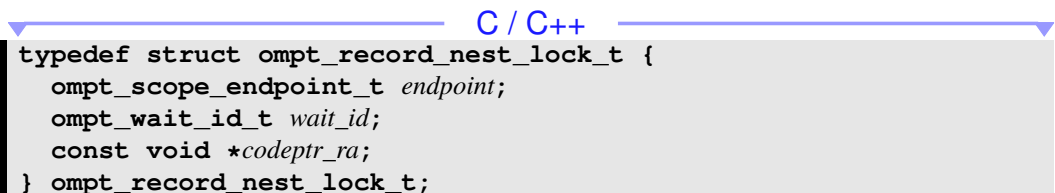
2 Summary

3 The **ompt_callback_nest_lock_t** type is used for callbacks that indicate that a thread that
4 owns a nested lock has performed an action related to the lock but has not relinquished ownership
5 of it.

6 Format

7  C / C++
8 `typedef void (*ompt_callback_nest_lock_t) (
9 ompt_scope_endpoint_t endpoint,
10 ompt_wait_id_t wait_id,
11 const void *codeptr_ra
);`

12 Trace Record

13  C / C++
14 `typedef struct ompt_record_nest_lock_t {`
15 `ompt_scope_endpoint_t endpoint;`
16 `ompt_wait_id_t wait_id;`
17 `const void *codeptr_ra;`
`} ompt_record_nest_lock_t;`

18 Description of Arguments

19 The *endpoint* argument indicates that the callback signals the beginning of a scope or the end of a
20 scope.

21 The *wait_id* argument indicates the object being awaited.

22 The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a
23 runtime routine implements the region associated with a callback that has type signature
24 **ompt_callback_nest_lock_t** then *codeptr_ra* contains the return address of the call to that
25 runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return
26 address of the invocation of the callback. If attribution to source code is impossible or
27 inappropriate, *codeptr_ra* may be **NULL**.

Cross References

- `omp_set_nest_lock` routine, see Section 3.3.4 on page 388.
- `omp_unset_nest_lock` routine, see Section 3.3.5 on page 390.
- `omp_test_nest_lock` routine, see Section 3.3.6 on page 392.
- `ompt_scope_endpoint_t` type, see Section 4.4.4.11 on page 443.
- `ompt_wait_id_t` type, see Section 4.4.4.29 on page 456.

4.5.2.17 `ompt_callback_flush_t`

Summary

The `ompt_callback_flush_t` type is used for callbacks that are dispatched when **flush** constructs are encountered.

Format

C / C++

```
typedef void (*ompt_callback_flush_t) (  
    ompt_data_t *thread_data,  
    const void *codeptr_ra  
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_flush_t {  
    const void *codeptr_ra;  
} ompt_record_flush_t;
```

C / C++

Description of Arguments

The binding of the *thread_data* argument is the executing thread.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature `ompt_callback_flush_t` then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

- **flush** construct, see Section 2.17.8 on page 242.
- **ompt_data_t** type, see Section 4.4.4.4 on page 440.

4.5.2.18 ompt_callback_cancel_t

Summary

The **ompt_callback_cancel_t** type is used for callbacks that are dispatched for *cancellation*, *cancel* and *discarded-task* events.

Format

```
C / C++
typedef void (*ompt_callback_cancel_t) (
    ompt_data_t *task_data,
    int flags,
    const void *codeptr_ra
);
```

Trace Record

```
C / C++
typedef struct ompt_record_cancel_t {
    ompt_id_t task_id;
    int flags;
    const void *codeptr_ra;
} ompt_record_cancel_t;
```

Description of Arguments

The binding of the *task_data* argument is the task that encounters a **cancel** construct, a **cancellation point** construct, or a construct defined as having an implicit cancellation point.

The *flags* argument, defined by the **ompt_cancel_flag_t** enumeration type, indicates whether cancellation is activated by the current task, or detected as being activated by another task. The construct that is being canceled is also described in the *flags* argument. When several constructs are detected as being concurrently canceled, each corresponding bit in the argument will be set.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_cancel_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

- **ompt_cancel_flag_t** enumeration type, see Section 4.4.4.24 on page 450.

4.5.2.19 ompt_callback_device_initialize_t

Summary

The **ompt_callback_device_initialize_t** type is used for callbacks that initialize device tracing interfaces.

Format

C / C++

```
typedef void (*ompt_callback_device_initialize_t) (  
    int device_num,  
    const char *type,  
    ompt_device_t *device,  
    ompt_function_lookup_t lookup,  
    const char *documentation  
);
```

C / C++

Description

Registration of a callback with type signature **ompt_callback_device_initialize_t** for the **ompt_callback_device_initialize** event enables asynchronous collection of a trace for a device. The OpenMP implementation invokes this callback after OpenMP is initialized for the device but before execution of any OpenMP construct is started on the device.

Description of Arguments

The *device_num* argument identifies the logical device that is being initialized.

The *type* argument is a character string that indicates the type of the device. A device type string is a semicolon separated character string that includes at a minimum the vendor and model name of the device. These names may be followed by a semicolon-separated sequence of properties that describe the hardware or software of the device.

The *device* argument is a pointer to an opaque object that represents the target device instance. Functions in the device tracing interface use this pointer to identify the device that is being addressed.

The *lookup* argument points to a runtime callback that a tool must use to obtain pointers to runtime entry points in the device's OMPT tracing interface. If a device does not support tracing then *lookup* is **NULL**.

The *documentation* argument is a string that describes how to use any device-specific runtime entry points that can be obtained through the *lookup* argument. This documentation string may be a pointer to external documentation, or it may be inline descriptions that include names and type signatures for any device-specific interfaces that are available through the *lookup* argument along with descriptions of how to use these interface functions to control monitoring and analysis of device traces.

Constraints on Arguments

The *type* and *documentation* arguments must be immutable strings that are defined for the lifetime of a program execution.

Effect

A device initializer must fulfill several duties. First, the *type* argument should be used to determine if any special knowledge about the hardware and/or software of a device is employed. Second, the *lookup* argument should be used to look up pointers to runtime entry points in the OMPT tracing interface for the device. Finally, these runtime entry points should be used to set up tracing for the device.

Initialization of tracing for a target device is described in Section 4.2.5 on page 427.

Cross References

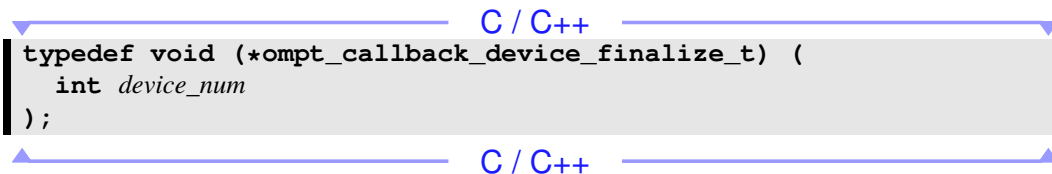
- `ompt_function_lookup_t` type, see Section 4.6.3 on page 531.

1 4.5.2.20 `ompt_callback_device_finalize_t`

2 Summary

3 The `ompt_callback_device_initialize_t` type is used for callbacks that finalize device
4 tracing interfaces.

5 Format

```
6  typedef void (*ompt_callback_device_finalize_t) (  
7     int device_num  
8 );
```

9 Description of Arguments

10 The *device_num* argument identifies the logical device that is being finalized.

11 Description

12 A registered callback with type signature `ompt_callback_device_finalize_t` is
13 dispatched for a device immediately prior to finalizing the device. Prior to dispatching a finalization
14 callback for a device on which tracing is active, the OpenMP implementation stops tracing on the
15 device and synchronously flushes all trace records for the device that have not yet been reported.
16 These trace records are flushed through one or more buffer completion callbacks with type
17 signature `ompt_callback_buffer_complete_t` as needed prior to the dispatch of the
18 callback with type signature `ompt_callback_device_finalize_t`.

19 Cross References

- 20 • `ompt_callback_buffer_complete_t` callback type, see Section [4.5.2.24](#) on page [487](#).

21 4.5.2.21 `ompt_callback_device_load_t`

22 Summary

23 The `ompt_callback_device_load_t` type is used for callbacks that the OpenMP runtime
24 invokes to indicate that it has just loaded code onto the specified device.

Format

C / C++

```
typedef void (*ompt_callback_device_load_t) (  
    int device_num,  
    const char *filename,  
    int64_t offset_in_file,  
    void *vma_in_file,  
    size_t bytes,  
    void *host_addr,  
    void *device_addr,  
    uint64_t module_id  
);
```

C / C++

Description of Arguments

The *device_num* argument specifies the device.

The *filename* argument indicates the name of a file in which the device code can be found. A NULL *filename* indicates that the code is not available in a file in the file system.

The *offset_in_file* argument indicates an offset into *filename* at which the code can be found. A value of -1 indicates that no offset is provided.

ompt_addr_none is defined as a pointer with the value ~0.

The *vma_in_file* argument indicates an virtual address in *filename* at which the code can be found. A value of **ompt_addr_none** indicates that a virtual address in the file is not available.

The *bytes* argument indicates the size of the device code object in bytes.

The *host_addr* argument indicates the address at which a copy of the device code is available in host memory. A value of **ompt_addr_none** indicates that a host code address is not available.

The *device_addr* argument indicates the address at which the device code has been loaded in device memory. A value of **ompt_addr_none** indicates that a device code address is not available.

The *module_id* argument is an identifier that is associated with the device code object.

Cross References


- Device directives, see Section [2.12](#) on page [160](#).

1 4.5.2.22 `ompt_callback_device_unload_t`

2 Summary

3 The `ompt_callback_device_unload_t` type is used for callbacks that the OpenMP
4 runtime invokes to indicate that it is about to unload code from the specified device.

5 Format

6 
7 `typedef void (*ompt_callback_device_unload_t) (`
8 `int device_num,`
9 `uint64_t module_id`
`);`

10 Description of Arguments

11 The *device_num* argument specifies the device.

12 The *module_id* argument is an identifier that is associated with the device code object.

13 Cross References


- 14
 - Device directives, see Section [2.12](#) on page [160](#).

15 4.5.2.23 `ompt_callback_buffer_request_t`

16 Summary

17 The `ompt_callback_buffer_request_t` type is used for callbacks that are dispatched
18 when a buffer to store event records for a device is requested.

19 Format

20 
21 `typedef void (*ompt_callback_buffer_request_t) (`
22 `int device_num,`
23 `ompt_buffer_t **buffer,`
24 `size_t *bytes`
`);`

Description

A callback with type signature `ompt_callback_buffer_request_t` requests a buffer to store trace records for the specified device. A buffer request callback may set `*bytes` to 0 if it does not provide a buffer. If a callback sets `*bytes` to 0, further recording of events for the device is disabled until the next invocation of `ompt_start_trace`. This action causes the device to drop future trace records until recording is restarted.

Description of Arguments

The `device_num` argument specifies the device.

The `*buffer` argument points to a buffer where device events may be recorded. The `*bytes` argument indicates the length of that buffer.

Cross References

- `ompt_buffer_t` type, see Section 4.4.4.7 on page 441.

4.5.2.24 `ompt_callback_buffer_complete_t`

Summary

The `ompt_callback_buffer_complete_t` type is used for callbacks that are dispatched when devices will not record any more trace records in an event buffer and all records written to the buffer are valid.

Format

```
C / C++
typedef void (*ompt_callback_buffer_complete_t) (
    int device_num,
    ompt_buffer_t *buffer,
    size_t bytes,
    ompt_buffer_cursor_t begin,
    int buffer_owned
);
```

Description

A callback with type signature `ompt_callback_buffer_complete_t` provides a buffer that contains trace records for the specified device. Typically, a tool will iterate through the records in the buffer and process them.

The OpenMP implementation makes these callbacks on a thread that is not an OpenMP master or worker thread.

The callee may not delete the buffer if the *buffer_owned* argument is 0.

The buffer completion callback is not required to be *async signal safe*.

Description of Arguments

The *device_num* argument indicates the device which the buffer contains events.

The *buffer* argument is the address of a buffer that was previously allocated by a *buffer request* callback.

The *bytes* argument indicates the full size of the buffer.

The *begin* argument is an opaque cursor that indicates the position of the beginning of the first record in the buffer.

The *buffer_owned* argument is 1 if the data to which the buffer points can be deleted by the callback and 0 otherwise. If multiple devices accumulate trace events into a single buffer, this callback may be invoked with a pointer to one or more trace records in a shared buffer with *buffer_owned* = 0. In this case, the callback may not delete the buffer.

Cross References

- `ompt_buffer_t` type, see Section 4.4.4.7 on page 441.
- `ompt_buffer_cursor_t` type, see Section 4.4.4.8 on page 442.

4.5.2.25 `ompt_callback_target_data_op_t`

Summary

The `ompt_callback_target_data_op_t` type is used for callbacks that are dispatched when a thread maps data to a device.

Format

```
typedef void (*ompt_callback_target_data_op_t) (  
    ompt_id_t target_id,  
    ompt_id_t host_op_id,  
    ompt_target_data_op_t optype,  
    void *src_addr,  
    int src_device_num,  
    void *dest_addr,  
    int dest_device_num,  
    size_t bytes,  
    const void *codeptr_ra  
);
```

Trace Record

```
typedef struct ompt_record_target_data_op_t {  
    ompt_id_t host_op_id;  
    ompt_target_data_op_t optype;  
    void *src_addr;  
    int src_device_num;  
    void *dest_addr;  
    int dest_device_num;  
    size_t bytes;  
    ompt_device_time_t end_time;  
    const void *codeptr_ra;  
} ompt_record_target_data_op_t;
```

Description

A registered **ompt_callback_target_data_op** callback is dispatched when device memory is allocated or freed, as well as when data is copied to or from a device.

Note – An OpenMP implementation may aggregate program variables and data operations upon them. For instance, an OpenMP implementation may synthesize a composite to represent multiple scalars and then allocate, free, or copy this composite as a whole rather than performing data operations on each scalar individually. Thus, callbacks may not be dispatched as separate data operations on each variable.

Description of Arguments

The *host_op_id* argument is a unique identifier for a data operations on a target device.

The *optype* argument indicates the kind of data mapping.

The *src_addr* argument indicates the data address before the operation, where applicable.

The *src_device_num* argument indicates the source device number for the data operation, where applicable.

The *dest_addr* argument indicates the data address after the operation.

The *dest_device_num* argument indicates the destination device number for the data operation.

It is implementation defined whether in some operations *src_addr* or *dest_addr* may point to an intermediate buffer.

The *bytes* argument indicates the size of data.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_target_data_op_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

- **map** clause, see Section [2.19.7.1](#) on page [315](#).
- **ompt_id_t** type, see Section [4.4.4.3](#) on page [439](#).
- **ompt_target_data_op_t** type, see Section [4.4.4.14](#) on page [444](#).

4.5.2.26 ompt_callback_target_t

Summary

The **ompt_callback_target_t** type is used for callbacks that are dispatched when a thread begins to execute a device construct.

Format

C / C++

```
typedef void (*ompt_callback_target_t) (  
    ompt_target_t kind,  
    ompt_scope_endpoint_t endpoint,  
    int device_num,  
    ompt_data_t *task_data,  
    ompt_id_t target_id,  
    const void *codeptr_ra  
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_target_t {  
    ompt_target_t kind;  
    ompt_scope_endpoint_t endpoint;  
    int device_num;  
    ompt_id_t task_id;  
    ompt_id_t target_id;  
    const void *codeptr_ra;  
} ompt_record_target_t;
```

C / C++

Description of Arguments

The *kind* argument indicates the kind of target region.

The *endpoint* argument indicates that the callback signals the beginning of a scope or the end of a scope.

The *device_num* argument indicates the id of the device that will execute the target region.

The binding of the *task_data* argument is the generating task.

The binding of the *target_id* argument is the target region.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_target_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

- **target data** construct, see Section 2.12.2 on page 161.
- **target enter data** construct, see Section 2.12.3 on page 164.
- **target exit data** construct, see Section 2.12.4 on page 166.
- **target** construct, see Section 2.12.5 on page 170.
- **target update** construct, see Section 2.12.6 on page 176.
- **ompt_id_t** type, see Section 4.4.4.3 on page 439.
- **ompt_data_t** type, see Section 4.4.4.4 on page 440.
- **ompt_scope_endpoint_t** type, see Section 4.4.4.11 on page 443.
- **ompt_target_t** type, see Section 4.4.4.20 on page 448.

4.5.2.27 ompt_callback_target_map_t

Summary

The **ompt_callback_target_map_t** type is used for callbacks that are dispatched to indicate data mapping relationships.

Format

```
typedef void (*ompt_callback_target_map_t) (  
    ompt_id_t target_id,  
    unsigned int nitems,  
    void **host_addr,  
    void **device_addr,  
    size_t *bytes,  
    unsigned int *mapping_flags,  
    const void *codeptr_ra  
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_target_map_t {  
    ompt_id_t target_id;  
    unsigned int nitems;  
    void **host_addr;  
    void **device_addr;  
    size_t *bytes;  
    unsigned int *mapping_flags;  
    const void *codeptr_ra;  
} ompt_record_target_map_t;
```

C / C++

Description

An instance of a **target**, **target data**, **target enter data**, or **target exit data** construct may contain one or more **map** clauses. An OpenMP implementation may report the set of mappings associated with **map** clauses for a construct with a single **ompt_callback_target_map** callback to report the effect of all mappings or multiple **ompt_callback_target_map** callbacks with each reporting a subset of the mappings. Furthermore, an OpenMP implementation may omit mappings that it determines are unnecessary. If an OpenMP implementation issues multiple **ompt_callback_target_map** callbacks, these callbacks may be interleaved with **ompt_callback_target_data_op** callbacks used to report data operations associated with the mappings.

Description of Arguments

The binding of the *target_id* argument is the target region.

The *nitems* argument indicates the number of data mappings that this callback reports.

The *host_addr* argument indicates an array of host data addresses.

The *device_addr* argument indicates an array of device data addresses.

The *bytes* argument indicates an array of size of data.

The *mapping_flags* argument indicates the kind of data mapping. Flags for a mapping include one or more values specified by the **ompt_target_map_flag_t** type.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_target_map_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

- **target data** construct, see Section 2.12.2 on page 161.
- **target enter data** construct, see Section 2.12.3 on page 164.
- **target exit data** construct, see Section 2.12.4 on page 166.
- **target** construct, see Section 2.12.5 on page 170.
- **ompt_id_t** type, see Section 4.4.4.3 on page 439.
- **ompt_target_map_flag_t** type, see Section 4.4.4.22 on page 449.
- **ompt_callback_target_data_op_t** callback type, see Section 4.5.2.25 on page 488.

4.5.2.28 ompt_callback_target_submit_t

Summary

The **ompt_callback_target_submit_t** type is used for callbacks that are dispatched when an initial task is created on a device.

Format

C / C++

```
typedef void (*ompt_callback_target_submit_t) (  
    ompt_id_t target_id,  
    ompt_id_t host_op_id,  
    unsigned int requested_num_teams  
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_target_kernel_t {  
    ompt_id_t host_op_id;  
    unsigned int requested_num_teams;  
    unsigned int granted_num_teams;  
    ompt_device_time_t end_time;  
} ompt_record_target_kernel_t;
```

C / C++

Description

A thread dispatches a registered `ompt_callback_target_submit` callback on the host when a target task creates an initial task on a target device.

Description of Arguments

The *target_id* argument is a unique identifier for the associated target region.

The *host_op_id* argument is a unique identifier for the initial task on the target device.

The *requested_num_teams* argument is the number of teams that the host requested to execute the kernel. The actual number of teams that execute the kernel may be smaller and generally will not be known until the kernel begins to execute on the device.

If `ompt_set_trace_ompt` has configured the device to trace kernel execution then the device will log a `ompt_record_target_kernel_t` record in a trace. The fields in the record are as follows:

- The *host_op_id* field contains a unique identifier that can be used to correlate a `ompt_record_target_kernel_t` record with its associated `ompt_callback_target_submit` callback on the host;
- The *requested_num_teams* field contains the number of teams that the host requested to execute the kernel;
- The *granted_num_teams* field contains the number of teams that the device actually used to execute the kernel;
- The time when the initial task began execution on the device is recorded in the *time* field of an enclosing `ompt_record_t` structure; and
- The time when the initial task completed execution on the device is recorded in the *end_time* field.

Cross References

- `target` construct, see Section [2.12.5](#) on page [170](#).
- `ompt_id_t` type, see Section [4.4.4.3](#) on page [439](#).

4.5.2.29 `ompt_callback_control_tool_t`

Summary

The `ompt_callback_control_tool_t` type is used for callbacks that dispatch *tool-control* events.

Format

```
C / C++
typedef int (*ompt_callback_control_tool_t) (
    uint64_t command,
    uint64_t modifier,
    void *arg,
    const void *codeptr_ra
);
```

Trace Record

```
C / C++
typedef struct ompt_record_control_tool_t {
    uint64_t command;
    uint64_t modifier;
    const void *codeptr_ra;
} ompt_record_control_tool_t;
```

Description

Callbacks with type signature **ompt_callback_control_tool_t** may return any non-negative value, which will be returned to the application as the return value of the **omp_control_tool** call that triggered the callback.

Description of Arguments

The *command* argument passes a command from an application to a tool. Standard values for *command* are defined by **omp_control_tool_t** in Section 3.8 on page 415.

The *modifier* argument passes a command modifier from an application to a tool.

The *command* and *modifier* arguments may have tool-specific values. Tools must ignore *command* values that they are not designed to handle.

The *arg* argument is a void pointer that enables a tool and an application to exchange arbitrary state. The *arg* argument may be **NULL**.

The *codeptr_ra* argument relates the implementation of an OpenMP region to its source code. If a runtime routine implements the region associated with a callback that has type signature **ompt_callback_control_tool_t** then *codeptr_ra* contains the return address of the call to that runtime routine. If the implementation of the region is inlined then *codeptr_ra* contains the return address of the invocation of the callback. If attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Constraints on Arguments

Tool-specific values for *command* must be ≥ 64 .

Cross References

- `omp_control_tool_t` enumeration type, see Section 3.8 on page 415.

4.6 OMPT Runtime Entry Points for Tools

OMPT supports two principal sets of runtime entry points for tools. One set of runtime entry points enables a tool to register callbacks for OpenMP events and to inspect the state of an OpenMP thread while executing in a tool callback or a signal handler. The second set of runtime entry points enables a tool to trace activities on a device. When directed by the tracing interface, an OpenMP implementation will trace activities on a device, collect buffers of trace records, and invoke callbacks on the host to process these records. OMPT runtime entry points should not be global symbols since tools cannot rely on the visibility of such symbols.

OMPT also supports runtime entry points for two classes of lookup routines. The first class of lookup routines contains a single member: a routine that returns runtime entry points in the OMPT callback interface. The second class of lookup routines includes a unique lookup routine for each kind of device that can return runtime entry points in a device's OMPT tracing interface.

The C/C++ header file (`omp-tools.h`) provides the definitions of the types that are specified throughout this subsection.

Restrictions

OMPT runtime entry points have the following restrictions:

- OMPT runtime entry points must not be called from a signal handler on a native thread before a *native-thread-begin* or after a *native-thread-end* event.
- OMPT device runtime entry points must not be called after a *device-finalize* event for that device.

4.6.1 Entry Points in the OMPT Callback Interface

Entry points in the OMPT callback interface enable a tool to register callbacks for OpenMP events and to inspect the state of an OpenMP thread while executing in a tool callback or a signal handler. Pointers to these runtime entry points are obtained through the lookup function that is provided through the OMPT initializer.

1 4.6.1.1 `ompt_enumerate_states_t`

2 Summary

3 The `ompt_enumerate_states_t` type is the type signature of the
4 `ompt_enumerate_states` runtime entry point, which enumerates the thread states that an
5 OpenMP implementation supports.

6 Format

```
7                                     C / C++
8 typedef int (*ompt_enumerate_states_t) (
9     int current_state,
10    int *next_state,
11    const char **next_state_name
12 );
```

7 C / C++

12 Description

13 An OpenMP implementation may support only a subset of the states defined by the
14 `ompt_state_t` enumeration type. An OpenMP implementation may also support
15 implementation-specific states. The `ompt_enumerate_states` runtime entry point, which has
16 type signature `ompt_enumerate_states_t`, enables a tool to enumerate the supported thread
17 states.

18 When a supported thread state is passed as *current_state*, the runtime entry point assigns the next
19 thread state in the enumeration to the variable passed by reference in *next_state* and assigns the
20 name associated with that state to the character pointer passed by reference in *next_state_name*.

21 Whenever one or more states are left in the enumeration, the `ompt_enumerate_states`
22 runtime entry point returns 1. When the last state in the enumeration is passed as *current_state*,
23 `ompt_enumerate_states` returns 0, which indicates that the enumeration is complete.

24 Description of Arguments

25 The *current_state* argument must be a thread state that the OpenMP implementation supports. To
26 begin enumerating the supported states, a tool should pass `ompt_state_undefined` as
27 *current_state*. Subsequent invocations of `ompt_enumerate_states` should pass the value
28 assigned to the variable passed by reference in *next_state* to the previous call.

29 The value `ompt_state_undefined` is reserved to indicate an invalid thread state.
30 `ompt_state_undefined` is defined as an integer with the value 0.

31 The *next_state* argument is a pointer to an integer in which `ompt_enumerate_states` returns
32 the value of the next state in the enumeration.

The *next_state_name* argument is a pointer to a character string pointer through which **ompt_enumerate_states** returns a string that describes the next state.

Constraints on Arguments

Any string returned through the *next_state_name* argument must be immutable and defined for the lifetime of a program execution.

Cross References

- **ompt_state_t** type, see Section 4.4.4.26 on page 452.

4.6.1.2 **ompt_enumerate_mutex_impls_t**

Summary

The **ompt_enumerate_mutex_impls_t** type is the type signature of the **ompt_enumerate_mutex_impls** runtime entry point, which enumerates the kinds of mutual exclusion implementations that an OpenMP implementation employs.

Format

C / C++

```
typedef int (*ompt_enumerate_mutex_impls_t) (  
    int current_impl,  
    int *next_impl,  
    const char **next_impl_name  
);
```

C / C++

Description

Mutual exclusion for locks, **critical** sections, and **atomic** regions may be implemented in several ways. The **ompt_enumerate_mutex_impls** runtime entry point, which has type signature **ompt_enumerate_mutex_impls_t**, enables a tool to enumerate the supported mutual exclusion implementations.

When a supported mutex implementation is passed as *current_impl*, the runtime entry point assigns the next mutex implementation in the enumeration to the variable passed by reference in *next_impl* and assigns the name associated with that mutex implementation to the character pointer passed by reference in *next_impl_name*.

Whenever one or more mutex implementations are left in the enumeration, the **ompt_enumerate_mutex_impls** runtime entry point returns 1. When the last mutex implementation in the enumeration is passed as *current_impl*, the runtime entry point returns 0, which indicates that the enumeration is complete.

Description of Arguments

The *current_impl* argument must be a mutex implementation that an OpenMP implementation supports. To begin enumerating the supported mutex implementations, a tool should pass **ompt_mutex_impl_none** as *current_impl*. Subsequent invocations of **ompt_enumerate_mutex_impls** should pass the value assigned to the variable passed in *next_impl* to the previous call.

The value **ompt_mutex_impl_none** is reserved to indicate an invalid mutex implementation. **ompt_mutex_impl_none** is defined as an integer with the value 0.

The *next_impl* argument is a pointer to an integer in which **ompt_enumerate_mutex_impls** returns the value of the next mutex implementation in the enumeration.

The *next_impl_name* argument is a pointer to a character string pointer in which **ompt_enumerate_mutex_impls** returns a string that describes the next mutex implementation.

Constraints on Arguments

Any string returned through the *next_impl_name* argument must be immutable and defined for the lifetime of a program execution.

Cross References

- **ompt_mutex_t** type, see Section [4.4.4.16](#) on page [445](#).

4.6.1.3 **ompt_set_callback_t**

Summary

The **ompt_set_callback_t** type is the type signature of the **ompt_set_callback** runtime entry point, which registers a pointer to a tool callback that an OpenMP implementation invokes when a host OpenMP event occurs.

Format

C / C++

```
typedef ompt_set_result_t (*ompt_set_callback_t) (  
    ompt_callbacks_t event,  
    ompt_callback_t callback  
);
```

C / C++

Description

OpenMP implementations can use callbacks to indicate the occurrence of events during the execution of an OpenMP program. The **ompt_set_callback** runtime entry point, which has type signature **ompt_set_callback_t**, registers a callback for an OpenMP event on the current device. The return value of **ompt_set_callback** indicates the outcome of registering the callback.

Description of Arguments

The *event* argument indicates the event for which the callback is being registered.

The *callback* argument is a tool callback function. If *callback* is **NULL** then callbacks associated with *event* are disabled. If callbacks are successfully disabled then **ompt_set_always** is returned.

Constraints on Arguments

When a tool registers a callback for an event, the type signature for the callback must match the type signature appropriate for the event.

Restrictions

The **ompt_set_callback** runtime entry point has the following restriction:

- The entry point must not return **ompt_set_impossible**.

Cross References



- Monitoring activity on the host with OMPT, see Section 4.2.4 on page 425.
- **ompt_callbacks_t** enumeration type, see Section 4.4.2 on page 434.
- **ompt_callback_t** type, see Section 4.4.4.1 on page 438.
- **ompt_set_result_t** type, see Section 4.4.4.2 on page 438.
- **ompt_get_callback_t** host callback type signature, see Section 4.6.1.4 on page 502.

1 4.6.1.4 ompt_get_callback_t

2 Summary

3 The **ompt_get_callback_t** type is the type signature of the **ompt_get_callback** runtime
4 entry point, which retrieves a pointer to a registered tool callback routine (if any) that an OpenMP
5 implementation invokes when a host OpenMP event occurs.

6 Format

```
7 
  typedef int (*ompt_get_callback_t) (
8      ompt_callbacks_t event,
9      ompt_callback_t *callback
10 );
  
```

11 Description

12 The **ompt_get_callback** runtime entry point, which has type signature
13 **ompt_get_callback_t**, retrieves a pointer to the tool callback that an OpenMP
14 implementation may invoke when a host OpenMP event occurs. If a non-null tool callback is
15 registered for the specified event, the pointer to the tool callback is assigned to the variable passed
16 by reference in *callback* and **ompt_get_callback** returns 1; otherwise, it returns 0. If
17 **ompt_get_callback** returns 0, the value of the variable passed by reference as *callback* is
18 undefined.

19 Description of Arguments

20 The *event* argument indicates the event for which the callback would be invoked.

21 The *callback* argument returns a pointer to the callback associated with *event*.

22 Constraints on Arguments

23 The *callback* argument must be a reference to a variable of specified type.

24 Cross References

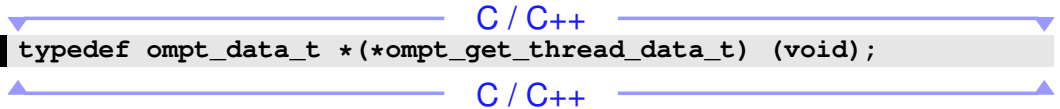
- 25 • **ompt_callbacks_t** enumeration type, see Section 4.4.2 on page 434.
- 26 • **ompt_callback_t** type, see Section 4.4.4.1 on page 438.
- 27 • **ompt_set_callback_t** type signature, see Section 4.6.1.3 on page 500.

1 4.6.1.5 `ompt_get_thread_data_t`

2 Summary

3 The `ompt_get_thread_data_t` type is the type signature of the
4 `ompt_get_thread_data` runtime entry point, which returns the address of the thread data
5 object for the current thread.

6 Format

7  `typedef ompt_data_t *(*ompt_get_thread_data_t) (void);`
C / C++

8 Binding

9 The binding thread for the `ompt_get_thread_data` runtime entry point is the current thread.

10 Description

11 Each OpenMP thread can have an associated thread data object of type `ompt_data_t`. The
12 `ompt_get_thread_data` runtime entry point, which has type signature
13 `ompt_get_thread_data_t`, retrieves a pointer to the thread data object, if any, that is
14 associated with the current thread. A tool may use a pointer to an OpenMP thread's data object that
15 `ompt_get_thread_data` retrieves to inspect or to modify the value of the data object. When
16 an OpenMP thread is created, its data object is initialized with value `ompt_data_none`.

17 This runtime entry point is *async signal safe*.

18 Cross References

- 19 • `ompt_data_t` type, see Section [4.4.4.4](#) on page [440](#).

20 4.6.1.6 `ompt_get_num_procs_t`

21 Summary

22 The `ompt_get_num_procs_t` type is the type signature of the `ompt_get_num_procs`
23 runtime entry point, which returns the number of processors currently available to the execution
24 environment on the host device.

Format

C / C++
typedef int (*ompt_get_num_procs_t) (void);
C / C++

Binding

The binding thread set for the **ompt_get_num_procs** runtime entry point is all threads on the host device.

Description

The **ompt_get_num_procs** runtime entry point, which has type signature **ompt_get_num_procs_t**, returns the number of processors that are available on the host device at the time the routine is called. This value may change between the time that it is determined and the time that it is read in the calling context due to system actions outside the control of the OpenMP implementation.

This runtime entry point is *async signal safe*.

4.6.1.7 ompt_get_num_places_t

Summary

The **ompt_get_num_places_t** type is the type signature of the **ompt_get_num_places** runtime entry point, which returns the number of places currently available to the execution environment in the place list.

Format

C / C++
typedef int (*ompt_get_num_places_t) (void);
C / C++

Binding

The binding thread set for the **ompt_get_num_places** runtime entry point is all threads on a device.

Description

The `ompt_get_num_places` runtime entry point, which has type signature `ompt_get_num_places_t`, returns the number of places in the place list. This value is equivalent to the number of places in the *place-partition-var* ICV in the execution environment of the initial task.

This runtime entry point is *async signal safe*.

Cross References

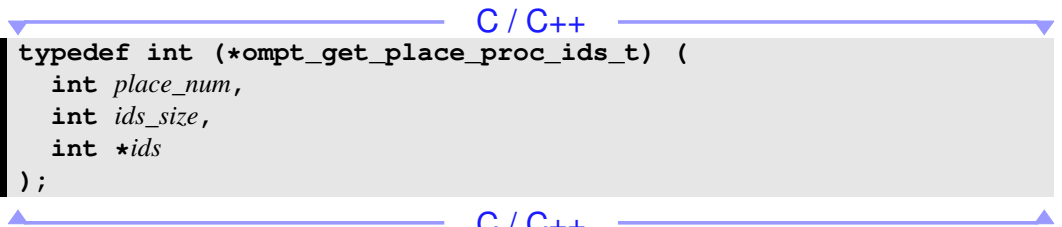
- *place-partition-var* ICV, see Section 2.5 on page 63.
- `OMP_PLACES` environment variable, see Section 6.5 on page 605.

4.6.1.8 `ompt_get_place_proc_ids_t`

Summary

The `ompt_get_place_procs_ids_t` type is the type signature of the `ompt_get_num_place_procs_ids` runtime entry point, which returns the numerical identifiers of the processors that are available to the execution environment in the specified place.

Format



```
typedef int (*ompt_get_place_proc_ids_t) (  
    int place_num,  
    int ids_size,  
    int *ids  
);
```

Binding

The binding thread set for the `ompt_get_place_proc_ids` runtime entry point is all threads on a device.

Description

The `ompt_get_place_proc_ids` runtime entry point, which has type signature `ompt_get_place_proc_ids_t`, returns the numerical identifiers of each processor that is associated with the specified place. These numerical identifiers are non-negative and their meaning is implementation defined.

Description of Arguments

The *place_num* argument specifies the place that is being queried.

The *ids* argument is an array in which the routine can return a vector of processor identifiers in the specified place.

The *ids_size* argument indicates the size of the result array that is specified by *ids*.

Effect

If the *ids* array of size *ids_size* is large enough to contain all identifiers then they are returned in *ids* and their order in the array is implementation defined. Otherwise, if the *ids* array is too small the values in *ids* when the function returns are unspecified. The routine always returns the number of numerical identifiers of the processors that are available to the execution environment in the specified place.

4.6.1.9 `ompt_get_place_num_t`

Summary

The `ompt_get_place_num_t` type is the type signature of the `ompt_get_place_num` runtime entry point, which returns the place number of the place to which the current thread is bound.

Format

C / C++

```
typedef int (*ompt_get_place_num_t) (void);
```

C / C++

Binding

The binding thread set of the `ompt_get_place_num` runtime entry point is the current thread.

Description

When the current thread is bound to a place, `ompt_get_place_num` returns the place number associated with the thread. The returned value is between 0 and one less than the value returned by `ompt_get_num_places`, inclusive. When the current thread is not bound to a place, the routine returns -1.

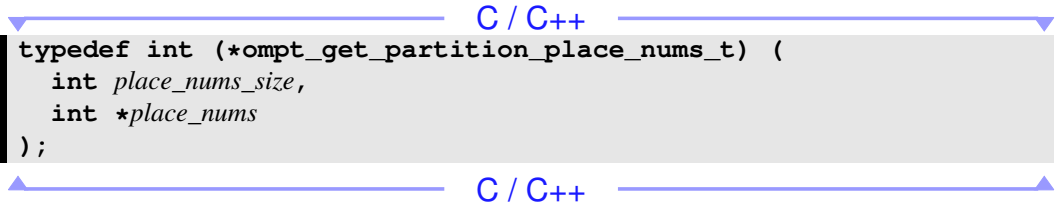
This runtime entry point is *async signal safe*.

1 4.6.1.10 `ompt_get_partition_place_nums_t`

2 Summary

3 The `ompt_get_partition_place_nums_t` type is the type signature of the
4 `ompt_get_partition_place_nums` runtime entry point, which returns a list of place
5 numbers that correspond to the places in the *place-partition-var* ICV of the innermost implicit task.

6 Format

7 The diagram shows the C/C++ signature of the `ompt_get_partition_place_nums_t` type. It consists of a typedef declaration: `typedef int (*ompt_get_partition_place_nums_t) (` on line 7, `int place_nums_size,` on line 8, `int *place_nums` on line 9, and `);` on line 10. The entire signature is enclosed in a light gray box. Above and below the box are blue double-headed arrows with the text "C / C++" in the center.

```
typedef int (*ompt_get_partition_place_nums_t) (  
    int place_nums_size,  
    int *place_nums  
);
```

11 Binding

12 The binding task set for the `ompt_get_partition_place_nums` runtime entry point is the
13 current implicit task.

14 Description

15 The `ompt_get_partition_place_nums` runtime entry point, which has type signature
16 `ompt_get_partition_place_nums_t`, returns a list of place numbers that correspond to
17 the places in the *place-partition-var* ICV of the innermost implicit task.

18 This runtime entry point is *async signal safe*.

19 Description of Arguments

20 The *place_nums* argument is an array in which the routine can return a vector of place identifiers.

21 The *place_nums_size* argument indicates the size of the result array that the *place_nums* argument
22 specifies.

23 Effect

24 If the *place_nums* array of size *place_nums_size* is large enough to contain all identifiers then they
25 are returned in *place_nums* and their order in the array is implementation defined. Otherwise, if the
26 *place_nums* array is too small, the values in *place_nums* when the function returns are unspecified.
27 The routine always returns the number of places in the *place-partition-var* ICV of the innermost
28 implicit task.

Cross References

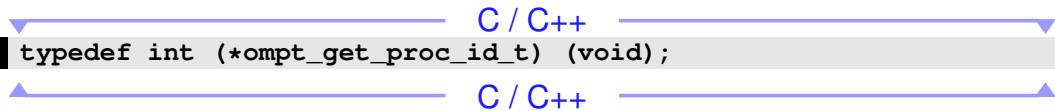
- *place-partition-var* ICV, see Section 2.5 on page 63.
- **OMP_PLACES** environment variable, see Section 6.5 on page 605.

4.6.1.11 `ompt_get_proc_id_t`

Summary

The `ompt_get_proc_id_t` type is the type signature of the `ompt_get_proc_id` runtime entry point, which returns the numerical identifier of the processor of the current thread.

Format


`typedef int (*ompt_get_proc_id_t) (void);`

Binding

The binding thread set for the `ompt_get_proc_id` runtime entry point is the current thread.

Description

The `ompt_get_proc_id` runtime entry point, which has type signature `ompt_get_proc_id_t`, returns the numerical identifier of the processor of the current thread. A defined numerical identifier is non-negative and its meaning is implementation defined. A negative number indicates a failure to retrieve the numerical identifier.

This runtime entry point is *async signal safe*.

4.6.1.12 `ompt_get_state_t`

Summary

The `ompt_get_state_t` type is the type signature of the `ompt_get_state` runtime entry point, which returns the state and the wait identifier of the current thread.

Format

```
typedef int (*ompt_get_state_t) (  
    ompt_wait_id_t *wait_id  
);
```

Binding

The binding thread for the **ompt_get_state** runtime entry point is the current thread.

Description

Each OpenMP thread has an associated state and a wait identifier. If a thread's state indicates that the thread is waiting for mutual exclusion then its wait identifier contains an opaque handle that indicates the data object upon which the thread is waiting. The **ompt_get_state** runtime entry point, which has type signature **ompt_get_state_t**, retrieves the state and wait identifier of the current thread. The returned value may be any one of the states predefined by **ompt_state_t** or a value that represents any implementation specific state. The tool may obtain a string representation for each state with the **ompt_enumerate_states** function.

If the returned state indicates that the thread is waiting for a lock, nest lock, critical section, atomic region, or ordered region then the value of the thread's wait identifier is assigned to a non-null wait identifier passed as the *wait_id* argument.

This runtime entry point is *async signal safe*.

Description of Arguments

The *wait_id* argument is a pointer to an opaque handle that is available to receive the value of the thread's wait identifier. If *wait_id* is not **NULL** then the entry point assigns the value of the thread's wait identifier to the object to which *wait_id* points. If the returned state is not one of the specified wait states then the value of opaque object to which *wait_id* points is undefined after the call.

Constraints on Arguments

The argument passed to the entry point must be a reference to a variable of the specified type or **NULL**.

Cross References

- `ompt_state_t` type, see Section 4.4.4.26 on page 452.
- `ompt_wait_id_t` type, see Section 4.4.4.29 on page 456.
- `ompt_enumerate_states_t` type, see Section 4.6.1.1 on page 498.

4.6.1.13 `ompt_get_parallel_info_t`

Summary

The `ompt_get_parallel_info_t` type is the type signature of the `ompt_get_parallel_info` runtime entry point, which returns information about the parallel region, if any, at the specified ancestor level for the current execution context.

Format

C / C++

```
typedef int (*ompt_get_parallel_info_t) (  
    int ancestor_level,  
    ompt_data_t **parallel_data,  
    int *team_size  
);
```

C / C++

Description

During execution, an OpenMP program may employ nested parallel regions. The `ompt_get_parallel_info` runtime entry point known, which has type signature `ompt_get_parallel_info_t`, retrieves information, about the current parallel region and any enclosing parallel regions for the current execution context. The entry point returns 2 if there is a parallel region at the specified ancestor level and the information is available, 1 if there is a parallel region at the specified ancestor level but the information is currently unavailable, and 0 otherwise.

A tool may use the pointer to a parallel region's data object that it obtains from this runtime entry point to inspect or to modify the value of the data object. When a parallel region is created, its data object will be initialized with the value `ompt_data_none`.

This runtime entry point is *async signal safe*.

Between a *parallel-begin* event and an *implicit-task-begin* event, a call to **ompt_get_parallel_info(0, ...)** may return information about the outer parallel team, the new parallel team or an inconsistent state.

If a thread is in the state **ompt_state_wait_barrier_implicit_parallel** then a call to **ompt_get_parallel_info** may return a pointer to a copy of the specified parallel region's *parallel_data* rather than a pointer to the data word for the region itself. This convention enables the master thread for a parallel region to free storage for the region immediately after the region ends, yet avoid having some other thread in the region's team potentially reference the region's *parallel_data* object after it has been freed.

Description of Arguments

The *ancestor_level* argument specifies the parallel region of interest by its ancestor level. Ancestor level 0 refers to the innermost parallel region; information about enclosing parallel regions may be obtained using larger values for *ancestor_level*.

The *parallel_data* argument returns the parallel data if the argument is not **NULL**.

The *team_size* argument returns the team size if the argument is not **NULL**.

Effect

If the runtime entry point returns 0 or 1, no argument is modified. Otherwise, **ompt_get_parallel_info** has the following effects:

- If a non-null value was passed for *parallel_data*, the value returned in *parallel_data* is a pointer to a data word that is associated with the parallel region at the specified level; and
- If a non-null value was passed for *team_size*, the value returned in the integer to which *team_size* point is the number of threads in the team that is associated with the parallel region.

Constraints on Arguments

While argument *ancestor_level* is passed by value, all other arguments to the entry point must be pointers to variables of the specified types or **NULL**.

Cross References

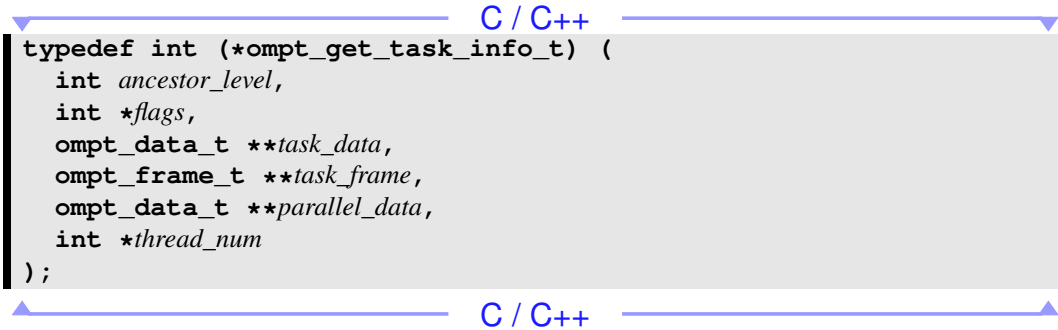
- **ompt_data_t** type, see Section 4.4.4.4 on page 440.

1 4.6.1.14 ompt_get_task_info_t

2 Summary

3 The **ompt_get_task_info_t** type is the type signature of the **ompt_get_task_info**
4 runtime entry point, which returns information about the task, if any, at the specified ancestor level
5 in the current execution context.

6 Format

7 The diagram shows the C/C++ type signature for **ompt_get_task_info_t**. It consists of a blue double-headed arrow at the top with "C / C++" in the center, followed by a gray box containing the typedef signature, and another blue double-headed arrow at the bottom with "C / C++" in the center.
8 `typedef int (*ompt_get_task_info_t) (
9 int ancestor_level,
10 int *flags,
11 ompt_data_t **task_data,
12 ompt_frame_t **task_frame,
13 ompt_data_t **parallel_data,
14 int *thread_num
15);`

15 Description

16 During execution, an OpenMP thread may be executing an OpenMP task. Additionally, the thread's
17 stack may contain procedure frames that are associated with suspended OpenMP tasks or OpenMP
18 runtime system routines. To obtain information about any task on the current thread's stack, a tool
19 uses the **ompt_get_task_info** runtime entry point, which has type signature
20 **ompt_get_task_info_t**.

21 Ancestor level 0 refers to the active task; information about other tasks with associated frames
22 present on the stack in the current execution context may be queried at higher ancestor levels.

23 The **ompt_get_task_info** runtime entry point returns 2 if there is a task region at the
24 specified ancestor level and the information is available, 1 if there is a task region at the specified
25 ancestor level but the information is currently unavailable, and 0 otherwise.

26 If a task exists at the specified ancestor level and the information is available then information is
27 returned in the variables passed by reference to the entry point. If no task region exists at the
28 specified ancestor level or the information is unavailable then the values of variables passed by
29 reference to the entry point are undefined when **ompt_get_task_info** returns.

30 A tool may use a pointer to a data object for a task or parallel region that it obtains from
31 **ompt_get_task_info** to inspect or to modify the value of the data object. When either a
32 parallel region or a task region is created, its data object will be initialized with the value
33 **ompt_data_none**.

34 This runtime entry point is *async signal safe*.

Description of Arguments

The *ancestor_level* argument specifies the task region of interest by its ancestor level. Ancestor level 0 refers to the active task; information about ancestor tasks found in the current execution context may be queried at higher ancestor levels.

The *flags* argument returns the task type if the argument is not **NULL**.

The *task_data* argument returns the task data if the argument is not **NULL**.

The *task_frame* argument returns the task frame pointer if the argument is not **NULL**.

The *parallel_data* argument returns the parallel data if the argument is not **NULL**.

The *thread_num* argument returns the thread number if the argument is not **NULL**.

Effect

If the runtime entry point returns 0 or 1, no argument is modified. Otherwise, **ompt_get_task_info** has the following effects:

- If a non-null value was passed for *flags* then the value returned in the integer to which *flags* points represents the type of the task at the specified level; possible task types include initial, implicit, explicit, and target tasks;
- If a non-null value was passed for *task_data* then the value that is returned in the object to which it points is a pointer to a data word that is associated with the task at the specified level;
- If a non-null value was passed for *task_frame* then the value that is returned in the object to which *task_frame* points is a pointer to the **ompt_frame_t** structure that is associated with the task at the specified level;
- If a non-null value was passed for *parallel_data* then the value that is returned in the object to which *parallel_data* points is a pointer to a data word that is associated with the parallel region that contains the task at the specified level or, if the task at the specified level is an initial task, **NULL**; and
- If a non-null value was passed for *thread_num* then the value that is returned in the object to which *thread_num* points indicates the number of the thread in the parallel region that is executing the task at the specified level.

Constraints on Arguments

While argument *ancestor_level* is passed by value, all other arguments to **ompt_get_task_info** must be pointers to variables of the specified types or **NULL**.

Cross References

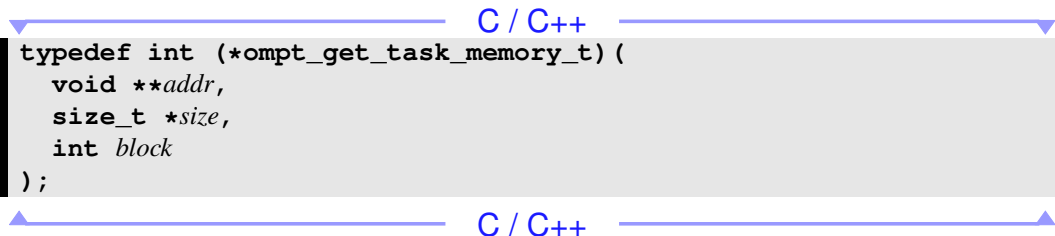
- `ompt_data_t` type, see Section 4.4.4.4 on page 440.
- `ompt_task_flag_t` type, see Section 4.4.4.18 on page 446.
- `ompt_frame_t` type, see Section 4.4.4.27 on page 454.

4.6.1.15 `ompt_get_task_memory_t`

Summary

The `ompt_get_task_memory_t` type is the type signature of the `ompt_get_task_memory` runtime entry point, which returns information about memory ranges that are associated with the task.

Format



```
typedef int (*ompt_get_task_memory_t) (  
    void **addr,  
    size_t *size,  
    int block  
);
```

Description

During execution, an OpenMP thread may be executing an OpenMP task. The OpenMP implementation must preserve the data environment from the creation of the task for the execution of the task. The `ompt_get_task_memory` runtime entry point, which has type signature `ompt_get_task_memory_t`, provides information about the memory ranges used to store the data environment for the current task.

Multiple memory ranges may be used to store these data. The *block* argument supports iteration over these memory ranges.

The `ompt_get_task_memory` runtime entry point returns 1 if there are more memory ranges available, and 0 otherwise. If no memory is used for a task, *size* is set to 0. In this case, *addr* is unspecified.

This runtime entry point is *async signal safe*.

Description of Arguments

The *addr* argument is a pointer to a void pointer return value to provide the start address of a memory block.

The *size* argument is a pointer to a size type return value to provide the size of the memory block.

The *block* argument is an integer value to specify the memory block of interest.

4.6.1.16 ompt_get_target_info_t

Summary

The `ompt_get_target_info_t` type is the type signature of the `ompt_get_target_info` runtime entry point, which returns identifiers that specify a thread's current **target** region and target operation ID, if any.

Format

```
C / C++
typedef int (*ompt_get_target_info_t) (
    uint64_t *device_num,
    ompt_id_t *target_id,
    ompt_id_t *host_op_id
);
```

Description

The `ompt_get_target_info` entry point, which has type signature `ompt_get_target_info_t`, returns 1 if the current thread is in a **target** region and 0 otherwise. If the entry point returns 0 then the values of the variables passed by reference as its arguments are undefined.

If the current thread is in a **target** region then `ompt_get_target_info` returns information about the current device, active **target** region, and active host operation, if any.

This runtime entry point is *async signal safe*.

Description of Arguments

The `device_num` argument returns the device number if the current thread is in a **target** region.

The `target_id` argument returns the **target** region identifier if the current thread is in a **target** region.

If the current thread is in the process of initiating an operation on a target device (for example, copying data to or from an accelerator or launching a kernel) then `host_op_id` returns the identifier for the operation; otherwise, `host_op_id` returns **ompt_id_none**.

Constraints on Arguments

Arguments passed to the entry point must be valid references to variables of the specified types.

Cross References

- `ompt_id_t` type, see Section 4.4.4.3 on page 439.

4.6.1.17 `ompt_get_num_devices_t`

Summary

The `ompt_get_num_devices_t` type is the type signature of the `ompt_get_num_devices` runtime entry point, which returns the number of available devices.

Format

C / C++

```
typedef int (*ompt_get_num_devices_t) (void);
```

C / C++

Description

The `ompt_get_num_devices` runtime entry point, which has type signature `ompt_get_num_devices_t`, returns the number of devices available to an OpenMP program.



This runtime entry point is *async signal safe*.

1 4.6.1.18 ompt_get_unique_id_t

2 Summary

3 The **ompt_get_unique_id_t** type is the type signature of the **ompt_get_unique_id**
4 runtime entry point, which returns a unique number.

5 Format

6  **typedef uint64_t (*ompt_get_unique_id_t) (void);**
7 

7 Description

8 The **ompt_get_unique_id** runtime entry point, which has type signature
9 **ompt_get_unique_id_t**, returns a number that is unique for the duration of an OpenMP
10 program. Successive invocations may not result in consecutive or even increasing numbers.



11 This runtime entry point is *async signal safe*.

12 4.6.1.19 ompt_finalize_tool_t

13 Summary

14 The **ompt_finalize_tool_t** type is the type signature of the **ompt_finalize_tool**
15 runtime entry point, which enables a tool to finalize itself.

16 Format

17  **typedef void (*ompt_finalize_tool_t) (void);**
18 

18 Description

19 A tool may detect that the execution of an OpenMP program is ending before the OpenMP
20 implementation does. To facilitate clean termination of the tool, the tool may invoke the
21 **ompt_finalize_tool** runtime entry point, which has type signature
22 **ompt_finalize_tool_t**. Upon completion of **ompt_finalize_tool**, no OMPT
23 callbacks are dispatched.

Effect

The `ompt_finalize_tool` routine detaches the tool from the runtime, unregisters all callbacks and invalidates all OMPT entry points passed to the tool in the *lookup-function*. Upon completion of `ompt_finalize_tool`, no further callbacks will be issued on any thread.

Before the callbacks are unregistered, the OpenMP runtime should attempt to dispatch all outstanding registered callbacks as well as the callbacks that would be encountered during shutdown of the runtime, if possible in the current execution context.

4.6.2 Entry Points in the OMPT Device Tracing Interface

The runtime entry points with type signatures of the types that are specified in this section enable a tool to trace activities on a device.

4.6.2.1 `ompt_get_device_num_procs_t`

Summary

The `ompt_get_device_num_procs_t` type is the type signature of the `ompt_get_device_num_procs` runtime entry point, which returns the number of processors currently available to the execution environment on the specified device.

Format

```
typedef int (*ompt_get_device_num_procs_t) (  
    ompt_device_t *device  
);
```

Description

The `ompt_get_device_num_procs` runtime entry point, which has type signature `ompt_get_device_num_procs_t`, returns the number of processors that are available on the device at the time the routine is called. This value may change between the time that it is determined and the time that it is read in the calling context due to system actions outside the control of the OpenMP implementation.

Description of Arguments

The *device* argument is a pointer to an opaque object that represents the target device instance. The pointer to the device instance object is used by functions in the device tracing interface to identify the device being addressed.

Cross References

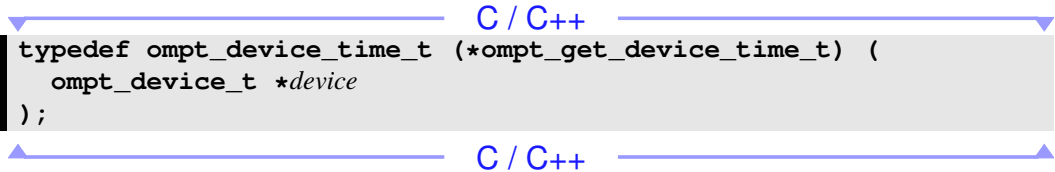
- `ompt_device_t` type, see Section 4.4.4.5 on page 441.

4.6.2.2 `ompt_get_device_time_t`

Summary

The `ompt_get_device_time_t` type is the type signature of the `ompt_get_device_time` runtime entry point, which returns the current time on the specified device.

Format



```
typedef ompt_device_time_t (*ompt_get_device_time_t) (  
    ompt_device_t *device  
);
```

Description

Host and target devices are typically distinct and run independently. If host and target devices are different hardware components, they may use different clock generators. For this reason, a common time base for ordering host-side and device-side events may not be available.

The `ompt_get_device_time` runtime entry point, which has type signature `ompt_get_device_time_t`, returns the current time on the specified device. A tool can use this information to align time stamps from different devices.

Description of Arguments

The *device* argument is a pointer to an opaque object that represents the target device instance. The pointer to the device instance object is used by functions in the device tracing interface to identify the device being addressed.

Cross References

- `ompt_device_t` type, see Section 4.4.4.5 on page 441.
- `ompt_device_time_t` type, see Section 4.4.4.6 on page 441.

4.6.2.3 `ompt_translate_time_t`

Summary

The `ompt_translate_time_t` type is the type signature of the `ompt_translate_time` runtime entry point, which translates a time value that is obtained from the specified device to a corresponding time value on the host device.

Format

C / C++

```
typedef double (*ompt_translate_time_t) (  
    ompt_device_t *device,  
    ompt_device_time_t time  
);
```

C / C++

Description

The `ompt_translate_time` runtime entry point, which has type signature `ompt_translate_time_t`, translates a time value obtained from the specified device to a corresponding time value on the host device. The returned value for the host time has the same meaning as the value returned from `omp_get_wtime`.

Note – The accuracy of time translations may degrade if they are not performed promptly after a device time value is received and if either the host or device vary their clock speeds. Prompt translation of device times to host times is recommended.

Description of Arguments

The *device* argument is a pointer to an opaque object that represents the target device instance. The pointer to the device instance object is used by functions in the device tracing interface to identify the device being addressed.

The *time* argument is a time from the specified device.

Cross References

- `omp_get_wtime` routine, see Section 3.4.1 on page 394.
- `ompt_device_t` type, see Section 4.4.4.5 on page 441.
- `ompt_device_time_t` type, see Section 4.4.4.6 on page 441.

4.6.2.4 `ompt_set_trace_ompt_t`

Summary

The `ompt_set_trace_ompt_t` type is the type signature of the `ompt_set_trace_ompt` runtime entry point, which enables or disables the recording of trace records for one or more types of OMPT events.

Format

```
C / C++
typedef ompt_set_result_t (*ompt_set_trace_ompt_t) (
    ompt_device_t *device,
    unsigned int enable,
    unsigned int etype
);
```

C / C++

Description of Arguments

The *device* argument points to an opaque object that represents the target device instance. Functions in the device tracing interface use this pointer to identify the device that is being addressed.

The *etype* argument indicates the events to which the invocation of `ompt_set_trace_ompt` applies. If the value of *etype* is 0 then the invocation applies to all events. If *etype* is positive then it applies to the event in `ompt_callbacks_t` that matches that value.

The *enable* argument indicates whether tracing should be enabled or disabled for the event or events that the *etype* argument specifies. A positive value for *enable* indicates that recording should be enabled; a value of 0 for *enable* indicates that recording should be disabled.

Restrictions

The `ompt_set_trace_ompt` runtime entry point has the following restriction:

- The entry point must not return `ompt_set_sometimes_paired`.

Cross References

- Tracing activity on target devices with OMPT, see Section 4.2.5 on page 427.
- `ompt_callbacks_t` type, see Section 4.4.2 on page 434.
- `ompt_set_result_t` type, see Section 4.4.4.2 on page 438.
- `ompt_device_t` type, see Section 4.4.4.5 on page 441.

4.6.2.5 `ompt_set_trace_native_t`

Summary

The `ompt_set_trace_native_t` type is the type signature of the `ompt_set_trace_native` runtime entry point, which enables or disables the recording of native trace records for a device.

Format

```
C / C++
typedef ompt_set_result_t (*ompt_set_trace_native_t) (
    ompt_device_t *device,
    int enable,
    int flags
);
```

Description

This interface is designed for use by a tool that cannot directly use native control functions for the device. If a tool can directly use the native control functions then it can invoke native control functions directly using pointers that the *lookup* function associated with the device provides and that are described in the *documentation* string that is provided to the device initializer callback.

Description of Arguments

The *device* argument points to an opaque object that represents the target device instance. Functions in the device tracing interface use this pointer to identify the device that is being addressed.

The *enable* argument indicates whether this invocation should enable or disable recording of events.

The *flags* argument specifies the kinds of native device monitoring to enable or to disable. Each kind of monitoring is specified by a flag bit. Flags can be composed by using logical `or` to combine enumeration values from type `ompt_native_mon_flag_t`.

To start, to pause, to flush, or to stop tracing for a specific target device associated with *device*, a tool invokes the `ompt_start_trace`, `ompt_pause_trace`, `ompt_flush_trace`, or `ompt_stop_trace` runtime entry point for the device.

Restrictions

The `ompt_set_trace_native` runtime entry point has the following restriction:

- The entry point must not return `ompt_set_sometimes_paired`.

Cross References

- Tracing activity on target devices with OMPT, see Section 4.2.5 on page 427.
- `ompt_set_result_t` type, see Section 4.4.4.2 on page 438.
- `ompt_device_t` type, see Section 4.4.4.5 on page 441.

4.6.2.6 `ompt_start_trace_t`

Summary

The `ompt_start_trace_t` type is the type signature of the `ompt_start_trace` runtime entry point, which starts tracing of activity on a specific device.

Format

```
C / C++
typedef int (*ompt_start_trace_t) (
    ompt_device_t *device,
    ompt_callback_buffer_request_t request,
    ompt_callback_buffer_complete_t complete
);
```

Description

A device's **ompt_start_trace** runtime entry point, which has type signature **ompt_start_trace_t**, initiates tracing on the device. Under normal operating conditions, every event buffer provided to a device by a tool callback is returned to the tool before the OpenMP runtime shuts down. If an exceptional condition terminates execution of an OpenMP program, the OpenMP runtime may not return buffers provided to the device.

An invocation of **ompt_start_trace** returns 1 if the command succeeds and 0 otherwise.

Description of Arguments

The *device* argument points to an opaque object that represents the target device instance. Functions in the device tracing interface use this pointer to identify the device that is being addressed.

The *request* argument specifies a tool callback that supplies a device with a buffer to deposit events.

The *complete* argument specifies a tool callback that is invoked by the OpenMP implementation to empty a buffer that contains event records.

Cross References

- **ompt_device_t** type, see Section 4.4.4.5 on page 441.
- **ompt_callback_buffer_request_t** callback type, see Section 4.5.2.23 on page 486.
- **ompt_callback_buffer_complete_t** callback type, see Section 4.5.2.24 on page 487.

4.6.2.7 ompt_pause_trace_t

Summary

The **ompt_pause_trace_t** type is the type signature of the **ompt_pause_trace** runtime entry point, which pauses or restarts activity tracing on a specific device.

Format

```
C / C++
typedef int (*ompt_pause_trace_t) (
    ompt_device_t *device,
    int begin_pause
);
```

C / C++

Description

A device's **ompt_pause_trace** runtime entry point, which has type signature **ompt_pause_trace_t**, pauses or resumes tracing on a device. An invocation of **ompt_pause_trace** returns 1 if the command succeeds and 0 otherwise. Redundant pause or resume commands are idempotent and will return the same value as the prior command.

Description of Arguments

The *device* argument points to an opaque object that represents the target device instance. Functions in the device tracing interface use this pointer to identify the device that is being addressed.

The *begin_pause* argument indicates whether to pause or to resume tracing. To resume tracing, zero should be supplied for *begin_pause*; To pause tracing, any other value should be supplied.

Cross References

- **ompt_device_t** type, see Section 4.4.4.5 on page 441.

4.6.2.8 ompt_flush_trace_t

Summary

The **ompt_flush_trace_t** type is the type signature of the **ompt_flush_trace** runtime entry point, which causes all pending trace records for the specified device to be delivered.

Format

```
typedef int (*ompt_flush_trace_t) (  
    ompt_device_t *device  
);
```

Description

A device's **ompt_flush_trace** runtime entry point, which has type signature **ompt_flush_trace_t**, causes the OpenMP implementation to issue a sequence of zero or more buffer completion callbacks to deliver all trace records that have been collected prior to the flush. An invocation of **ompt_flush_trace** returns 1 if the command succeeds and 0 otherwise.

Description of Arguments

The *device* argument points to an opaque object that represents the target device instance. Functions in the device tracing interface use this pointer to identify the device that is being addressed.

Cross References

- `ompt_device_t` type, see Section 4.4.4.5 on page 441.

4.6.2.9 `ompt_stop_trace_t`

Summary

The `ompt_stop_trace_t` type is the type signature of the `ompt_stop_trace` runtime entry point, which stops tracing for a device.

Format

C / C++

```
typedef int (*ompt_stop_trace_t) (  
    ompt_device_t *device  
);
```

C / C++

Description

A device's `ompt_stop_trace` runtime entry point, which has type signature `ompt_stop_trace_t`, halts tracing on the device and requests that any pending trace records are flushed. An invocation of `ompt_stop_trace` returns 1 if the command succeeds and 0 otherwise.

Description of Arguments

The *device* argument points to an opaque object that represents the target device instance. Functions in the device tracing interface use this pointer to identify the device that is being addressed.

Cross References

- `ompt_device_t` type, see Section 4.4.4.5 on page 441.

1 4.6.2.10 `ompt_advance_buffer_cursor_t`

2 Summary

3 The `ompt_advance_buffer_cursor_t` type is the type signature of the
4 `ompt_advance_buffer_cursor` runtime entry point, which advances a trace buffer cursor to
5 the next record.

6 Format

```
7                                     C / C++
8 typedef int (*ompt_advance_buffer_cursor_t) (
9     ompt_device_t *device,
10    ompt_buffer_t *buffer,
11    size_t size,
12    ompt_buffer_cursor_t current,
13    ompt_buffer_cursor_t *next
14 );
```

14 Description

15 A device's `ompt_advance_buffer_cursor` runtime entry point, which has type signature
16 `ompt_advance_buffer_cursor_t`, advances a trace buffer pointer to the next trace record.
17 An invocation of `ompt_advance_buffer_cursor` returns *true* if the advance is successful
18 and the next position in the buffer is valid.

19 Description of Arguments

20 The *device* argument points to an opaque object that represents the target device instance. Functions
21 in the device tracing interface use this pointer to identify the device that is being addressed.

22 The *buffer* argument indicates a trace buffer that is associated with the cursors.

23 The argument *size* indicates the size of *buffer* in bytes.

24 The *current* argument is an opaque buffer cursor.

25 The *next* argument returns the next value of an opaque buffer cursor.

26 Cross References



- 27 • `ompt_device_t` type, see Section [4.4.4.5](#) on page [441](#).
- 28 • `ompt_buffer_cursor_t` type, see Section [4.4.4.8](#) on page [442](#).

1 4.6.2.11 `ompt_get_record_type_t`

2 Summary

3 The `ompt_get_record_type_t` type is the type signature of the
4 `ompt_get_record_type` runtime entry point, which inspects the type of a trace record.

5 Format

```
6 
7 typedef ompt_record_t (*ompt_get_record_type_t) (
8     ompt_buffer_t *buffer,
9     ompt_buffer_cursor_t current
10 );
11 
```

10 Description

11 Trace records for a device may be in one of two forms: *native* record format, which may be
12 device-specific, or *OMPT* record format, in which each trace record corresponds to an OpenMP
13 *event* and most fields in the record structure are the arguments that would be passed to the OMPT
14 callback for the event.

15 A device's `ompt_get_record_type` runtime entry point, which has type signature
16 `ompt_get_record_type_t`, inspects the type of a trace record and indicates whether the
17 record at the current position in the trace buffer is an OMPT record, a native record, or an invalid
18 record. An invalid record type is returned if the cursor is out of bounds.

19 Description of Arguments

20 The *buffer* argument indicates a trace buffer.

21 The *current* argument is an opaque buffer cursor.

22 Cross References

- 23 • `ompt_record_t` type, see Section [4.4.3.1](#) on page [435](#).
- 24 • `ompt_buffer_t` type, see Section [4.4.4.7](#) on page [441](#).
- 25 • `ompt_buffer_cursor_t` type, see Section [4.4.4.8](#) on page [442](#).

4.6.2.12 `ompt_get_record_ompt_t`

Summary

The `ompt_get_record_ompt_t` type is the type signature of the `ompt_get_record_ompt` runtime entry point, which obtains a pointer to an OMPT trace record from a trace buffer associated with a device.

Format

```
C / C++
typedef ompt_record_ompt_t *(*ompt_get_record_ompt_t) (
    ompt_buffer_t *buffer,
    ompt_buffer_cursor_t current
);
C / C++
```

Description

A device's `ompt_get_record_ompt` runtime entry point, which has type signature `ompt_get_record_ompt_t`, returns a pointer that may point to a record in the trace buffer, or it may point to a record in thread local storage in which the information extracted from a record was assembled. The information available for an event depends upon its type.

The return value of the `ompt_record_ompt_t` type includes a field of a union type that can represent information for any OMPT event record type. Another call to the runtime entry point may overwrite the contents of the fields in a record returned by a prior invocation.

Description of Arguments

The *buffer* argument indicates a trace buffer.

The *current* argument is an opaque buffer cursor.

Cross References



- `ompt_record_ompt_t` type, see Section 4.4.3.4 on page 436.
- `ompt_device_t` type, see Section 4.4.4.5 on page 441.
- `ompt_buffer_cursor_t` type, see Section 4.4.4.8 on page 442.

1 4.6.2.13 `ompt_get_record_native_t`

2 Summary

3 The `ompt_get_record_native_t` type is the type signature of the
4 `ompt_get_record_native` runtime entry point, which obtains a pointer to a native trace
5 record from a trace buffer associated with a device.

6 Format

7  `typedef void *(*ompt_get_record_native_t) (`
8 `ompt_buffer_t *buffer,`
9 `ompt_buffer_cursor_t current,`
10 `ompt_id_t *host_op_id`
11 `);`
12 

12 Description

13 A device's `ompt_get_record_native` runtime entry point, which has type signature
14 `ompt_get_record_native_t`, returns a pointer that may point into the specified
15 trace buffer, or into thread local storage in which the information extracted from a trace record was
16 assembled. The information available for a native event depends upon its type. If the function
17 returns a non-null result, it will also set the object to which `host_op_id` points to a host-side
18 identifier for the operation that is associated with the record. A subsequent call to
19 `ompt_get_record_native` may overwrite the contents of the fields in a record returned by a
20 prior invocation.

21 Description of Arguments

22 The *buffer* argument indicates a trace buffer.

23 The *current* argument is an opaque buffer cursor.

24 The *host_op_id* argument is a pointer to an identifier that is returned by the function. The entry
25 point sets the identifier to which *host_op_id* points to the value of a host-side identifier for an
26 operation on a target device that was created when the operation was initiated by the host.

27 Cross References

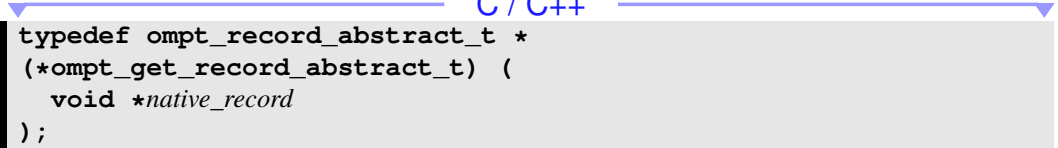

- 28 • `ompt_id_t` type, see Section [4.4.4.3](#) on page [439](#).
- 29 • `ompt_buffer_t` type, see Section [4.4.4.7](#) on page [441](#).
- 30 • `ompt_buffer_cursor_t` type, see Section [4.4.4.8](#) on page [442](#).

1 4.6.2.14 `ompt_get_record_abstract_t`

2 Summary

3 The `ompt_get_record_abstract_t` type is the type signature of the
4 `ompt_get_record_abstract` runtime entry point, which summarizes the context of a native
5 (device-specific) trace record.

6 Format

```
7 
8 typedef ompt_record_abstract_t *  
9 (*ompt_get_record_abstract_t) (  
10     void *native_record  
);  

```

11 Description

12 An OpenMP implementation may execute on a device that logs trace records in a native
13 (device-specific) format that a tool cannot interpret directly. A device's
14 `ompt_get_record_abstract` runtime entry point, which has type signature
15 `ompt_get_record_abstract_t`, translates a native trace record into a standard form.

16 Description of Arguments

17 The *native_record* argument is a pointer to a native trace record.

18 Cross References

- 19 • `ompt_record_abstract_t` type, see Section [4.4.3.3](#) on page [436](#).

20 4.6.3 Lookup Entry Points: `ompt_function_lookup_t`

21 Summary

22 The `ompt_function_lookup_t` type is the type signature of the lookup runtime entry points
23 that provide pointers to runtime entry points that are part of the OMPT interface.

Format

C / C++

```
typedef void (*ompt_interface_fn_t) (void);

typedef ompt_interface_fn_t (*ompt_function_lookup_t) (
    const char *interface_function_name
);
```

C / C++

Description

An OpenMP implementation provides a pointer to a lookup routine that provides pointers to OMPT runtime entry points. When the implementation invokes a tool initializer to configure the OMPT callback interface, it provides a lookup function that provides pointers to runtime entry points that implement routines that are part of the OMPT callback interface. Alternatively, when it invokes a tool initializer to configure the OMPT tracing interface for a device, it provides a lookup function that provides pointers to runtime entry points that implement tracing control routines appropriate for that device.

Description of Arguments

The *interface_function_name* argument is a C string that represents the name of a runtime entry point.

Cross References

- Tool initializer for a device's OMPT tracing interface, see Section 4.2.5 on page 427.
- Tool initializer for the OMPT callback interface, see Section 4.5.1.1 on page 457.
- Entry points in the OMPT callback interface, see Table 4.1 on page 426 for a list and Section 4.6.1 on page 497 for detailed definitions.
- Entry points in the OMPT tracing interface, see Table 4.3 on page 430 for a list and Section 4.6.2 on page 518 for detailed definitions.

CHAPTER 5

OMPDP Interface

This chapter describes OMPDP, which is an interface for *third-party* tools. *Third-party* tools exist in separate processes from the OpenMP program. To provide OMPDP support, an OpenMP implementation must provide an OMPDP library to be loaded by the *third-party* tool. An OpenMP implementation does not need to maintain any extra information to support OMPDP inquiries from *third-party* tools *unless* it is explicitly instructed to do so.

OMPDP allows *third-party tools* such as a debuggers to inspect the OpenMP state of a live program or core file in an implementation-agnostic manner. That is, a tool that uses OMPDP should work with any conforming OpenMP implementation. An OpenMP implementor provides a library for OMPDP that a third-party tool can dynamically load. Using the interface exported by the OMPDP library, the external tool can inspect the OpenMP state of a program. In order to satisfy requests from the third-party tool, the OMPDP library may need to read data from, or to find the addresses of symbols in the OpenMP program. The OMPDP library provides this functionality through a callback interface that the third-party tool must instantiate for the OMPDP library.

To use OMPDP, the third-party tool loads the OMPDP library. The OMPDP library exports the API that is defined throughout this section and that the tool uses to determine OpenMP information about the OpenMP program. The OMPDP library must look up the symbols and read data out of the program. It does not perform these operations directly, but instead it uses the callback interface that the tool exports to cause the tool to perform them.

The OMPDP architecture insulates tools from the internal structure of the OpenMP runtime while the OMPDP library is insulated from the details of how to access the OpenMP program. This decoupled design allows for flexibility in how the OpenMP program and tool are deployed, so that, for example, the tool and the OpenMP program are not required to execute on the same machine.

Generally the tool does not interact directly with the OpenMP runtime and, instead, interacts with it through the OMPDP library. However, a few cases require the tool to access the OpenMP runtime directly. These cases fall into two broad categories. The first is during initialization, where the tool must look up symbols and read variables in the OpenMP runtime in order to identify the OMPDP library that it should use, which is discussed in Section 5.2.2 on page 535 and Section 5.2.3 on page 536. The second category relates to arranging for the tool to be notified when certain events

occur during the execution of the OpenMP program. For this purpose, the OpenMP implementation must define certain symbols in the runtime code, as is discussed in Section 5.6 on page 594. Each of these symbols corresponds to an event type. The runtime must ensure that control passes through the appropriate named location when events occur. If the tool requires notification of an event, it can plant a breakpoint at the matching location. The location can, but may not, be a function. It can, for example, simply be a label. However, the names of the locations must have external C linkage.

5.1 OMPD Interfaces Definitions

C / C++

A compliant implementation must supply a set of definitions for the OMPD runtime entry points, OMPD tool callback signatures, OMPD tool interface routines, and the special data types of their parameters and return values. These definitions, which are listed throughout this chapter, and their associated declarations shall be provided in a header file named **omp-tools.h**. In addition, the set of definitions may specify other implementation-specific values.

The **ompd_dll_locations** function, all OMPD tool interface functions, and all OMPD runtime entry points are external functions with C linkage.

C / C++

5.2 Activating an OMPD Tool

The tool and the OpenMP program exist as separate processes. Thus, coordination is required between the OpenMP runtime and the external tool for OMPD.

5.2.1 Enabling the Runtime for OMPD

In order to support third-party tools, the OpenMP runtime may need to collect and to maintain information that it might not otherwise. The OpenMP runtime collects whatever information is necessary to support OMPD if the environment variable **OMP_DEBUG** is set to *enabled*.

Cross References

- Activating an OMPT Tool, Section 4.2 on page 420
- `OMP_DEBUG`, Section 6.20 on page 617

5.2.2 `ompd_dll_locations`

Summary

The `ompd_dll_locations` global variable indicates the location of OMPD libraries that are compatible with the OpenMP implementation.

Format

```
const char **ompd_dll_locations;
```

Description

An OpenMP runtime may have more than one OMPD library. The tool must be able to locate the right library to use for the OpenMP program that it is examining. The OpenMP runtime system must provide a public variable `ompd_dll_locations`, which is an `argv`-style vector of filename string pointers that provides the name(s) of any compatible OMPD library. This variable must have `C` linkage. The tool uses the name of the variable verbatim and, in particular, does not apply any name mangling before performing the look up.

The programming model or architecture of the tool and, thus, that of OMPD does not have to match that of the OpenMP program that is being examined. The tool must interpret the contents of `ompd_dll_locations` to find a suitable OMPD that matches its own architectural characteristics. On platforms that support different programming models (for example, 32-bit vs 64-bit), OpenMP implementations are encouraged to provide OMPD libraries for all models, and that can handle OpenMP programs of any model. Thus, for example, a 32-bit debugger that uses OMPD should be able to debug a 64-bit OpenMP program by loading a 32-bit OMPD implementation that can manage a 64-bit OpenMP runtime.

`ompd_dll_locations` points to a NULL-terminated vector of zero or more NULL-terminated pathname strings that do not have any filename conventions. This vector must be fully initialized *before* `ompd_dll_locations` is set to a non-null value, such that if a tool, such as a debugger, stops execution of the OpenMP program at any point at which `ompd_dll_locations` is non-null, then the vector of strings to which it points is valid and complete.

Cross References

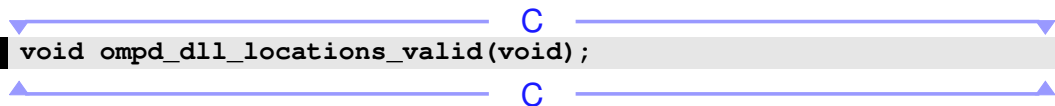
- `ompd_dll_locations_valid`, see Section 5.2.3 on page 536

5.2.3 `ompd_dll_locations_valid`

Summary

The OpenMP runtime notifies third-party tools that `ompd_dll_locations` is valid by allowing execution to pass through a location that the symbol `ompd_dll_locations_valid` identifies.

Format



A diagram showing the C language signature for the function `ompd_dll_locations_valid`. It consists of a horizontal line with a downward-pointing triangle on the left and an upward-pointing triangle on the right. The letter 'C' is centered above the line. Below the line, the signature `void ompd_dll_locations_valid(void);` is written in a light gray box.

```
void ompd_dll_locations_valid(void);
```

Description

Since `ompd_dll_locations` may not be a static variable, it may require runtime initialization. The OpenMP runtime notifies third-party tools that `ompd_dll_locations` is valid by having execution pass through a location that the symbol `ompd_dll_locations_valid` identifies. If `ompd_dll_locations` is NULL, a third-party tool can place a breakpoint at `ompd_dll_locations_valid` to be notified that `ompd_dll_locations` is initialized. In practice, the symbol `ompd_dll_locations_valid` may not be a function; instead, it may be a labeled machine instruction through which execution passes once the vector is valid.

5.3 OMPD Data Types

This section defines the OMPD types.

5.3.1 Size Type

Summary

The `ompd_size_t` type specifies the number of bytes in opaque data objects that are passed across the OMPD API.

Format

C / C++

```
typedef uint64_t ompd_size_t;
```

C / C++

5.3.2 Wait ID Type

Summary

This `ompd_wait_id_t` type identifies the object on which a thread.

Format

C / C++

```
typedef uint64_t ompd_wait_id_t;
```

C / C++

5.3.3 Basic Value Types

Summary

These definitions represent a word, address, and segment value types.

Format

C / C++

```
typedef uint64_t ompd_addr_t;  
typedef int64_t  ompd_word_t;  
typedef uint64_t ompd_seg_t;
```

C / C++

Description


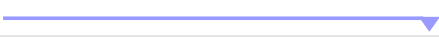


The `ompd_addr_t` type represents an unsigned integer address in an OpenMP process. The `ompd_word_t` type represents a signed version of `ompd_addr_t` to hold a signed integer of the OpenMP process. The `ompd_seg_t` type represents an unsigned integer segment value.

1 5.3.4 Address Type

2 Summary

3 The `ompd_address_t` type is used to specify device addresses.

4 Format

5  
6 `typedef struct ompd_address_t {`
7 `ompd_seg_t segment;`
8 `ompd_addr_t address;`
9 `} ompd_address_t;`
10  

9 Description





10 The `ompd_address_t` type is a structure that OMPD uses to specify device addresses, which
11 may or may not be segmented. For non-segmented architectures, `ompd_segment_none` is used
12 in the `segment` field of `ompd_address_t`; it is an instance of the `ompd_seg_t` type that has the
13 value 0.

14 5.3.5 Frame Information Type

15 Summary

16 The `ompd_frame_info_t` type is used to specify frame information.

17 Format

18  
19 `typedef struct ompd_frame_info_t {`
20 `ompd_address_t frame_address;`
21 `ompd_word_t frame_flag;`
22 `} ompd_frame_info_t;`
23  

Description

The `ompd_frame_info_t` type is a structure that OMPD uses to specify frame information. The `frame_address` field of `ompd_frame_info_t` identifies a frame. The `frame_flag` field of `ompd_frame_info_t` indicates what type of information is provided in `frame_address`. The values and meaning is the same as defined for the `ompt_frame_t` enumeration type.

Cross References

- `ompt_frame_t`, see Section 4.4.4.27 on page 454

5.3.6 System Device Identifiers

Summary

The `ompd_device_t` type provides information about OpenMP devices.

Format

C / C++

```
typedef uint64_t ompd_device_t;
```

C / C++

Description

Different OpenMP runtimes may utilize different underlying devices. The Device identifiers can vary in size and format and, thus, are not explicitly represented in OMPD. Instead, device identifiers are passed across the interface via the `ompd_device_t` type, which is a pointer to where the device identifier is stored, and the size of the device identifier in bytes. The OMPD library and a tool that uses it must agree on the format of the object that is passed. Each different kind of device identifier uses a unique unsigned 64-bit integer value.

Recommended values of `ompd_device_t` are defined in the `ompd-types.h` header file, which is available on <http://www.openmp.org/>.

5.3.7 Native Thread Identifiers

Summary

The `ompd_thread_id_t` type provides information about native threads.

Format

```
typedef uint64_t ompd_thread_id_t;
```

Description

Different OpenMP runtimes may use different native thread implementations. Native thread identifiers can vary in size and format and, thus, are not explicitly represented in the OMPD API. Instead, native thread identifiers are passed across the interface via the `ompd_thread_id_t` type, which is a pointer to where the native thread identifier is stored, and the size of the native thread identifier in bytes. The OMPD library and a tool that uses it must agree on the format of the object that is passed. Each different kind of native thread identifier uses a unique unsigned 64-bit integer value.

Recommended values of `ompd_thread_id_t` are defined in the `ompd-types.h` header file, which is available on <http://www.openmp.org/>.

5.3.8 OMPD Handle Types

Summary

OMPD handle types are opaque types.

Format

```
typedef struct _ompd_aspace_handle ompd_address_space_handle_t;  
typedef struct _ompd_thread_handle ompd_thread_handle_t;  
typedef struct _ompd_parallel_handle ompd_parallel_handle_t;  
typedef struct _ompd_task_handle ompd_task_handle_t;
```


Description

OMPD uses handles for address spaces (`ompd_address_space_handle_t`), threads (`ompd_thread_handle_t`), parallel regions (`ompd_parallel_handle_t`), and tasks (`ompd_task_handle_t`). Each operation of the OMPD interface that applies to a particular address space, thread, parallel region, or task must explicitly specify a corresponding handle. A handle for an entity is constant while the entity itself is alive. Handles are defined by the OMPD library, and are opaque to the tool.

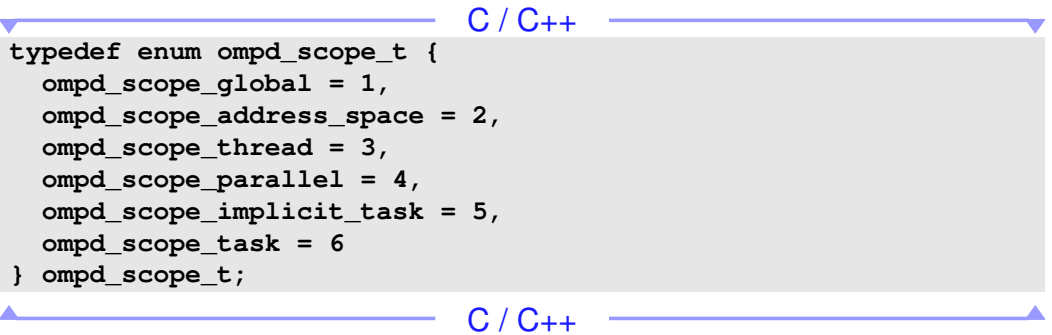
Defining externally visible type names in this way introduces type safety to the interface, and helps to catch instances where incorrect handles are passed by the tool to the OMPD library. The structures do not need to be defined; instead, the OMPD library must cast incoming (pointers to) handles to the appropriate internal, private types.

5.3.9 OMPD Scope Types

Summary

The `ompd_scope_t` type identifies OMPD scopes.

Format



```
typedef enum ompd_scope_t {  
    ompd_scope_global = 1,  
    ompd_scope_address_space = 2,  
    ompd_scope_thread = 3,  
    ompd_scope_parallel = 4,  
    ompd_scope_implicit_task = 5,  
    ompd_scope_task = 6  
} ompd_scope_t;
```

Description

The `ompd_scope_t` type identifies OpenMP scopes, including those related to parallel regions and tasks. When used in an OMPD interface function call, the scope type and the ompd handle must match according to Table 5.1.

TABLE 5.1: Mapping of Scope Type and OMPD Handles

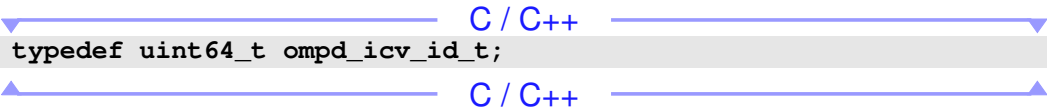
Scope types	Handles
<i>ompd_scope_global</i>	Address space handle for the host device
<i>ompd_scope_address_space</i>	Any address space handle
<i>ompd_scope_thread</i>	Any thread handle
<i>ompd_scope_parallel</i>	Any parallel handle
<i>ompd_scope_implicit_task</i>	Task handle for an implicit task
<i>ompd_scope_task</i>	Any task handle

1 **5.3.10 ICV ID Type**

2 **Summary**

3 The `ompd_icv_id_t` type identifies an OpenMP implementation ICV.

4 **Format**

5  `typedef uint64_t ompd_icv_id_t;`

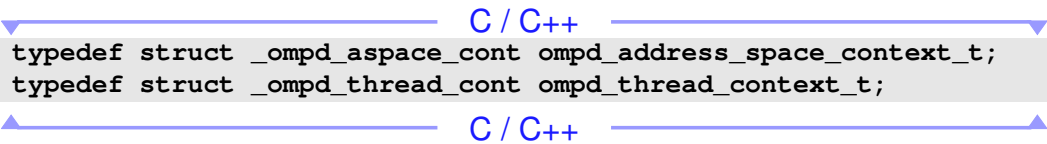
6 The `ompd_icv_id_t` type identifies OpenMP implementation ICVs. `ompd_icv_undefined`
7 is an instance of this type with the value 0.

8 **5.3.11 Tool Context Types**

9 **Summary**

10 A third-party tool uses contexts to uniquely identify abstractions. These contexts are opaque to the
11 OMPD library and are defined as follows:

12 **Format**

13  `typedef struct _ompd_aspace_cont ompd_address_space_context_t;`
14 `typedef struct _ompd_thread_cont ompd_thread_context_t;`

1 5.3.12 Return Code Types

2 Summary

3 The `ompd_rc_t` type is the return code type of OMPD operations

4 Format

C / C++

```
5 typedef enum ompd_rc_t {  
6     ompd_rc_ok = 0,  
7     ompd_rc_unavailable = 1,  
8     ompd_rc_stale_handle = 2,  
9     ompd_rc_bad_input = 3,  
10    ompd_rc_error = 4,  
11    ompd_rc_unsupported = 5,  
12    ompd_rc_needs_state_tracking = 6,  
13    ompd_rc_incompatible = 7,  
14    ompd_rc_device_read_error = 8,  
15    ompd_rc_device_write_error = 9,  
16    ompd_rc_nomem = 10,  
17 } ompd_rc_t;
```

C / C++

18 Description

19 The `ompd_rc_t` type is used for the return codes of OMPD operations. The return code types and
20 their semantics are defined as follows:

- 21 • `ompd_rc_ok` is returned when the operation is successful;
- 22 • `ompd_rc_unavailable` is returned when information is not available for the specified
23 context;
- 24 • `ompd_rc_stale_handle` is returned when the specified handle is no longer valid;
- 25 • `ompd_rc_bad_input` is returned when the input parameters (other than handle) are invalid;
- 26 • `ompd_rc_error` is returned when a fatal error occurred;
- 27 • `ompd_rc_unsupported` is returned when the requested operation is not supported;
- 28 • `ompd_rc_needs_state_tracking` is returned when the state tracking operation failed
29 because state tracking is not currently enabled;
- 30 • `ompd_rc_device_read_error` is returned when a read operation failed on the device;
- 31 • `ompd_rc_device_write_error` is returned when a write operation failed on the device;

- **ompd_rc_incompatible** is returned when this OMPD library is incompatible with, or is not capable of handling, the OpenMP program; and
- **ompd_rc_nomem** is returned when a memory allocation fails.

5.3.13 Primitive Type Sizes

Summary

The **ompd_device_type_sizes_t** type provides the “sizeof” of primitive types in the OpenMP architecture address space.

Format

C / C++

```
typedef struct ompd_device_type_sizes_t {  
    uint8_t sizeof_char;  
    uint8_t sizeof_short;  
    uint8_t sizeof_int;  
    uint8_t sizeof_long;  
    uint8_t sizeof_long_long;  
    uint8_t sizeof_pointer;  
} ompd_device_type_sizes_t;
```

C / C++

Description

The **ompd_device_type_sizes_t** type is used in operations through which the OMPD library can interrogate the tool about the “sizeof” of primitive types in the OpenMP architecture address space. The fields of **ompd_device_type_sizes_t** give the sizes of the eponymous basic types used by the OpenMP runtime. As the tool and the OMPD library, by definition, have the same architecture and programming model, the size of the fields can be given as **uint8_t**.

Cross References

- **ompd_callback_sizeof_fn_t**, see Section [5.4.2.2](#) on page [549](#)

1 5.4 OMPD Tool Callback Interface

2 For the OMPD library to provide information about the internal state of the OpenMP runtime
3 system in an OpenMP process or core file, it must have a means to extract information from the
4 OpenMP process that the tool is debugging. The OpenMP process on which the tool is operating
5 may be either a “live” process or a core file, and a thread may be either a “live” thread in an
6 OpenMP process, or a thread in a core file. To enable the OMPD library to extract state information
7 from an OpenMP process or core file, the tool must supply the OMPD library with callback
8 functions to inquire about the size of primitive types in the device of the OpenMP process, to look
9 up the addresses of symbols, and to read and to write memory in the device. The OMPD library
10 uses these callbacks to implement its interface operations. The OMPD library only invokes the
11 callback functions in direct response to calls made by the tool to the OMPD library.

12 5.4.1 Memory Management of OMPD Library

13 The OMPD library must not access the heap manager directly. Instead, if it needs heap memory it
14 must use the memory allocation and deallocation callback functions that are described in this
15 section, `ompd_callback_memory_alloc_fn_t` (see Section 5.4.1.1 on page 546) and
16 `ompd_callback_memory_free_fn_t` (see Section 5.4.1.2 on page 546), which are provided
17 by the tool to obtain and to release heap memory. This mechanism ensures that the library does not
18 interfere with any custom memory management scheme that the tool may use.

19 If the OMPD library is implemented in C++, memory management operators like **new** and
20 **delete** in all their variants, *must all* be overloaded and implemented in terms of the callbacks that
21 the tool provides. The OMPD library must be coded so that any of its definitions of **new** or
22 **delete** do not interfere with any that the tool defines.

23 In some cases, the OMPD library must allocate memory to return results to the tool. The tool then
24 owns this memory and has the responsibility to release it. Thus, the OMPD library and the tool
25 must use the same memory manager.

26 The OMPD library creates OMPD handles, which are opaque to the tool and may have a complex
27 internal structure. The tool cannot determine if the handle pointers that the API returns correspond
28 to discrete heap allocations. Thus, the tool must not simply deallocate a handle by passing an
29 address that it receives from the OMPD library to its own memory manager. Instead, the API
30 includes functions that the tool must use when it no longer needs a handle.

31 A tool creates contexts and passes them to the OMPD library. The OMPD library does not release
32 contexts; instead the tool release them after it releases any handles that may reference the contexts.

1 5.4.1.1 ompd_callback_memory_alloc_fn_t

2 Summary

3 The **ompd_callback_memory_alloc_fn_t** type is the type signature of the callback routine
4 that the tool provides to the OMPD library to allocate memory.

5 Format

```
6 typedef ompd_rc_t (*ompd_callback_memory_alloc_fn_t) (  
7     ompd_size_t nbytes,  
8     void **ptr  
9 );
```

10 Description

11 The **ompd_callback_memory_alloc_fn_t** type is the type signature of the memory
12 allocation callback routine that the tool provides. The OMPD library may call the
13 **ompd_callback_memory_alloc_fn_t** callback function to allocate memory.

14 Description of Arguments

15 The *nbytes* argument is the size in bytes of the block of memory to allocate.

16 The address of the newly allocated block of memory is returned in the location to which the *ptr*
17 argument points. The newly allocated block is suitably aligned for any type of variable, and is not
18 guaranteed to be zeroed.

19 Cross References

- 20 • **ompd_size_t**, see Section [5.3.1](#) on page [536](#).
- 21 • **ompd_rc_t**, see Section [5.3.12](#) on page [543](#).

22 5.4.1.2 ompd_callback_memory_free_fn_t

23 Summary

24 The **ompd_callback_memory_free_fn_t** type is the type signature of the callback routine
25 that the tool provides to the OMPD library to deallocate memory.

Format

```
typedef ompd_rc_t (*ompd_callback_memory_free_fn_t) (  
    void *ptr  
);
```

Description

The `ompd_callback_memory_free_fn_t` type is the type signature of the memory deallocation callback routine that the tool provides. The OMPD library may call the `ompd_callback_memory_free_fn_t` callback function to deallocate memory that was obtained from a prior call to the `ompd_callback_memory_alloc_fn_t` callback function.

Description of Arguments

The *ptr* argument is the address of the block to be deallocated.

Cross References

- `ompd_rc_t`, see Section 5.3.12 on page 543.
- `ompd_callback_memory_alloc_fn_t`, see Section 5.4.1.1 on page 546.
- `ompd_callbacks_t`, see Section 5.4.6 on page 556.

5.4.2 Context Management and Navigation

Summary

The tool provides the OMPD library with callbacks to manage and to navigate context relationships.

5.4.2.1 `ompd_callback_get_thread_context_for_thread_id_fn_t`

Summary

The `ompd_callback_get_thread_context_for_thread_id_fn_t` is the type signature of the callback routine that the tool provides to the OMPD library to map a thread identifier to a tool thread context.

Format

```
typedef ompd_rc_t  
(*ompd_callback_get_thread_context_for_thread_id_fn_t) (  
    ompd_address_space_context_t *address_space_context,  
    ompd_thread_id_t kind,  
    ompd_size_t sizeof_thread_id,  
    const void *thread_id,  
    ompd_thread_context_t **thread_context  
);
```

Description

The `ompd_callback_get_thread_context_for_thread_id_fn_t` is the type signature of the context mapping callback routine that the tool provides. This callback maps a thread identifier to a tool thread context. The thread identifier is within the address space that `address_space_context` identifies. The OMPD library can use the thread context, for example, to access thread local storage.

Description of Arguments

The `address_space_context` argument is an opaque handle that the tool provides to reference an address space. The `kind`, `sizeof_thread_id`, and `thread_id` arguments represent a native thread identifier. On return, the `thread_context` argument provides an opaque handle that maps a native thread identifier to a tool thread context.

Restrictions

Routines that use `ompd_callback_get_thread_context_for_thread_id_fn_t` have the following restriction:

- The provided `thread_context` must be valid until the OMPD library returns from the OMPD tool interface routine.

Cross References

- `ompd_size_t`, see Section 5.3.1 on page 536.
- `ompd_thread_id_t`, see Section 5.3.7 on page 539.
- `ompd_address_space_context_t`, see Section 5.3.11 on page 542.
- `ompd_thread_context_t`, see Section 5.3.11 on page 542.
- `ompd_rc_t`, see Section 5.3.12 on page 543.

1 5.4.2.2 ompd_callback_sizeof_fn_t

2 Summary

3 The `ompd_callback_sizeof_fn_t` type is the type signature of the callback routine that the
4 tool provides to the OMPD library to determine the sizes of the primitive types in an address space.

5 Format

```
6 typedef ompd_rc_t (*ompd_callback_sizeof_fn_t) (  
7     ompd_address_space_context_t *address_space_context,  
8     ompd_device_type_sizes_t *sizes  
9 );
```

10 Description

11 The `ompd_callback_sizeof_fn_t` is the type signature of the type-size query callback
12 routine that the tool provides. This callback provides the sizes of the basic primitive types for a
13 given address space.

14 Description of Arguments

15 The callback returns the sizes of the basic primitive types used by the address space context that the
16 *address_space_context* argument specifies in the location to which the *sizes* argument points.

17 Cross References

- 18 • `ompd_address_space_context_t`, see Section 5.3.11 on page 542.
- 19 • `ompd_rc_t`, see Section 5.3.12 on page 543.
- 20 • `ompd_device_type_sizes_t`, see Section 5.3.13 on page 544.
- 21 • `ompd_callbacks_t`, see Section 5.4.6 on page 556.

22 5.4.3 Accessing Memory in the OpenMP Program or Runtime

23 The OMPD library may need to read from or to write to the OpenMP program. It cannot do this
24 directly. Instead the OMPD library must use callbacks that the tool provides so that the tool
25 performs the operation.

1 5.4.3.1 ompd_callback_symbol_addr_fn_t

2 Summary

3 The **ompd_callback_symbol_addr_fn_t** type is the type signature of the callback that the
4 tool provides to look up the addresses of symbols in an OpenMP program.

5 Format

```
6 typedef ompd_rc_t (*ompd_callback_symbol_addr_fn_t) (  
7     ompd_address_space_context_t *address_space_context,  
8     ompd_thread_context_t *thread_context,  
9     const char *symbol_name,  
10    ompd_address_t *symbol_addr,  
11    const char *file_name  
12 );
```

13 Description

14 The **ompd_callback_symbol_addr_fn_t** is the type signature of the symbol-address query
15 callback routine that the tool provides. This callback looks up addresses of symbols within a
16 specified address space.

17 Description of Arguments

18 This callback looks up the symbol provided in the *symbol_name* argument.

19 The *address_space_context* argument is the tool's representation of the address space of the
20 process, core file, or device.

21 The *thread_context* argument is NULL for global memory access. If *thread_context* is not NULL,
22 *thread_context* gives the thread specific context for the symbol lookup, for the purpose of
23 calculating thread local storage addresses. If *thread_context* is non-null then the thread to which
24 *thread_context* refers must be associated with either the process or the device that corresponds to
25 the *address_space_context* argument.

26 The tool uses the *symbol_name* argument that the OMPD library supplies verbatim. In particular,
27 no name mangling, demangling or other transformations are performed prior to the lookup. The
28 *symbol_name* parameter must correspond to a statically allocated symbol within the specified
29 address space. The symbol can correspond to any type of object, such as a variable, thread local
30 storage variable, function, or untyped label. The symbol can have a local, global, or weak binding.

31 The *file_name* argument is an optional input parameter that indicates the name of the shared library
32 in which the symbol is defined, and is intended to help the third party tool disambiguate symbols

that are defined multiple times across the executable or shared library files. The shared library name may not be an exact match for the name seen by the tool. If *file_name* is NULL then the tool first tries to find the symbol in the executable file, and, if the symbol is not found, the tool tries to find the symbol in the shared libraries in the order in which the shared libraries are loaded into the address space. If *file_name* is non-null then the tool first tries to find the symbol in the libraries that match the name in the *file_name* argument and, if the symbol is not found, the tool then uses the same procedure as when *file_name* is NULL.

The callback does not support finding symbols that are dynamically allocated on the call stack, or statically allocated symbols that are defined within the scope of a function or subroutine.

The callback returns the symbol's address in the location to which *symbol_addr* points.

Restrictions

Routines that use the `ompd_callback_symbol_addr_fn_t` type have the following restrictions:

- The *address_space_context* argument must be non-null.
- The symbol that the *symbol_name* argument specifies must be defined.

Cross References

- `ompd_address_t`, see Section 5.3.4 on page 538.
- `ompd_address_space_context_t`, see Section 5.3.11 on page 542.
- `ompd_thread_context_t`, see Section 5.3.11 on page 542.
- `ompd_rc_t`, see Section 5.3.12 on page 543.
- `ompd_callbacks_t`, see Section 5.4.6 on page 556.

5.4.3.2 `ompd_callback_memory_read_fn_t`

Summary

The `ompd_callback_memory_read_fn_t` type is the type signature of the callback that the tool provides to read data from an OpenMP program.

Format

```
typedef ompd_rc_t (*ompd_callback_memory_read_fn_t) (  
    ompd_address_space_context_t *address_space_context,  
    ompd_thread_context_t *thread_context,  
    const ompd_address_t *addr,  
    ompd_size_t nbytes,  
    void *buffer  
);
```

Description

The **ompd_callback_memory_read_fn_t** is the type signature of the read callback routines that the tool provides.

The **read_memory** callback copies a block of data from *addr* within the address space to the tool *buffer*.

The **read_string** callback copies a string to which *addr* points, including the terminating null byte ('`\0`'), to the tool *buffer*. At most *nbytes* bytes are copied. If a null byte is not among the first *nbytes* bytes, the string placed in *buffer* is not null-terminated.

Description of Arguments

The address from which the data are to be read from the OpenMP program specified by *address_space_context* is given by *addr*. while *nbytes* gives the number of bytes to be transferred. The *thread_context* argument is optional for global memory access, and in this case should be NULL. If it is non-null, *thread_context* identifies the thread specific context for the memory access for the purpose of accessing thread local storage.

The data are returned through *buffer*, which is allocated and owned by the OMPD library. The contents of the buffer are unstructured, raw bytes. The OMPD library must arrange for any transformations such as byte-swapping that may be necessary (see Section 5.4.4 on page 554) to interpret the data.

Cross References

- `ompd_size_t`, see Section 5.3.1 on page 536.
- `ompd_address_t`, see Section 5.3.4 on page 538.
- `ompd_address_space_context_t`, see Section 5.3.11 on page 542.
- `ompd_thread_context_t`, see Section 5.3.11 on page 542.
- `ompd_rc_t`, see Section 5.3.12 on page 543.
- `ompd_callback_device_host_fn_t`, see Section 5.4.4 on page 554.
- `ompd_callbacks_t`, see Section 5.4.6 on page 556.

5.4.3.3 `ompd_callback_memory_write_fn_t`

Summary

The `ompd_callback_memory_write_fn_t` type is the type signature of the callback that the tool provides to write data to an OpenMP program.

Format

```
typedef ompd_rc_t (*ompd_callback_memory_write_fn_t) (  
    ompd_address_space_context_t *address_space_context,  
    ompd_thread_context_t *thread_context,  
    const ompd_address_t *addr,  
    ompd_size_t nbytes,  
    const void *buffer  
);
```

Description

The `ompd_callback_memory_write_fn_t` is the type signature of the write callback routine that the tool provides. The OMPD library may call this callback to have the tool write a block of data to a location within an address space from a provided buffer.

Description of Arguments

The address to which the data are to be written in the OpenMP program that *address_space_context* specifies is given by *addr*. The *nbytes* argument is the number of bytes to be transferred. The *thread_context* argument is optional for global memory access, and, in this case, should be NULL. If it is non-null then *thread_context* identifies the thread-specific context for the memory access for the purpose of accessing thread local storage.

The data to be written are passed through *buffer*, which is allocated and owned by the OMPD library. The contents of the buffer are unstructured, raw bytes. The OMPD library must arrange for any transformations such as byte-swapping that may be necessary (see Section 5.4.4 on page 554) to render the data into a form that is compatible with the OpenMP runtime.

Cross References

- `ompd_size_t`, see Section 5.3.1 on page 536.
- `ompd_address_t`, see Section 5.3.4 on page 538.
- `ompd_address_space_context_t`, see Section 5.3.11 on page 542.
- `ompd_thread_context_t`, see Section 5.3.11 on page 542.
- `ompd_rc_t`, see Section 5.3.12 on page 543.
- `ompd_callback_device_host_fn_t`, see Section 5.4.4 on page 554.
- `ompd_callbacks_t`, see Section 5.4.6 on page 556.

5.4.4 Data Format Conversion: `ompd_callback_device_host_fn_t`

Summary

The `ompd_callback_device_host_fn_t` type is the type signature of the callback that the tool provides to convert data between the formats that the tool and the OMPD library use and that the OpenMP program uses.

Format

```
typedef ompd_rc_t (*ompd_callback_device_host_fn_t) (  
    ompd_address_space_context_t *address_space_context,  
    const void *input,  
    ompd_size_t unit_size,  
    ompd_size_t count,  
    void *output  
);
```

Description

The architecture and/or programming-model of the tool and the OMPD library may be different from that of the OpenMP program that is being examined. Thus, the conventions for representing data may differ. The callback interface includes operations to convert between the conventions, such as the byte order (endianness), that the tool and OMPD library use and the one that the OpenMP program uses. The callback with the `ompd_callback_device_host_fn_t` type signature convert data between formats

Description of Arguments

The `address_space_context` argument specifies the OpenMP address space that is associated with the data. The `input` argument is the source buffer and the `output` argument is the destination buffer. The `unit_size` argument is the size of each of the elements to be converted. The `count` argument is the number of elements to be transformed.

The OMPD library allocates and owns the input and output buffers. It must ensure that the buffers have the correct size, and are eventually deallocated when they are no longer needed.

Cross References

- `ompd_size_t`, see Section 5.3.1 on page 536.
- `ompd_address_space_context_t`, see Section 5.3.11 on page 542.
- `ompd_rc_t`, see Section 5.3.12 on page 543.
- `ompd_callbacks_t`, see Section 5.4.6 on page 556.

1 5.4.5 Output: `ompd_callback_print_string_fn_t`

2 Summary

3 The `ompd_callback_print_string_fn_t` type is the type signature of the callback that
4 tool provides so that the OMPD library can emit output.

5 Format

```
6 typedef ompd_rc_t (*ompd_callback_print_string_fn_t) (  
7     const char *string,  
8     int category  
9 );
```

10 Description

11 The OMPD library may call the `ompd_callback_print_string_fn_t` callback function to
12 emit output, such as logging or debug information. The tool may set the
13 `ompd_callback_print_string_fn_t` callback function to NULL to prevent the OMPD
14 library from emitting output; the OMPD may not write to file descriptors that it did not open.

15 Description of Arguments

16 The *string* argument is the null-terminated string to be printed. No conversion or formatting is
17 performed on the string.

18 The *category* argument is the implementation-defined category of the string to be printed.

19 Cross References

- 20 • `ompd_rc_t`, see Section [5.3.12](#) on page [543](#).
- 21 • `ompd_callbacks_t`, see Section [5.4.6](#) on page [556](#).

22 5.4.6 The Callback Interface

23 Summary

24 All OMPD library interactions with the OpenMP program must be through a set of callbacks that
25 the tool provides. These callbacks must also be used for allocating or releasing resources, such as
26 memory, that the library needs.

Format

```
typedef struct ompd_callbacks_t {
    ompd_callback_memory_alloc_fn_t alloc_memory;
    ompd_callback_memory_free_fn_t free_memory;
    ompd_callback_print_string_fn_t print_string;
    ompd_callback_sizeof_fn_t sizeof_type;
    ompd_callback_symbol_addr_fn_t symbol_addr_lookup;
    ompd_callback_memory_read_fn_t read_memory;
    ompd_callback_memory_write_fn_t write_memory;
    ompd_callback_memory_read_fn_t read_string;
    ompd_callback_device_host_fn_t device_to_host;
    ompd_callback_device_host_fn_t host_to_device;
    ompd_callback_get_thread_context_for_thread_id_fn_t
        get_thread_context_for_thread_id;
} ompd_callbacks_t;
```

Description

The set of callbacks that the OMPD library must use is collected in the **ompd_callbacks_t** record structure. An instance of this type is passed to the OMPD library as a parameter to **ompd_initialize** (see Section 5.5.1.1 on page 558). Each field points to a function that the OMPD library must use to interact with the OpenMP program or for memory operations.

The *alloc_memory* and *free_memory* fields are pointers to functions the OMPD library uses to allocate and to release dynamic memory.

print_string points to a function that prints a string.

The architectures or programming models of the OMPD library and third party tool may be different from that of the OpenMP program that is being examined. *sizeof_type* points to function that allows the OMPD library to determine the sizes of the basic integer and pointer types that the OpenMP program uses. Because of the differences in architecture or programming model, the conventions for representing data in the OMPD library and the OpenMP program may be different. The *device_to_host* field points to a function that translates data from the conventions that the OpenMP program uses to those that the tool and OMPD library use. The reverse operation is performed by the function to which the *host_to_device* field points.

The *symbol_addr_lookup* field points to a callback that the OMPD library can use to find the address of a global or thread local storage symbol. The *read_memory*, *read_string*, and *write_memory* fields are pointers to functions for reading from and writing to global memory or thread local storage in the OpenMP program.

The *get_thread_context_for_thread_id* field is a pointer to a function that the OMPD library can use to obtain a thread context that corresponds to a native thread identifier.

Cross References

- `ompd_callback_memory_alloc_fn_t`, see Section 5.4.1.1 on page 546.
- `ompd_callback_memory_free_fn_t`, see Section 5.4.1.2 on page 546.
- `ompd_callback_get_thread_context_for_thread_id_fn_t`, see Section 5.4.2.1 on page 547.
- `ompd_callback_sizeof_fn_t`, see Section 5.4.2.2 on page 549.
- `ompd_callback_symbol_addr_fn_t`, see Section 5.4.3.1 on page 550.
- `ompd_callback_memory_read_fn_t`, see Section 5.4.3.2 on page 551.
- `ompd_callback_memory_write_fn_t`, see Section 5.4.3.3 on page 553.
- `ompd_callback_device_host_fn_t`, see Section 5.4.4 on page 554.
- `ompd_callback_print_string_fn_t`, see Section 5.4.5 on page 556

5.5 OMPD Tool Interface Routines

5.5.1 Per OMPD Library Initialization and Finalization

The OMPD library must be initialized exactly once after it is loaded, and finalized exactly once before it is unloaded. Per OpenMP process or core file initialization and finalization are also required.

Once loaded, the tool can determine the version of the OMPD API that the library supports by calling `ompd_get_api_version` (see Section 5.5.1.2 on page 559). If the tool supports the version that `ompd_get_api_version` returns, the tool starts the initialization by calling `ompd_initialize` (see Section 5.5.1.1 on page 558) using the version of the OMPD API that the library supports. If the tool does not support the version that `ompd_get_api_version` returns, it may attempt to call `ompd_initialize` with a different version.

5.5.1.1 `ompd_initialize`

Summary

The `ompd_initialize` function initializes the OMPD library.

Format

```
ompd_rc_t ompd_initialize(  
    ompd_word_t api_version,  
    const ompd_callbacks_t *callbacks  
);
```

Description

A tool that uses OMPD calls **ompd_initialize** to initialize each OMPD library that it loads. More than one library may be present in a third-party tool, such as a debugger, because the tool may control multiple devices, which may use different runtime systems that require different OMPD libraries. This initialization must be performed exactly once before the tool can begin to operate on an OpenMP process or core file.

Description of Arguments

The *api_version* argument is the OMPD API version that the tool requests to use. The tool may call **ompd_get_api_version** to obtain the latest version that the OMPD library supports.

The tool provides the OMPD library with a set of callback functions in the *callbacks* input argument which enables the OMPD library to allocate and to deallocate memory in the tool's address space, to lookup the sizes of basic primitive types in the device, to lookup symbols in the device, and to read and to write memory in the device.

Cross References

- **ompd_rc_t** type, see Section 5.3.12 on page 543.
- **ompd_callbacks_t** type, see Section 5.4.6 on page 556.
- **ompd_get_api_version** call, see Section 5.5.1.2 on page 559.

5.5.1.2 ompd_get_api_version

Summary

The **ompd_get_api_version** function returns the OMPD API version.

Format

```
ompd_rc_t ompd_get_api_version(ompd_word_t *version);
```

Description

The tool may call the **ompd_get_api_version** function to obtain the latest OMPD API version number of the OMPD library.

Description of Arguments

The latest version number is returned into the location to which the *version* argument points.

Cross References

- **ompd_rc_t** type, see Section 5.3.12 on page 543.

5.5.1.3 ompd_get_version_string

Summary

The **ompd_get_version_string** function returns a descriptive string for the OMPD API version.

Format

```
ompd_rc_t ompd_get_version_string(const char **string);
```

Description

The tool may call this function to obtain a pointer to a descriptive version string of the OMPD API version.

Description of Arguments

A pointer to a descriptive version string is placed into the location to which *string* output argument points. The OMPD library owns the string that the OMPD library returns; the tool must not modify or release this string. The string remains valid for as long as the library is loaded. The `ompd_get_version_string` function may be called before `ompd_initialize` (see Section 5.5.1.1 on page 558). Accordingly, the OMPD library must not use heap or stack memory for the string.

The signatures of `ompd_get_api_version` (see Section 5.5.1.2 on page 559) and `ompd_get_version_string` are guaranteed not to change in future versions of the API. In contrast, the type definitions and prototypes in the rest of the API do not carry the same guarantee. Therefore a tool that uses OMPD should check the version of the API of the loaded OMPD library before it calls any other function of the API.

Cross References

- `ompd_rc_t` type, see Section 5.3.12 on page 543.

5.5.1.4 `ompd_finalize`

Summary

When the tool is finished with the OMPD library it should call `ompd_finalize` before it unloads the library.

Format

```
ompd_rc_t ompd_finalize(void);
```

Description

The call to `ompd_finalize` must be the last OMPD call that the tool makes before it unloads the library. This call allows the OMPD library to free any resources that it may be holding.

The OMPD library may implement a *finalizer* section, which executes as the library is unloaded and therefore after the call to `ompd_finalize`. During finalization, the OMPD library may use the callbacks that the tool earlier provided after the call to `ompd_initialize`.

Cross References

- `ompd_rc_t` type, see Section 5.3.12 on page 543.

5.5.2 Per OpenMP Process Initialization and Finalization

5.5.2.1 `ompd_process_initialize`

Summary

A tool calls `ompd_process_initialize` to obtain an address space handle when it initializes a session on a live process or core file.

Format

```
ompd_rc_t ompd_process_initialize(  
    ompd_address_space_context_t *context,  
    ompd_address_space_handle_t **handle  
);
```

Description

A tool calls `ompd_process_initialize` to obtain an address space handle when it initializes a session on a live process or core file. On return from `ompd_process_initialize`, the tool owns the address space handle, which it must release with `ompd_rel_address_space_handle`. The initialization function must be called before any OMPD operations are performed on the OpenMP process. This call allows the OMPD library to confirm that it can handle the OpenMP process or core file that the *context* identifies. Incompatibility is signaled by a return value of `ompd_rc_incompatible`.

Description of Arguments

The *context* argument is an opaque handle that the tool provides to address an address space. On return, the *handle* argument provides an opaque handle to the tool for this address space, which the tool must release when it is no longer needed.

Cross References

- `ompd_address_space_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_address_space_context_t` type, see Section 5.3.11 on page 542.
- `ompd_rc_t` type, see Section 5.3.12 on page 543.
- `ompd_rel_address_space_handle` type, see Section 5.5.2.3 on page 564.

5.5.2.2 `ompd_device_initialize`

Summary

A tool calls `ompd_device_initialize` to obtain an address space handle for a device that has at least one active target region.

Format

```
ompd_rc_t ompd_device_initialize(  
    ompd_address_space_handle_t *process_handle,  
    ompd_address_space_context_t *device_context,  
    ompd_device_t kind,  
    ompd_size_t sizeof_id,  
    void *id,  
    ompd_address_space_handle_t **device_handle  
);
```

Description

A tool calls `ompd_device_initialize` to obtain an address space handle for a device that has at least one active target region. On return from `ompd_device_initialize`, the tool owns the address space handle.

Description of Arguments

The *process_handle* argument is an opaque handle that the tool provides to reference the address space of the OpenMP process. The *device_context* argument is an opaque handle that the tool provides to reference a device address space. The *kind*, *sizeof_id*, and *id* arguments represent a device identifier. On return the *device_handle* argument provides an opaque handle to the tool for this address space.

Cross References

- `ompd_size_t` type, see Section 5.3.1 on page 536.
- `ompd_device_t` type, see Section 5.3.6 on page 539.
- `ompd_address_space_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_address_space_context_t` type, see Section 5.3.11 on page 542.
- `ompd_rc_t` type, see Section 5.3.12 on page 543.

5.5.2.3 `ompd_rel_address_space_handle`

Summary

A tool calls `ompd_rel_address_space_handle` to release an address space handle.

Format

```
ompd_rc_t ompd_rel_address_space_handle(  
    ompd_address_space_handle_t *handle  
);
```

Description

When the tool is finished with the OpenMP process address space handle it should call `ompd_rel_address_space_handle` to release the handle, which allows the OMPD library to release any resources that it has related to the address space.

Description of Arguments

The *handle* argument is an opaque handle for the address space to be released.

Restrictions

The `ompd_rel_address_space_handle` has the following restriction:

- An address space context must not be used after the corresponding address space handle is released.

Cross References

- `ompd_address_space_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_rc_t` type, see Section 5.3.12 on page 543.

5.5.3 Thread and Signal Safety

The OMPD library does not need to be reentrant. The tool must ensure that only one thread enters the OMPD library at a time. The OMPD library must not install signal handlers or otherwise interfere with the tool's signal configuration.

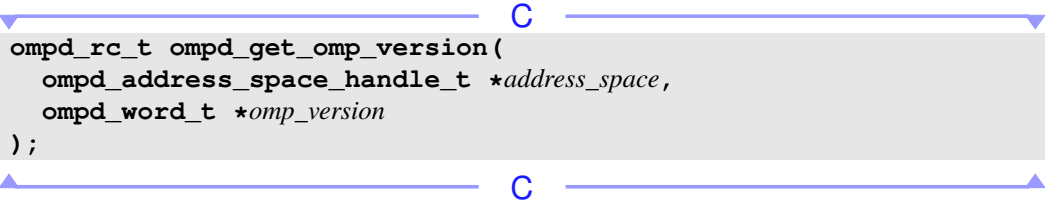
5.5.4 Address Space Information

5.5.4.1 `ompd_get_omp_version`

Summary

The tool may call the `ompd_get_omp_version` function to obtain the version of the OpenMP API that is associated with an address space.

Format

A diagram showing the C function signature for `ompd_get_omp_version`. The signature is enclosed in a light gray box. Above the box, a blue arrow points from the left to the box, and another blue arrow points from the box to the right. Below the box, a blue arrow points from the left to the box, and another blue arrow points from the box to the right. The signature is:

```
ompd_rc_t ompd_get_omp_version(  
    ompd_address_space_handle_t *address_space,  
    ompd_word_t *omp_version  
);
```

```
ompd_rc_t ompd_get_omp_version(  
    ompd_address_space_handle_t *address_space,  
    ompd_word_t *omp_version  
);
```

Description

The tool may call the `ompd_get_omp_version` function to obtain the version of the OpenMP API that is associated with the address space.

Description of Arguments

The *address_space* argument is an opaque handle that the tool provides to reference the address space of the OpenMP process or device.

Upon return, the *omp_version* argument contains the version of the OpenMP runtime in the `_OPENMP` version macro format.

Cross References

- `ompd_address_space_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_rc_t` type, see Section 5.3.12 on page 543.

5.5.4.2 `ompd_get_omp_version_string`

Summary

The `ompd_get_omp_version_string` function returns a descriptive string for the OpenMP API version that is associated with an address space.

Format

```
ompd_rc_t ompd_get_omp_version_string(  
    ompd_address_space_handle_t *address_space,  
    const char **string  
);
```

Description

After initialization, the tool may call the `ompd_get_omp_version_string` function to obtain the version of the OpenMP API that is associated with an address space.

Description of Arguments

The *address_space* argument is an opaque handle that the tool provides to reference the address space of the OpenMP process or device. A pointer to a descriptive version string is placed into the location to which the *string* output argument points. After returning from the call, the tool owns the string. The OMPD library must use the memory allocation callback that the tool provides to allocate the string storage. The tool is responsible for releasing the memory.

Cross References

- `ompd_address_space_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_rc_t` type, see Section 5.3.12 on page 543.

5.5.5 Thread Handles

5.5.5.1 `ompd_get_thread_in_parallel`

Summary

The `ompd_get_thread_in_parallel` function enables a tool to obtain handles for OpenMP threads that are associated with a parallel region.

Format

```
ompd_rc_t ompd_get_thread_in_parallel(  
    ompd_parallel_handle_t *parallel_handle,  
    int thread_num,  
    ompd_thread_handle_t **thread_handle  
);
```

Description

A successful invocation of `ompd_get_thread_in_parallel` returns a pointer to a thread handle in the location to which `thread_handle` points. This call yields meaningful results only if all OpenMP threads in the parallel region are stopped.

Description of Arguments

The `parallel_handle` argument is an opaque handle for a parallel region and selects the parallel region on which to operate. The `thread_num` argument selects the thread of the team to be returned. On return, the `thread_handle` argument is an opaque handle for the selected thread.

Restrictions

The `ompd_get_thread_in_parallel` function has the following restriction:

- The value of `thread_num` must be a non-negative integer smaller than the team size that was provided as the `ompd-team-size-var` from `ompd_get_icv_from_scope`.

Cross References

- `ompd_parallel_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_thread_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_rc_t` type, see Section 5.3.12 on page 543.
- `ompd_get_icv_from_scope` call, see Section 5.5.9.2 on page 590.

5.5.5.2 `ompd_get_thread_handle`

Summary

The `ompd_get_thread_handle` function maps a native thread to an OMPD thread handle.

Format

```
ompd_rc_t ompd_get_thread_handle(  
    ompd_address_space_handle_t *handle,  
    ompd_thread_id_t kind,  
    ompd_size_t sizeof_thread_id,  
    const void *thread_id,  
    ompd_thread_handle_t **thread_handle  
);
```

Description

The `ompd_get_thread_handle` function determines if the native thread identifier to which *thread_id* points represents an OpenMP thread. If so, the function returns `ompd_rc_ok` and the location to which *thread_handle* points is set to the thread handle for the OpenMP thread.

Description of Arguments

The *handle* argument is an opaque handle that the tool provides to reference an address space. The *kind*, *sizeof_thread_id*, and *thread_id* arguments represent a native thread identifier. On return, the *thread_handle* argument provides an opaque handle to the thread within the provided address space.

The native thread identifier to which *thread_id* points is guaranteed to be valid for the duration of the call. If the OMPD library must retain the native thread identifier, it must copy it.

Cross References

- `ompd_size_t` type, see Section 5.3.1 on page 536.
- `ompd_thread_id_t` type, see Section 5.3.7 on page 539.
- `ompd_address_space_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_thread_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_rc_t` type, see Section 5.3.12 on page 543.

5.5.5.3 `ompd_rel_thread_handle`

Summary

The `ompd_rel_thread_handle` function releases a thread handle.

Format

```
ompd_rc_t ompd_rel_thread_handle(  
    ompd_thread_handle_t *thread_handle  
);
```

Description

Thread handles are opaque to tools, which therefore cannot release them directly. Instead, when the tool is finished with a thread handle it must pass it to `ompd_rel_thread_handle` for disposal.

Description of Arguments

The *thread_handle* argument is an opaque handle for a thread to be released.

Cross References

- `ompd_thread_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_rc_t` type, see Section 5.3.12 on page 543.

1 5.5.5.4 ompd_thread_handle_compare

2 Summary

3 The **ompd_thread_handle_compare** function allows tools to compare two thread handles.

4 Format

```
5      ompd_rc_t ompd_thread_handle_compare(  
6          ompd_thread_handle_t *thread_handle_1,  
7          ompd_thread_handle_t *thread_handle_2,  
8          int *cmp_value  
9      );
```

10 Description

11 The internal structure of thread handles is opaque to a tool. While the tool can easily compare
12 pointers to thread handles, it cannot determine whether handles of two different addresses refer to
13 the same underlying thread. The **ompd_thread_handle_compare** function compares thread
14 handles.

15 On success, **ompd_thread_handle_compare** returns in the location to which *cmp_value*
16 points a signed integer value that indicates how the underlying threads compare: a value less than,
17 equal to, or greater than 0 indicates that the thread corresponding to *thread_handle_1* is,
18 respectively, less than, equal to, or greater than that corresponding to *thread_handle_2*.

19 Description of Arguments

20 The *thread_handle_1* and *thread_handle_2* arguments are opaque handles for threads. On return
21 the *cmp_value* argument is set to a signed integer value.

22 Cross References

- 23 • **ompd_thread_handle_t** type, see Section 5.3.8 on page 540.
- 24 • **ompd_rc_t** type, see Section 5.3.12 on page 543.

25 5.5.5.5 ompd_get_thread_id

26 Summary

27 The **ompd_get_thread_id** maps an OMPD thread handle to a native thread.

Format

```
ompd_rc_t ompd_get_thread_id(  
    ompd_thread_handle_t *thread_handle,  
    ompd_thread_id_t kind,  
    ompd_size_t sizeof_thread_id,  
    void *thread_id  
);
```

Description

The `ompd_get_thread_id` function maps an OMPD thread handle to a native thread identifier.

Description of Arguments

The `thread_handle` argument is an opaque thread handle. The `kind` argument represents the native thread identifier. The `sizeof_thread_id` argument represents the size of the native thread identifier. On return, the `thread_id` argument is a buffer that represents a native thread identifier.

Cross References

- `ompd_size_t` type, see Section 5.3.1 on page 536.
- `ompd_thread_id_t` type, see Section 5.3.7 on page 539.
- `ompd_thread_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_rc_t` type, see Section 5.3.12 on page 543.

5.5.6 Parallel Region Handles

5.5.6.1 `ompd_get_curr_parallel_handle`

Summary

The `ompd_get_curr_parallel_handle` function obtains a pointer to the parallel handle for an OpenMP thread's current parallel region.

Format

```
ompd_rc_t ompd_get_curr_parallel_handle(  
    ompd_thread_handle_t *thread_handle,  
    ompd_parallel_handle_t **parallel_handle  
);
```

Description

The `ompd_get_curr_parallel_handle` function enables the tool to obtain a pointer to the parallel handle for the current parallel region that is associated with an OpenMP thread. This call is meaningful only if the associated thread is stopped. The parallel handle must be released by calling `ompd_rel_parallel_handle`.

Description of Arguments

The *thread_handle* argument is an opaque handle for a thread and selects the thread on which to operate. On return, the *parallel_handle* argument is set to a handle for the parallel region that the associated thread is currently executing, if any.

Cross References

- `ompd_thread_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_parallel_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_rc_t` type, see Section 5.3.12 on page 543.
- `ompd_rel_parallel_handle` call, see Section 5.5.6.4 on page 574.

5.5.6.2 `ompd_get_enclosing_parallel_handle`

Summary

The `ompd_get_enclosing_parallel_handle` function obtains a pointer to the parallel handle for an enclosing parallel region.

Format

```
ompd_rc_t ompd_get_enclosing_parallel_handle(  
    ompd_parallel_handle_t *parallel_handle,  
    ompd_parallel_handle_t **enclosing_parallel_handle  
);
```

Description

The **ompd_get_enclosing_parallel_handle** function enables a tool to obtain a pointer to the parallel handle for the parallel region that encloses the parallel region that **parallel_handle** specifies. This call is meaningful only if at least one thread in the parallel region is stopped. A pointer to the parallel handle for the enclosing region is returned in the location to which *enclosing_parallel_handle* points. After the call, the tool owns the handle; the tool must release the handle with **ompd_rel_parallel_handle** when it is no longer required.

Description of Arguments

The *parallel_handle* argument is an opaque handle for a parallel region that selects the parallel region on which to operate. On return, the *enclosing_parallel_handle* argument is set to a handle for the parallel region that encloses the selected parallel region.

Cross References

- **ompd_parallel_handle_t** type, see Section 5.3.8 on page 540.
- **ompd_rc_t** type, see Section 5.3.12 on page 543.
- **ompd_rel_parallel_handle** call, see Section 5.5.6.4 on page 574.

5.5.6.3 ompd_get_task_parallel_handle

Summary

The **ompd_get_task_parallel_handle** function obtains a pointer to the parallel handle for the parallel region that encloses a task region.

Format

```
ompd_rc_t ompd_get_task_parallel_handle(  
    ompd_task_handle_t *task_handle,  
    ompd_parallel_handle_t **task_parallel_handle  
);
```

Description

The **ompd_get_task_parallel_handle** function enables a tool to obtain a pointer to the parallel handle for the parallel region that encloses the task region that *task_handle* specifies. This call is meaningful only if at least one thread in the parallel region is stopped. A pointer to the parallel regions handle is returned in the location to which *task_parallel_handle* points. The tool owns that parallel handle, which it must release with **ompd_rel_parallel_handle**.

Description of Arguments

The *task_handle* argument is an opaque handle that selects the task on which to operate. On return, the *parallel_handle* argument is set to a handle for the parallel region that encloses the selected task.

Cross References

- **ompd_task_handle_t** type, see Section 5.3.8 on page 540.
- **ompd_parallel_handle_t** type, see Section 5.3.8 on page 540.
- **ompd_rc_t** type, see Section 5.3.12 on page 543.
- **ompd_rel_parallel_handle** call, see Section 5.5.6.4 on page 574.

5.5.6.4 ompd_rel_parallel_handle

Summary

The **ompd_rel_parallel_handle** function releases a parallel region handle.

Format

```
ompd_rc_t ompd_rel_parallel_handle(  
    ompd_parallel_handle_t *parallel_handle  
);
```

Description

Parallel region handles are opaque so tools cannot release them directly. Instead, a tool must pass a parallel region handle to the `ompd_rel_parallel_handle` function for disposal when finished with it.

Description of Arguments

The *parallel_handle* argument is an opaque handle to be released.

Cross References

- `ompd_parallel_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_rc_t` type, see Section 5.3.12 on page 543.

5.5.6.5 `ompd_parallel_handle_compare`

Summary

The `ompd_parallel_handle_compare` function compares two parallel region handles.

Format

```
ompd_rc_t ompd_parallel_handle_compare(  
    ompd_parallel_handle_t *parallel_handle_1,  
    ompd_parallel_handle_t *parallel_handle_2,  
    int *cmp_value  
);
```

Description

The internal structure of parallel region handles is opaque to tools. While tools can easily compare pointers to parallel region handles, they cannot determine whether handles at two different addresses refer to the same underlying parallel region and, instead must use the `ompd_parallel_handle_compare` function.

On success, `ompd_parallel_handle_compare` returns a signed integer value in the location to which *cmp_value* points that indicates how the underlying parallel regions compare. A value less than, equal to, or greater than 0 indicates that the region corresponding to *parallel_handle_1* is, respectively, less than, equal to, or greater than that corresponding to *parallel_handle_2*. This function is provided since the means by which parallel region handles are ordered is implementation defined.

Description of Arguments

The *parallel_handle_1* and *parallel_handle_2* arguments are opaque handles that correspond to parallel regions. On return the *cmp_value* argument points to a signed integer value that indicates how the underlying parallel regions compare.

Cross References

- `ompd_parallel_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_rc_t` type, see Section 5.3.12 on page 543.

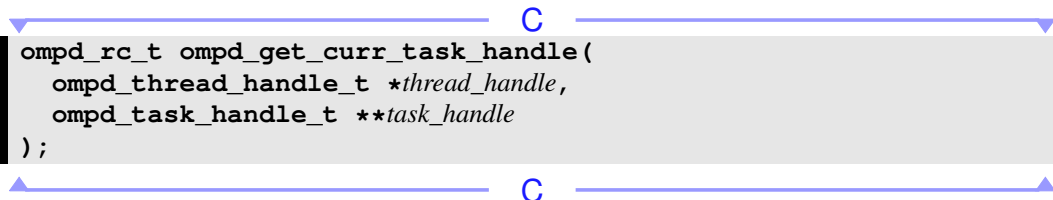
5.5.7 Task Handles

5.5.7.1 `ompd_get_curr_task_handle`

Summary

The `ompd_get_curr_task_handle` function obtains a pointer to the task handle for the current task region that is associated with an OpenMP thread.

Format



```
ompd_rc_t ompd_get_curr_task_handle(  
    ompd_thread_handle_t *thread_handle,  
    ompd_task_handle_t **task_handle  
);
```

Description

The `ompd_get_curr_task_handle` function obtains a pointer to the task handle for the current task region that is associated with an OpenMP thread. This call is meaningful only if the thread for which the handle is provided is stopped. The task handle must be released with `ompd_rel_task_handle`.

Description of Arguments

The *thread_handle* argument is an opaque handle that selects the thread on which to operate. On return, the *task_handle* argument points to a location that points to a handle for the task that the thread is currently executing.

Cross References

- `ompd_thread_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_task_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_rc_t` type, see Section 5.3.12 on page 543.
- `ompd_rel_task_handle` call, see Section 5.5.7.5 on page 580.

5.5.7.2 `ompd_get_generating_task_handle`

Summary

The `ompd_get_generating_task_handle` function obtains a pointer to the task handle of the generating task region.

Format

```
C  
ompd_rc_t ompd_get_generating_task_handle(  
    ompd_task_handle_t *task_handle,  
    ompd_task_handle_t **generating_task_handle  
) ;  
C
```

Description

The `ompd_get_generating_task_handle` function obtains a pointer to the task handle for the task that encountered the OpenMP task construct that generated the task represented by *task_handle*. The generating task is the OpenMP task that was active when the task specified by *task_handle* was created. This call is meaningful only if the thread that is executing the task that *task_handle* specifies is stopped. The generating task handle must be released with `ompd_rel_task_handle`.

Description of Arguments

The *task_handle* argument is an opaque handle that selects the task on which to operate. On return, the *generating_task_handle* argument points to a location that points to a handle for the generating task.

Cross References

- `ompd_task_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_rc_t` type, see Section 5.3.12 on page 543.
- `ompd_rel_task_handle` call, see Section 5.5.7.5 on page 580.

5.5.7.3 `ompd_get_scheduling_task_handle`

Summary

The `ompd_get_scheduling_task_handle` function obtains a task handle for the task that was active at a task scheduling point.

Format

```
ompd_rc_t ompd_get_scheduling_task_handle(  
    ompd_task_handle_t *task_handle,  
    ompd_task_handle_t **scheduling_task_handle  
);
```

Description

The `ompd_get_scheduling_task_handle` function obtains a task handle for the task that was active when the task that *task_handle* represents was scheduled. This call is meaningful only if the thread that is executing the task that *task_handle* specifies is stopped. The scheduling task handle must be released with `ompd_rel_task_handle`.

Description of Arguments

The *task_handle* argument is an opaque handle for a task and selects the task on which to operate. On return, the *scheduling_task_handle* argument points to a location that points to a handle for the task that is still on the stack of execution on the same thread and was deferred in favor of executing the selected task.

Cross References

- `ompd_task_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_rc_t` type, see Section 5.3.12 on page 543.
- `ompd_rel_task_handle` call, see Section 5.5.7.5 on page 580.

1 5.5.7.4 ompd_get_task_in_parallel

2 Summary

3 The **ompd_get_task_in_parallel** function obtains handles for the implicit tasks that are
4 associated with a parallel region.

5 Format

```
6  ompd_rc_t ompd_get_task_in_parallel(  
7      ompd_parallel_handle_t *parallel_handle,  
8      int thread_num,  
9      ompd_task_handle_t **task_handle  
10 );
```

11 Description

12 The **ompd_get_task_in_parallel** function obtains handles for the implicit tasks that are
13 associated with a parallel region. A successful invocation of **ompd_get_task_in_parallel**
14 returns a pointer to a task handle in the location to which *task_handle* points. This call yields
15 meaningful results only if all OpenMP threads in the parallel region are stopped.

16 Description of Arguments

17 The *parallel_handle* argument is an opaque handle that selects the parallel region on which to
18 operate. The *thread_num* argument selects the implicit task of the team that is returned. The
19 selected implicit task would return *thread_num* from a call of the **omp_get_thread_num()**
20 routine. On return, the *task_handle* argument points to a location that points to an opaque handle
21 for the selected implicit task.

22 Restrictions

23 The following restriction applies to the **ompd_get_task_in_parallel** function:

- 24 • The value of *thread_num* must be a non-negative integer that is smaller than the size of the team
25 size that is the value of the *ompd-team-size-var* that **ompd_get_icv_from_scope** returns.

Cross References

- `ompd_parallel_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_task_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_rc_t` type, see Section 5.3.12 on page 543.
- `ompd_get_icv_from_scope` call, see Section 5.5.9.2 on page 590.

5.5.7.5 `ompd_rel_task_handle`

Summary

This `ompd_rel_task_handle` function releases a task handle.

Format

```
ompd_rc_t ompd_rel_task_handle(  
    ompd_task_handle_t *task_handle  
);
```

Description

Task handles are opaque so tools cannot release them directly. Instead, when a tool is finished with a task handle it must use the `ompd_rel_task_handle` function to release it.

Description of Arguments

The *task_handle* argument is an opaque task handle to be released.

Cross References

- `ompd_task_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_rc_t` type, see Section 5.3.12 on page 543.

5.5.7.6 `ompd_task_handle_compare`

Summary

The `ompd_task_handle_compare` function compares task handles.

Format

```
ompd_rc_t ompd_task_handle_compare(  
    ompd_task_handle_t *task_handle_1,  
    ompd_task_handle_t *task_handle_2,  
    int *cmp_value  
);
```

Description

The internal structure of task handles is opaque so tools cannot directly determine if handles at two different addresses refer to the same underlying task. The **ompd_task_handle_compare** function compares task handles. After a successful call to **ompd_task_handle_compare**, the value of the location to which *cmp_value* points is a signed integer that indicates how the underlying tasks compare: a value less than, equal to, or greater than 0 indicates that the task that corresponds to *task_handle_1* is, respectively, less than, equal to, or greater than the task that corresponds to *task_handle_2*. The means by which task handles are ordered is implementation defined.

Description of Arguments

The *task_handle_1* and *task_handle_2* arguments are opaque handles that correspond to tasks. On return, the *cmp_value* argument points to a location in which a signed integer value indicates how the underlying tasks compare.

Cross References

- **ompd_task_handle_t** type, see Section 5.3.8 on page 540.
- **ompd_rc_t** type, see Section 5.3.12 on page 543.

5.5.7.7 ompd_get_task_function

Summary

This **ompd_get_task_function** function returns the entry point of the code that corresponds to the body of a task.

Format

```
ompd_rc_t ompd_get_task_function (  
    ompd_task_handle_t *task_handle,  
    ompd_address_t *entry_point  
);
```

Description

The **ompd_get_task_function** function returns the entry point of the code that corresponds to the body of code that the task executes.

Description of Arguments

The *task_handle* argument is an opaque handle that selects the task on which to operate. On return, the *entry_point* argument is set to an address that describes the beginning of application code that executes the task region.

Cross References

- **ompd_address_t** type, see Section 5.3.4 on page 538.
- **ompd_task_handle_t** type, see Section 5.3.8 on page 540.
- **ompd_rc_t** type, see Section 5.3.12 on page 543.

5.5.7.8 ompd_get_task_frame

Summary

The **ompd_get_task_frame** function extracts the frame pointers of a task.

Format

```
ompd_rc_t ompd_get_task_frame (  
    ompd_task_handle_t *task_handle,  
    ompd_frame_info_t *exit_frame,  
    ompd_frame_info_t *enter_frame  
);
```

Description

An OpenMP implementation maintains an `ompt_frame_t` object for every implicit or explicit task. The `ompd_get_task_frame` function extracts the *enter_frame* and *exit_frame* fields of the `ompt_frame_t` object of the task that *task_handle* identifies.

Description of Arguments

The *task_handle* argument specifies an OpenMP task. On return, the *exit_frame* argument points to an `ompd_frame_info_t` object that has the frame information with the same semantics as the *exit_frame* field in the `ompt_frame_t` object that is associated with the specified task. On return, the *enter_frame* argument points to an `ompd_frame_info_t` object that has the frame information with the same semantics as the *enter_frame* field in the `ompt_frame_t` object that is associated with the specified task.

Cross References

- `ompt_frame_t` type, see Section 4.4.4.27 on page 454.
- `ompd_address_t` type, see Section 5.3.4 on page 538.
- `ompd_frame_info_t` type, see Section 5.3.5 on page 538.
- `ompd_task_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_rc_t` type, see Section 5.3.12 on page 543.

5.5.7.9 `ompd_enumerate_states`

Summary

The `ompd_enumerate_states` function enumerates thread states that an OpenMP implementation supports.

Format

```
ompd_rc_t ompd_enumerate_states (  
    ompd_address_space_handle_t *address_space_handle,  
    ompd_word_t current_state,  
    ompd_word_t *next_state,  
    const char **next_state_name,  
    ompd_word_t *more_enums  
);
```

Description

An OpenMP implementation may support only a subset of the states that the `ompt_state_t` enumeration type defines. In addition, an OpenMP implementation may support implementation-specific states. The `ompd_enumerate_states` call enables a tool to enumerate the thread states that an OpenMP implementation supports.

When the *current_state* argument is a thread state that an OpenMP implementation supports, the call assigns the value and string name of the next thread state in the enumeration to the locations to which the *next_state* and *next_state_name* arguments point.

On return, the third-party tool owns the *next_state_name* string. The OMPD library allocates storage for the string with the memory allocation callback that the tool provides. The tool is responsible for releasing the memory.

On return, the location to which the *more_enums* argument points has the value 1 whenever one or more states are left in the enumeration. On return, the location to which the *more_enums* argument points has the value 0 when *current_state* is the last state in the enumeration.

Description of Arguments

The *address_space_handle* argument identifies the address space. The *current_state* argument must be a thread state that the OpenMP implementation supports. To begin enumerating the supported states, a tool should pass `ompt_state_undefined` as the value of *current_state*. Subsequent calls to `ompd_enumerate_states` by the tool should pass the value that the call returned in the *next_state* argument. On return, the *next_state* argument points to an integer with the value of the next state in the enumeration. On return, the *next_state_name* argument points to a character string that describes the next state. On return, the *more_enums* argument points to an integer with a value of 1 when more states are left to enumerate and a value of 0 when no more states are left.

Constraints on Arguments

Any string that is returned through the *next_state_name* argument must be immutable and defined for the lifetime of program execution.

Cross References

- `ompt_state_t` type, see Section [4.4.4.26](#) on page [452](#).
- `ompd_address_space_handle_t` type, see Section [5.3.8](#) on page [540](#).
- `ompd_rc_t` type, see Section [5.3.12](#) on page [543](#).

1 5.5.7.10 ompd_get_state

2 Summary

3 The **ompd_get_state** function obtains the state of a thread.

4 Format

```
5      ompd_rc_t ompd_get_state (  
6          ompd_thread_handle_t *thread_handle,  
7          ompd_word_t *state,  
8          ompt_wait_id_t *wait_id  
9      );
```

10 Description

11 The **ompd_get_state** function returns the state of an OpenMP thread.

12 Description of Arguments

13 The *thread_handle* argument identifies the thread. The *state* argument represents the state of that
14 thread as represented by a value that **ompd_enumerate_states** returns. On return, if the
15 *wait_id* argument is non-null then it points to a handle that corresponds to the *wait_id* wait
16 identifier of the thread. If the thread state is not one of the specified wait states, the value to which
17 *wait_id* points is undefined.

18 Cross References

- 19 • **ompd_wait_id_t** type, see Section 5.3.2 on page 537.
- 20 • **ompd_thread_handle_t** type, see Section 5.3.8 on page 540.
- 21 • **ompd_rc_t** type, see Section 5.3.12 on page 543.
- 22 • **ompd_enumerate_states** call, see Section 5.5.7.9 on page 583.

1 5.5.8 Display Control Variables

2 5.5.8.1 ompd_get_display_control_vars

3 Summary

4 The **ompd_get_display_control_vars** function returns a list of name/value pairs for
5 OpenMP control variables.

6 Format

```
7      ompd_rc_t ompd_get_display_control_vars (  
8          ompd_address_space_handle_t *address_space_handle,  
9          const char * const **control_vars  
10     );
```

11 Description

12 The **ompd_get_display_control_vars** function returns a NULL-terminated vector of
13 NULL-terminated strings of name/value pairs of control variables that have user controllable
14 settings and are important to the operation or performance of an OpenMP runtime system. The
15 control variables that this interface exposes include all OpenMP environment variables, settings
16 that may come from vendor or platform-specific environment variables, and other settings that
17 affect the operation or functioning of an OpenMP runtime.

18 The format of the strings is **name=a string**.

19 On return, the third-party tool owns the vector and the strings. The OMP library must satisfy the
20 termination constraints; it may use static or dynamic memory for the vector and/or the strings and is
21 unconstrained in how it arranges them in memory. If it uses dynamic memory then the OMPD
22 library must use the allocate callback that the tool provides to **ompd_initialize**. The tool must
23 use **ompd_rel_display_control_vars()** to release the vector and the strings.

24 Description of Arguments

25 The *address_space_handle* argument identifies the address space. On return, the *control_vars*
26 argument points to the vector of display control variables.

Cross References

- `ompd_address_space_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_rc_t` type, see Section 5.3.12 on page 543.
- `ompd_initialize` call, see Section 5.5.1.1 on page 558.
- `ompd_rel_display_control_vars` type, see Section 5.5.8.2 on page 587.

5.5.8.2 `ompd_rel_display_control_vars`

Summary

The `ompd_rel_display_control_vars` releases a list of name/value pairs of OpenMP control variables previously acquired with `ompd_get_display_control_vars`.

Format

```
ompd_rc_t ompd_rel_display_control_vars (  
    const char * const **control_vars  
);
```

Description

The third-party tool owns the vector and strings that `ompd_get_display_control_vars` returns. The tool must call `ompd_rel_display_control_vars` to release the vector and the strings.

Description of Arguments

The `control_vars` argument is the vector of display control variables to be released.

Cross References

- `ompd_rc_t` type, see Section 5.3.12 on page 543.
- `ompd_get_display_control_vars` call, see Section 5.5.8.1 on page 586.

1 5.5.9 Accessing Scope-Specific Information

2 5.5.9.1 ompd_enumerate_icvs

3 Summary

4 The `ompd_enumerate_icvs` function enumerates ICVs.

5 Format

```
6      ompd_rc_t ompd_enumerate_icvs (  
7          ompd_address_space_handle_t *handle,  
8          ompd_icv_id_t current,  
9          ompd_icv_id_t *next_id,  
10         const char **next_icv_name,  
11         ompd_scope_t *next_scope,  
12         int *more  
13     );
```

14 Description

15 In addition to the ICVs listed in Table 2.1, an OpenMP implementation must support the OMPD
16 specific ICVs listed in Table 5.2. An OpenMP implementation may support additional
17 implementation specific variables. An implementation may store ICVs in a different scope than
18 Table 2.3 indicates. The `ompd_enumerate_icvs` function enables a tool to enumerate the
19 ICVs that an OpenMP implementation supports and their related scopes.

20 When the *current* argument is set to the identifier of a supported ICV, `ompd_enumerate_icvs`
21 assigns the value, string name, and scope of the next ICV in the enumeration to the locations to
22 which the *next_id*, *next_icv_name*, and *next_scope* arguments point. On return, the third-party tool
23 owns the *next_icv_name* string. The OMPD library uses the memory allocation callback that the
24 tool provides to allocate the string storage; the tool is responsible for releasing the memory.

25 On return, the location to which the *more* argument points has the value of 1 whenever one or more
26 ICV are left in the enumeration. on return, that location has the value 0 when *current* is the last
27 ICV in the enumeration.

Description of Arguments

The *address_space_handle* argument identifies the address space. The *current* argument must be an ICV that the OpenMP implementation supports. To begin enumerating the ICVs, a tool should pass **ompd_icv_undefined** as the value of *current*. Subsequent calls to **ompd_enumerate_icvs** should pass the value returned by the call in the *next_id* output argument. On return, the *next_id* argument points to an integer with the value of the ID of the next ICV in the enumeration. On return, the *next_icv* argument points to a character string with the name of the next ICV. On return, the *next_scope* argument points to the scope enum value of the scope of the next ICV. On return, the *more_enums* argument points to an integer with the value of 1 when more ICVs are left to enumerate and the value of 0 when no more ICVs are left.

Constraints on Arguments

Any string that *next_icv* returns must be immutable and defined for the lifetime of a program execution.

TABLE 5.2: OMPD-specific ICVs

Variable	Scope	Meaning
<i>ompd-num-procs-var</i>	device	return value of omp_get_num_procs() when executed on this device
<i>ompd-thread-num-var</i>	task	return value of omp_get_thread_num() when executed in this task
<i>ompd-final-var</i>	task	return value of omp_in_final() when executed in this task
<i>ompd-implicit-var</i>	task	the task is an implicit task
<i>ompd-team-size-var</i>	team	return value of omp_get_num_threads() when executed in this team

Cross References

- **ompd_address_space_handle_t** type, see Section 5.3.8 on page 540.
- **ompd_scope_t** type, see Section 5.3.9 on page 541.
- **ompd_icv_id_t** type, see Section 5.3.10 on page 542.
- **ompd_rc_t** type, see Section 5.3.12 on page 543.

1 5.5.9.2 ompd_get_icv_from_scope

2 Summary

3 The `ompd_get_icv_from_scope` function returns the value of an ICV.

4 Format

```
5      ompd_rc_t ompd_get_icv_from_scope (  
6          void *handle,  
7          ompd_scope_t scope,  
8          ompd_icv_id_t icv_id,  
9          ompd_word_t *icv_value  
10     );
```

11 Description

12 The `ompd_get_icv_from_scope` function provides access to the ICVs that
13 `ompd_enumerate_icvs` identifies.

14 Description of Arguments

15 The *handle* argument provides an OpenMP scope handle. The *scope* argument specifies the kind of
16 scope provided in *handle*. The *icv_id* argument specifies the ID of the requested ICV. On return,
17 the *icv_value* argument points to a location with the value of the requested ICV.

18 Constraints on Arguments

19 If the ICV cannot be represented by an integer type value then the function returns
20 `ompd_rc_incompatible`.

21 The provided *handle* must match the *scope* as defined in Section 5.3.10 on page 542.

22 The provided *scope* must match the scope for *icv_id* as requested by `ompd_enumerate_icvs`.

Cross References

- `ompd_address_space_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_thread_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_parallel_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_task_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_scope_t` type, see Section 5.3.9 on page 541.
- `ompd_icv_id_t` type, see Section 5.3.10 on page 542.
- `ompd_rc_t` type, see Section 5.3.12 on page 543.
- `ompd_enumerate_icvs`, see Section 5.5.9.1 on page 588.

5.5.9.3 `ompd_get_icv_string_from_scope`

Summary

The `ompd_get_icv_string_from_scope` function returns the value of an ICV.

Format

```
ompd_rc_t ompd_get_icv_string_from_scope (  
    void *handle,  
    ompd_scope_t scope,  
    ompd_icv_id_t icv_id,  
    const char **icv_string  
);
```

Description

The `ompd_get_icv_string_from_scope` function provides access to the ICVs that `ompd_enumerate_icvs` identifies.

Description of Arguments

The *handle* argument provides an OpenMP scope handle. The *scope* argument specifies the kind of scope provided in *handle*. The *icv_id* argument specifies the ID of the requested ICV. On return, the *icv_string* argument points to a string representation of the requested ICV.

On return, the third-party tool owns the *icv_string* string. The OMPD library allocates the string storage with the memory allocation callback that the tool provides. The tool is responsible for releasing the memory.

Constraints on Arguments

The provided *handle* must match the *scope* as defined in Section 5.3.10 on page 542.

The provided *scope* must match the scope for *icv_id* as requested by `ompd_enumerate_icvs`.

Cross References

- `ompd_address_space_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_thread_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_parallel_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_task_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_scope_t` type, see Section 5.3.9 on page 541.
- `ompd_icv_id_t` type, see Section 5.3.10 on page 542.
- `ompd_rc_t` type, see Section 5.3.12 on page 543.
- `ompd_enumerate_icvs`, see Section 5.5.9.1 on page 588.

5.5.9.4 `ompd_get_tool_data`

Summary

The `ompd_get_tool_data` function provides access to the OMPT data variable stored for each OpenMP scope.

Format

```
ompd_rc_t ompd_get_tool_data(  
    void* handle,  
    ompd_scope_t scope,  
    ompd_word_t *value,  
    ompd_address_t *ptr  
);
```

C

C

Description

The `ompd_get_tool_data` function provides access to the OMPT tool data stored for each scope. If the runtime library does not support OMPT then the function returns `ompd_rc_unsupported`.

Description of Arguments

The *handle* argument provides an OpenMP scope handle. The *scope* argument specifies the kind of scope provided in *handle*. On return, the *value* argument points to the *value* field of the `ompt_data_t` union stored for the selected scope. On return, the *ptr* argument points to the *ptr* field of the `ompt_data_t` union stored for the selected scope.

Cross References

- `ompt_data_t` type, see Section 4.4.4.4 on page 440.
- `ompd_address_space_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_thread_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_parallel_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_task_handle_t` type, see Section 5.3.8 on page 540.
- `ompd_scope_t` type, see Section 5.3.9 on page 541.
- `ompd_rc_t` type, see Section 5.3.12 on page 543.

1 5.6 Runtime Entry Points for OMPD

2 The OpenMP implementation must define several entry point symbols through which execution
3 must pass when particular events occur *and* data collection for OMPD is enabled. A tool can enable
4 notification of an event by setting a breakpoint at the address of the entry point symbol.

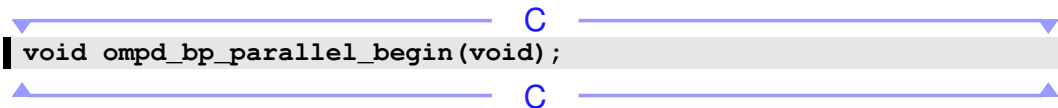
5 Entry point symbols have external **C** linkage and do not require demangling or other
6 transformations to look up their names to obtain the address in the OpenMP program. While each
7 entry point symbol conceptually has a function type signature, it may not be a function. It may be a
8 labeled location

9 5.6.1 Beginning Parallel Regions

10 Summary

11 Before starting the execution of an OpenMP parallel region, the implementation executes
12 **ompd_bp_parallel_begin**.

13 Format

14  **void ompd_bp_parallel_begin(void);**

The diagram shows a blue double-headed arrow above the function signature, with a 'C' in the center, indicating C linkage. A similar arrow is below the signature.

15 Description

16 The OpenMP implementation must execute **ompd_bp_parallel_begin** at every
17 *parallel-begin* event. At the point that the implementation reaches
18 **ompd_bp_parallel_begin**, the binding for **ompd_get_curr_parallel_handle** is the
19 parallel region that is beginning and the binding for **ompd_get_curr_task_handle** is the
20 task that encountered the **parallel** construct.

21 Cross References

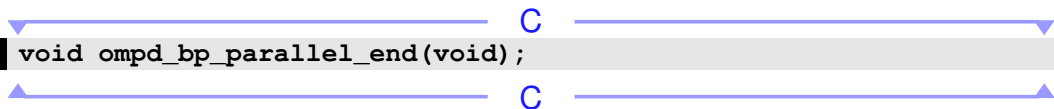
- 22 • **parallel** construct, see Section 2.6 on page 74.
- 23 • **ompd_get_curr_parallel_handle**, see Section 5.5.6.1 on page 571.
- 24 • **ompd_get_curr_task_handle**, see Section 5.5.7.1 on page 576.

1 5.6.2 Ending Parallel Regions

2 Summary

3 After finishing the execution of an OpenMP parallel region, the implementation executes
4 `ompd_bp_parallel_end`.

5 Format

6  `void ompd_bp_parallel_end(void);`

7 Description

8 The OpenMP implementation must execute `ompd_bp_parallel_end` at every *parallel-end*
9 event. At the point that the implementation reaches `ompd_bp_parallel_end`, the binding for
10 `ompd_get_curr_parallel_handle` is the `parallel` region that is ending and the binding
11 for `ompd_get_curr_task_handle` is the task that encountered the `parallel` construct.
12 After execution of `ompd_bp_parallel_end`, any *parallel_handle* that was acquired for the
13 `parallel` region is invalid and should be released.

14 Cross References



- 15 • `parallel` construct, see Section [2.6](#) on page [74](#).
- 16 • `ompd_get_curr_parallel_handle`, see Section [5.5.6.1](#) on page [571](#).
- 17 • `ompd_rel_parallel_handle`, see Section [5.5.6.4](#) on page [574](#).
- 18 • `ompd_get_curr_task_handle`, see Section [5.5.7.1](#) on page [576](#).

19 5.6.3 Beginning Task Regions

20 Summary

21 Before starting the execution of an OpenMP task region, the implementation executes
22 `ompd_bp_task_begin`.

Format

 
void ompd_bp_task_begin(void);

Description

The OpenMP implementation must execute **ompd_bp_task_begin** immediately before starting execution of a *structured-block* that is associated with a non-merged task. At the point that the implementation reaches **ompd_bp_task_begin**, the binding for **ompd_get_curr_task_handle** is the task that is scheduled to execute.

Cross References

- **ompd_get_curr_task_handle**, see Section [5.5.7.1](#) on page [576](#).

5.6.4 Ending Task Regions

Summary

After finishing the execution of an OpenMP task region, the implementation executes **ompd_bp_task_end**.

Format

 
void ompd_bp_task_end(void);

Description

The OpenMP implementation must execute **ompd_bp_task_end** immediately after completion of a *structured-block* that is associated with a non-merged task. At the point that the implementation reaches **ompd_bp_task_end**, the binding for **ompd_get_curr_task_handle** is the task that finished execution. After execution of **ompd_bp_task_end**, any *task_handle* that was acquired for the task region is invalid and should be released.

Cross References

- `ompd_get_curr_task_handle`, see Section 5.5.7.1 on page 576.
- `ompd_rel_task_handle`, see Section 5.5.7.5 on page 580.

5.6.5 Beginning OpenMP Threads

Summary

When starting an OpenMP thread, the implementation executes `ompd_bp_thread_begin`.

Format

C

```
void ompd_bp_thread_begin(void);
```

C

Description

The OpenMP implementation must execute `ompd_bp_thread_begin` at every *native-thread-begin* and *initial-thread-begin* event. This execution occurs before the thread starts the execution of any OpenMP region.

Cross References

- `parallel` construct, see Section 2.6 on page 74.
- Initial task, see Section 2.10.5 on page 148.

5.6.6 Ending OpenMP Threads

Summary

When terminating an OpenMP thread, the implementation executes `ompd_bp_thread_end`.

Format

C

```
void ompd_bp_thread_end(void);
```

C

Description

The OpenMP implementation must execute **ompd_bp_thread_end** at every *native-thread-end* and the *initial-thread-end* event. This execution occurs after the thread completes the execution of all OpenMP regions. After executing **ompd_bp_thread_end**, any *thread_handle* that was acquired for this thread is invalid and should be released.

Cross References

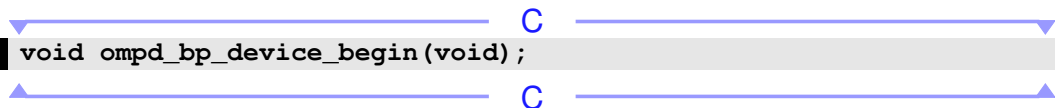
- **parallel** construct, see Section 2.6 on page 74.
- Initial task, see Section 2.10.5 on page 148.
- **ompd_rel_thread_handle**, see Section 5.5.5.3 on page 569.

5.6.7 Initializing OpenMP Devices

Summary

The OpenMP implementation must execute **ompd_bp_device_begin** at every *device-initialize* event.

Format



A diagram showing the C language signature for the function `void ompd_bp_device_begin(void);`. The signature is enclosed in a light gray box. Above and below the box are blue horizontal lines with downward-pointing triangles on the left and upward-pointing triangles on the right. A blue 'C' is positioned to the right of each line, indicating the C language.

```
void ompd_bp_device_begin(void);
```

Description

When initializing a device for execution of a **target** region, the implementation must execute **ompd_bp_device_begin**. This execution occurs before the work associated with any OpenMP region executes on the device.

Cross References



- Device Initialization, see Section 2.12.1 on page 160.

1 5.6.8 Finalizing OpenMP Devices

2 Summary

3 When terminating an OpenMP thread, the implementation executes `ompd_bp_device_end`.

4 Format

5  `void ompd_bp_device_end(void);` 

6 Description

7 The OpenMP implementation must execute `ompd_bp_device_end` at every *device-finalize*
8 event. This execution occurs after the thread executes all OpenMP regions. After execution of
9 `ompd_bp_device_end`, any *address_space_handle* that was acquired for this device is invalid
10 and should be released.

11 Cross References

- 12 • Device Initialization, see Section [2.12.1](#) on page [160](#).
- 13 • `ompd_rel_address_space_handle`, see Section [5.5.2.3](#) on page [564](#).

This page intentionally left blank

CHAPTER 6

Environment Variables

This chapter describes the OpenMP environment variables that specify the settings of the ICVs that affect the execution of OpenMP programs (see Section 2.5 on page 63). The names of the environment variables must be upper case. The values assigned to the environment variables are case insensitive and may have leading and trailing white space. Modifications to the environment variables after the program has started, even if modified by the program itself, are ignored by the OpenMP implementation. However, the settings of some of the ICVs can be modified during the execution of the OpenMP program by the use of the appropriate directive clauses or OpenMP API routines.

The following examples demonstrate how the OpenMP environment variables can be set in different environments:

- csh-like shells:

```
setenv OMP_SCHEDULE "dynamic"
```

- bash-like shells:

```
export OMP_SCHEDULE="dynamic"
```

- Windows Command Line:

```
set OMP_SCHEDULE=dynamic
```

6.1 OMP_SCHEDULE

The **OMP_SCHEDULE** environment variable controls the schedule kind and chunk size of all loop directives that have the schedule kind **runtime**, by setting the value of the *run-sched-var* ICV.

The value of this environment variable takes the form:

[modifier:]kind[, chunk]

where

- *modifier* is one of **monotonic** or **nonmonotonic**;
- *kind* is one of **static**, **dynamic**, **guided**, or **auto**;
- *chunk* is an optional positive integer that specifies the chunk size.

If the *modifier* is not present, the *modifier* is set to **monotonic** if *kind* is **static**; for any other *kind* it is set to **nonmonotonic**.

If *chunk* is present, white space may be on either side of the “,”. See Section 2.9.2 on page 101 for a detailed description of the schedule kinds.

The behavior of the program is implementation defined if the value of **OMP_SCHEDULE** does not conform to the above format.

Implementation specific schedules cannot be specified in **OMP_SCHEDULE**. They can only be specified by calling **omp_set_schedule**, described in Section 3.2.12 on page 345.

Examples:

```
setenv OMP_SCHEDULE "guided,4"
setenv OMP_SCHEDULE "dynamic"
setenv OMP_SCHEDULE "nonmonotonic:dynamic,4"
```

Cross References

- *run-sched-var* ICV, see Section 2.5 on page 63.
- Worksharing-Loop construct, see Section 2.9.2 on page 101.
- Parallel worksharing-loop construct, see Section 2.13.1 on page 185.
- **omp_set_schedule** routine, see Section 3.2.12 on page 345.
- **omp_get_schedule** routine, see Section 3.2.13 on page 347.

6.2 OMP_NUM_THREADS

The **OMP_NUM_THREADS** environment variable sets the number of threads to use for **parallel** regions by setting the initial value of the *nthreads-var* ICV. See Section 2.5 on page 63 for a comprehensive set of rules about the interaction between the **OMP_NUM_THREADS** environment variable, the **num_threads** clause, the **omp_set_num_threads** library routine and dynamic

adjustment of threads, and Section 2.6.1 on page 78 for a complete algorithm that describes how the number of threads for a **parallel** region is determined.

The value of this environment variable must be a list of positive integer values. The values of the list set the number of threads to use for **parallel** regions at the corresponding nested levels.

The behavior of the program is implementation defined if any value of the list specified in the **OMP_NUM_THREADS** environment variable leads to a number of threads that is greater than an implementation can support, or if any value is not a positive integer.

Example:

```
setenv OMP_NUM_THREADS 4,3,2
```

Cross References

- *nthreads-var* ICV, see Section 2.5 on page 63.
- **num_threads** clause, see Section 2.6 on page 74.
- **omp_set_num_threads** routine, see Section 3.2.1 on page 334.
- **omp_get_num_threads** routine, see Section 3.2.2 on page 335.
- **omp_get_max_threads** routine, see Section 3.2.3 on page 336.
- **omp_get_team_size** routine, see Section 3.2.20 on page 354.

6.3 OMP_DYNAMIC

The **OMP_DYNAMIC** environment variable controls dynamic adjustment of the number of threads to use for executing **parallel** regions by setting the initial value of the *dyn-var* ICV.

The value of this environment variable must be one of the following:

true | **false**

If the environment variable is set to **true**, the OpenMP implementation may adjust the number of threads to use for executing **parallel** regions in order to optimize the use of system resources. If the environment variable is set to **false**, the dynamic adjustment of the number of threads is disabled. The behavior of the program is implementation defined if the value of **OMP_DYNAMIC** is neither **true** nor **false**.

Example:

```
setenv OMP_DYNAMIC true
```

Cross References

- *dyn-var* ICV, see Section 2.5 on page 63.
- `omp_set_dynamic` routine, see Section 3.2.7 on page 340.
- `omp_get_dynamic` routine, see Section 3.2.8 on page 341.

6.4 OMP_PROC_BIND

The `OMP_PROC_BIND` environment variable sets the initial value of the *bind-var* ICV. The value of this environment variable is either **true**, **false**, or a comma separated list of **master**, **close**, or **spread**. The values of the list set the thread affinity policy to be used for parallel regions at the corresponding nested level.

If the environment variable is set to **false**, the execution environment may move OpenMP threads between OpenMP places, thread affinity is disabled, and `proc_bind` clauses on `parallel` constructs are ignored.

Otherwise, the execution environment should not move OpenMP threads between OpenMP places, thread affinity is enabled, and the initial thread is bound to the first place in the OpenMP place list prior to the first active parallel region.

The behavior of the program is implementation defined if the value in the `OMP_PROC_BIND` environment variable is not **true**, **false**, or a comma separated list of **master**, **close**, or **spread**. The behavior is also implementation defined if an initial thread cannot be bound to the first place in the OpenMP place list.

Examples:

```
setenv OMP_PROC_BIND false
setenv OMP_PROC_BIND "spread, spread, close"
```

Cross References

- *bind-var* ICV, see Section 2.5 on page 63.
- `proc_bind` clause, see Section 2.6.2 on page 80.
- `omp_get_proc_bind` routine, see Section 3.2.23 on page 357.

1 6.5 OMP_PLACES

2 A list of places can be specified in the **OMP_PLACES** environment variable. The
 3 *place-partition-var* ICV obtains its initial value from the **OMP_PLACES** value, and makes the list
 4 available to the execution environment. The value of **OMP_PLACES** can be one of two types of
 5 values: either an abstract name that describes a set of places or an explicit list of places described
 6 by non-negative numbers.

7 The **OMP_PLACES** environment variable can be defined using an explicit ordered list of
 8 comma-separated places. A place is defined by an unordered set of comma-separated non-negative
 9 numbers enclosed by braces. The meaning of the numbers and how the numbering is done are
 10 implementation defined. Generally, the numbers represent the smallest unit of execution exposed by
 11 the execution environment, typically a hardware thread.

12 Intervals may also be used to define places. Intervals can be specified using the *<lower-bound> :*
 13 *<length> : <stride>* notation to represent the following list of numbers: “*<lower-bound>*,
 14 *<lower-bound> + <stride>*, ..., *<lower-bound> + (<length> - 1)*<stride>*.” When *<stride>* is
 15 omitted, a unit stride is assumed. Intervals can specify numbers within a place as well as sequences
 16 of places.

17 An exclusion operator “!” can also be used to exclude the number or place immediately following
 18 the operator.

19 Alternatively, the abstract names listed in Table 6.1 should be understood by the execution and
 20 runtime environment. The precise definitions of the abstract names are implementation defined. An
 21 implementation may also add abstract names as appropriate for the target platform.

TABLE 6.1: Defined Abstract Names for **OMP_PLACES**

Abstract Name	Meaning
threads	Each place corresponds to a single hardware thread on the target machine.
cores	Each place corresponds to a single core (having one or more hardware threads) on the target machine.
sockets	Each place corresponds to a single socket (consisting of one or more cores) on the target machine.

22 The abstract name may be appended by a positive number in parentheses to denote the length of the
 23 place list to be created, that is *abstract_name(num-places)*. When requesting fewer places than
 24 available on the system, the determination of which resources of type *abstract_name* are to be
 25 included in the place list is implementation defined. When requesting more resources than
 26 available, the length of the place list is implementation defined.

27 The behavior of the program is implementation defined when the execution environment cannot
 28 map a numerical value (either explicitly defined or implicitly derived from an interval) within the

OMP_PLACES list to a processor on the target platform, or if it maps to an unavailable processor. The behavior is also implementation defined when the **OMP_PLACES** environment variable is defined using an abstract name.

The following grammar describes the values accepted for the **OMP_PLACES** environment variable.

$$\begin{aligned}
 \langle \text{list} \rangle &\models \langle \text{p-list} \rangle \mid \langle \text{aname} \rangle \\
 \langle \text{p-list} \rangle &\models \langle \text{p-interval} \rangle \mid \langle \text{p-list} \rangle, \langle \text{p-interval} \rangle \\
 \langle \text{p-interval} \rangle &\models \langle \text{place} \rangle : \langle \text{len} \rangle : \langle \text{stride} \rangle \mid \langle \text{place} \rangle : \langle \text{len} \rangle \mid \langle \text{place} \rangle \mid !\langle \text{place} \rangle \\
 \langle \text{place} \rangle &\models \{ \langle \text{res-list} \rangle \} \\
 \langle \text{res-list} \rangle &\models \langle \text{res-interval} \rangle \mid \langle \text{res-list} \rangle, \langle \text{res-interval} \rangle \\
 \langle \text{res-interval} \rangle &\models \langle \text{res} \rangle : \langle \text{num-places} \rangle : \langle \text{stride} \rangle \mid \langle \text{res} \rangle : \langle \text{num-places} \rangle \mid \langle \text{res} \rangle \mid !\langle \text{res} \rangle \\
 \langle \text{aname} \rangle &\models \langle \text{word} \rangle (\langle \text{num-places} \rangle) \mid \langle \text{word} \rangle \\
 \langle \text{word} \rangle &\models \text{sockets} \mid \text{cores} \mid \text{threads} \mid \text{<implementation-defined abstract name>} \\
 \langle \text{res} \rangle &\models \textit{non-negative integer} \\
 \langle \text{num-places} \rangle &\models \textit{positive integer} \\
 \langle \text{stride} \rangle &\models \textit{integer} \\
 \langle \text{len} \rangle &\models \textit{positive integer}
 \end{aligned}$$

Examples:

```

setenv OMP_PLACES threads
setenv OMP_PLACES "threads (4) "
setenv OMP_PLACES
    "{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}"
setenv OMP_PLACES "{0:4},{4:4},{8:4},{12:4}"
setenv OMP_PLACES "{0:4}:4:4"

```

where each of the last three definitions corresponds to the same 4 places including the smallest units of execution exposed by the execution environment numbered, in turn, 0 to 3, 4 to 7, 8 to 11, and 12 to 15.

Cross References

- *place-partition-var*, see Section 2.5 on page 63.
- Controlling OpenMP thread affinity, see Section 2.6.2 on page 80.
- **omp_get_num_places** routine, see Section 3.2.24 on page 358.
- **omp_get_place_num_procs** routine, see Section 3.2.25 on page 359.

- `omp_get_place_proc_ids` routine, see Section 3.2.26 on page 360.
- `omp_get_place_num` routine, see Section 3.2.27 on page 362.
- `omp_get_partition_num_places` routine, see Section 3.2.28 on page 362.
- `omp_get_partition_place_nums` routine, see Section 3.2.29 on page 363.

6.6 OMP_STACKSIZE

The **OMP_STACKSIZE** environment variable controls the size of the stack for threads created by the OpenMP implementation, by setting the value of the *stacksize-var* ICV. The environment variable does not control the size of the stack for an initial thread.

The value of this environment variable takes the form:

size | *size***B** | *size***K** | *size***M** | *size***G**

where:

- *size* is a positive integer that specifies the size of the stack for threads that are created by the OpenMP implementation.
- **B**, **K**, **M**, and **G** are letters that specify whether the given size is in Bytes, Kilobytes (1024 Bytes), Megabytes (1024 Kilobytes), or Gigabytes (1024 Megabytes), respectively. If one of these letters is present, there may be white space between *size* and the letter.

If only *size* is specified and none of **B**, **K**, **M**, or **G** is specified, then *size* is assumed to be in Kilobytes.

The behavior of the program is implementation defined if **OMP_STACKSIZE** does not conform to the above format, or if the implementation cannot provide a stack with the requested size.

Examples:

```
setenv OMP_STACKSIZE 2000500B
setenv OMP_STACKSIZE "3000 k "
setenv OMP_STACKSIZE 10M
setenv OMP_STACKSIZE " 10 M "
setenv OMP_STACKSIZE "20 m "
setenv OMP_STACKSIZE " 1G"
setenv OMP_STACKSIZE 20000
```

Cross References

- *stacksize-var* ICV, see Section 2.5 on page 63.

1 6.7 OMP_WAIT_POLICY

2 The **OMP_WAIT_POLICY** environment variable provides a hint to an OpenMP implementation
3 about the desired behavior of waiting threads by setting the *wait-policy-var* ICV. A compliant
4 OpenMP implementation may or may not abide by the setting of the environment variable.

5 The value of this environment variable must be one of the following:

6 **ACTIVE** | **PASSIVE**

7 The **ACTIVE** value specifies that waiting threads should mostly be active, consuming processor
8 cycles, while waiting. An OpenMP implementation may, for example, make waiting threads spin.

9 The **PASSIVE** value specifies that waiting threads should mostly be passive, not consuming
10 processor cycles, while waiting. For example, an OpenMP implementation may make waiting
11 threads yield the processor to other threads or go to sleep.

12 The details of the **ACTIVE** and **PASSIVE** behaviors are implementation defined.

13 The behavior of the program is implementation defined if the value of **OMP_WAIT_POLICY** is
14 neither **ACTIVE** nor **PASSIVE**.

15 Examples:

```
16 setenv OMP_WAIT_POLICY ACTIVE  
17 setenv OMP_WAIT_POLICY active  
18 setenv OMP_WAIT_POLICY PASSIVE  
19 setenv OMP_WAIT_POLICY passive
```

20 Cross References

- 21 • *wait-policy-var* ICV, see Section 2.5 on page 63.

22 6.8 OMP_MAX_ACTIVE_LEVELS

23 The **OMP_MAX_ACTIVE_LEVELS** environment variable controls the maximum number of nested
24 active **parallel** regions by setting the initial value of the *max-active-levels-var* ICV.

25 The value of this environment variable must be a non-negative integer. The behavior of the
26 program is implementation defined if the requested value of **OMP_MAX_ACTIVE_LEVELS** is
27 greater than the maximum number of nested active parallel levels an implementation can support,
28 or if the value is not a non-negative integer.

Cross References

- *max-active-levels-var* ICV, see Section 2.5 on page 63.
- `omp_set_max_active_levels` routine, see Section 3.2.16 on page 350.
- `omp_get_max_active_levels` routine, see Section 3.2.17 on page 351.

6.9 OMP_NESTED

The `OMP_NESTED` environment variable controls nested parallelism by setting the initial value of the *max-active-levels-var* ICV. If the environment variable is set to **true**, the initial value of *max-active-levels-var* is set to the number of active levels of parallelism supported by the implementation. If the environment variable is set to **false**, the initial value of *max-active-levels-var* is set to 1. The behavior of the program is implementation defined if the value of `OMP_NESTED` is neither **true** nor **false**.

If both the `OMP_NESTED` and `OMP_MAX_ACTIVE_LEVELS` environment variables are set, the value of `OMP_NESTED` is **false**, and the value of `OMP_MAX_ACTIVE_LEVELS` is greater than 1, the behavior is implementation defined. Otherwise, if both environment variables are set then the `OMP_NESTED` environment variable has no effect.

The `OMP_NESTED` environment variable has been deprecated.

Example:

```
setenv OMP_NESTED false
```

Cross References

- *max-active-levels-var* ICV, see Section 2.5 on page 63.
- `omp_set_nested` routine, see Section 3.2.10 on page 343.
- `omp_get_team_size` routine, see Section 3.2.20 on page 354.
- `OMP_MAX_ACTIVE_LEVELS` environment variable, see Section 6.8 on page 608.

1 6.10 OMP_THREAD_LIMIT

2 The **OMP_THREAD_LIMIT** environment variable sets the maximum number of OpenMP threads
3 to use in a contention group by setting the *thread-limit-var* ICV.

4 The value of this environment variable must be a positive integer. The behavior of the program is
5 implementation defined if the requested value of **OMP_THREAD_LIMIT** is greater than the
6 number of threads an implementation can support, or if the value is not a positive integer.

7 Cross References

- 8 • *thread-limit-var* ICV, see Section 2.5 on page 63.
- 9 • **omp_get_thread_limit** routine, see Section 3.2.14 on page 348.

10 6.11 OMP_CANCELLATION

11 The **OMP_CANCELLATION** environment variable sets the initial value of the *cancel-var* ICV.

12 The value of this environment variable must be one of the following:

13 **true** | **false**

14 If set to **true**, the effects of the **cancel** construct and of cancellation points are enabled and
15 cancellation is activated. If set to **false**, cancellation is disabled and the **cancel** construct and
16 cancellation points are effectively ignored. The behavior of the program is implementation defined
17 if **OMP_CANCELLATION** is set to neither **true** nor **false**.

18 Cross References

- 19 • *cancel-var*, see Section 2.5.1 on page 64.
- 20 • **cancel** construct, see Section 2.18.1 on page 263.
- 21 • **cancellation point** construct, see Section 2.18.2 on page 267.
- 22 • **omp_get_cancellation** routine, see Section 3.2.9 on page 342.

6.12 OMP_DISPLAY_ENV

The **OMP_DISPLAY_ENV** environment variable instructs the runtime to display the OpenMP version number and the value of the ICVs associated with the environment variables described in Chapter 6, as *name = value* pairs. The runtime displays this information once, after processing the environment variables and before any user calls to change the ICV values by runtime routines defined in Chapter 3.

The value of the **OMP_DISPLAY_ENV** environment variable may be set to one of these values:

TRUE | FALSE | VERBOSE

The **TRUE** value instructs the runtime to display the OpenMP version number defined by the **_OPENMP** version macro (or the **openmp_version** Fortran parameter) value and the initial ICV values for the environment variables listed in Chapter 6. The **VERBOSE** value indicates that the runtime may also display the values of runtime variables that may be modified by vendor-specific environment variables. The runtime does not display any information when the **OMP_DISPLAY_ENV** environment variable is **FALSE** or undefined. For all values of the environment variable other than **TRUE**, **FALSE**, and **VERBOSE**, the displayed information is unspecified.

The display begins with "OPENMP DISPLAY ENVIRONMENT BEGIN", followed by the **_OPENMP** version macro (or the **openmp_version** Fortran parameter) value and ICV values, in the format *NAME '=' VALUE*. *NAME* corresponds to the macro or environment variable name, optionally prepended by a bracketed *device-type*. *VALUE* corresponds to the value of the macro or ICV associated with this environment variable. Values are enclosed in single quotes. The display is terminated with "OPENMP DISPLAY ENVIRONMENT END".

For the **OMP_NESTED** environment variable, the printed value is *true* if the *max-active-levels-var* ICV is initialized to a value greater than 1; otherwise the printed value is *false*.

Example:

```
% setenv OMP_DISPLAY_ENV TRUE
```

The above example causes an OpenMP implementation to generate output of the following form:

```
OPENMP DISPLAY ENVIRONMENT BEGIN
  _OPENMP=' 201811'
  [host] OMP_SCHEDULE=' GUIDED, 4'
  [host] OMP_NUM_THREADS=' 4, 3, 2'
  [device] OMP_NUM_THREADS=' 2'
  [host,device] OMP_DYNAMIC=' TRUE'
  [host] OMP_PLACES=' {0:4}, {4:4}, {8:4}, {12:4}'
  ...
OPENMP DISPLAY ENVIRONMENT END
```

1 6.13 OMP_DISPLAY_AFFINITY

2 The **OMP_DISPLAY_AFFINITY** environment variable instructs the runtime to display formatted
3 affinity information for all OpenMP threads in the parallel region upon entering the first parallel
4 region and when any change occurs in the information accessible by the format specifiers listed in
5 Table 6.2. If affinity of any thread in a parallel region changes then thread affinity information for
6 all threads in that region is displayed. If the thread affinity for each respective parallel region at
7 each nesting level has already been displayed and the thread affinity has not changed, then the
8 information is not displayed again. There is no specific order in displaying thread affinity
9 information for all threads in the same parallel region.

10 The value of the **OMP_DISPLAY_AFFINITY** environment variable may be set to one of these
11 values:

12 **TRUE | FALSE**

13 The **TRUE** value instructs the runtime to display the OpenMP thread affinity information, and uses
14 the format setting defined in the *affinity-format-var* ICV.

15 The runtime does not display the OpenMP thread affinity information when the value of the
16 **OMP_DISPLAY_AFFINITY** environment variable is **FALSE** or undefined. For all values of the
17 environment variable other than **TRUE** or **FALSE**, the display action is implementation defined.

18 Example:

19 **setenv OMP_DISPLAY_AFFINITY TRUE**

20 The above example causes an OpenMP implementation to display OpenMP thread affinity
21 information during execution of the program, in a format given by the *affinity-format-var* ICV. The
22 following is a sample output:

23 **nesting_level= 1, thread_num= 0, thread_affinity= 0,1**
24 **nesting_level= 1, thread_num= 1, thread_affinity= 2,3**

25 Cross References

- 26 • Controlling OpenMP thread affinity, see Section 2.6.2 on page 80.
- 27 • **omp_set_affinity_format** routine, see Section 3.2.30 on page 364.
- 28 • **omp_get_affinity_format** routine, see Section 3.2.31 on page 366.
- 29 • **omp_display_affinity** routine, see Section 3.2.32 on page 367.
- 30 • **omp_capture_affinity** routine, see Section 3.2.33 on page 368.
- 31 • **OMP_AFFINITY_FORMAT** environment variable, see Section 6.14 on page 613.

1 6.14 OMP_AFFINITY_FORMAT

2 The **OMP_AFFINITY_FORMAT** environment variable sets the initial value of the
 3 *affinity-format-var* ICV which defines the format when displaying OpenMP thread affinity
 4 information.

5 The value of this environment variable is a character string that may contain as substrings one or
 6 more field specifiers, in addition to other characters. The format of each field specifier is

7 **%[[[0].] size] type**

8 where an individual field specifier must contain the percent symbol (%) and a type. The type can be
 9 a single character short name or its corresponding long name delimited with curly braces, such as
 10 %n or %{thread_num}. A literal percent is specified as %%. Field specifiers can be provided in
 11 any order.

12 The 0 modifier indicates whether or not to add leading zeros to the output, following any indication
 13 of sign or base. The . modifier indicates the output should be right justified when *size* is specified.
 14 By default, output is left justified. The minimum field length is *size*, which is a decimal digit string
 15 with a non-zero first digit. If no *size* is specified, the actual length needed to print the field will be
 16 used. If the 0 modifier is used with *type* of **A**, **{thread_affinity}**, **H**, **{host}**, or a type that
 17 is not printed as a number, the result is unspecified. Any other characters in the format string that
 18 are not part of a field specifier will be included literally in the output.

TABLE 6.2: Available Field Types for Formatting OpenMP Thread Affinity Information

Short Name	Long Name	Meaning
t	team_num	The value returned by <code>omp_get_team_num()</code> .
T	num_teams	The value returned by <code>omp_get_num_teams()</code> .
L	nesting_level	The value returned by <code>omp_get_level()</code> .
n	thread_num	The value returned by <code>omp_get_thread_num()</code> .
N	num_threads	The value returned by <code>omp_get_num_threads()</code> .
a	ancestor_tnum	The value returned by <code>omp_get_ancestor_thread_num(level)</code> , where <i>level</i> is <code>omp_get_level()</code> minus 1.

table continued on next page

table continued from previous page

Short Name	Long Name	Meaning
H	host	The name for the host machine on which the OpenMP program is running.
P	process_id	The process identifier used by the implementation.
i	native_thread_id	The native thread identifier used by the implementation.
A	thread_affinity	The list of numerical identifiers, in the format of a comma-separated list of integers or integer ranges, that represent processors on which a thread may execute, subject to OpenMP thread affinity control and/or other external affinity mechanisms.

Implementations may define additional field types. If an implementation does not have information for a field type, "undefined" is printed for this field when displaying the OpenMP thread affinity information.

Example:

```
setenv OMP_AFFINITY_FORMAT
      "Thread Affinity: %0.3L %.8n %.15{thread_affinity} %.12H"
```

The above example causes an OpenMP implementation to display OpenMP thread affinity information in the following form:

Thread Affinity: 001	0	0-1,16-17	nid003
Thread Affinity: 001	1	2-3,18-19	nid003

Cross References

- Controlling OpenMP thread affinity, see Section 2.6.2 on page 80.
- `omp_set_affinity_format` routine, see Section 3.2.30 on page 364.
- `omp_get_affinity_format` routine, see Section 3.2.31 on page 366.
- `omp_display_affinity` routine, see Section 3.2.32 on page 367.
- `omp_capture_affinity` routine, see Section 3.2.33 on page 368.
- `OMP_DISPLAY_AFFINITY` environment variable, see Section 6.13 on page 612.

1 6.15 OMP_DEFAULT_DEVICE

2 The **OMP_DEFAULT_DEVICE** environment variable sets the device number to use in device
3 constructs by setting the initial value of the *default-device-var* ICV.

4 The value of this environment variable must be a non-negative integer value.

5 Cross References

- 6 • *default-device-var* ICV, see Section 2.5 on page 63.
- 7 • device directives, Section 2.12 on page 160.

8 6.16 OMP_MAX_TASK_PRIORITY

9 The **OMP_MAX_TASK_PRIORITY** environment variable controls the use of task priorities by
10 setting the initial value of the *max-task-priority-var* ICV. The value of this environment variable
11 must be a non-negative integer.

12 Example:

13 **% setenv OMP_MAX_TASK_PRIORITY 20**

14 Cross References

- 15 • *max-task-priority-var* ICV, see Section 2.5 on page 63.
- 16 • Tasking Constructs, see Section 2.10 on page 135.
- 17 • **omp_get_max_task_priority** routine, see Section 3.2.42 on page 377.

18 6.17 OMP_TARGET_OFFLOAD

19 The **OMP_TARGET_OFFLOAD** environment variable sets the initial value of the *target-offload-var*
20 ICV. The value of the **OMP_TARGET_OFFLOAD** environment variable must be one of the
21 following:

22 **MANDATORY | DISABLED | DEFAULT**

The **MANDATORY** value specifies that program execution is terminated if a device construct or device memory routine is encountered and the device is not available or is not supported by the implementation. Support for the **DISABLED** value is implementation defined. If an implementation supports it, the behavior is as if the only device is the host device.

The **DEFAULT** value specifies the default behavior as described in Section 1.3 on page 20.

Example:

```
% setenv OMP_TARGET_OFFLOAD MANDATORY
```

Cross References

- *target-offload-var* ICV, see Section 2.5 on page 63.
- Device Directives, see Section 2.12 on page 160.
- Device Memory Routines, see Section 3.6 on page 397.

6.18 OMP_TOOL

The **OMP_TOOL** environment variable sets the *tool-var* ICV, which controls whether an OpenMP runtime will try to register a first party tool.

The value of this environment variable must be one of the following:

enabled | **disabled**

If **OMP_TOOL** is set to any value other than **enabled** or **disabled**, the behavior is unspecified. If **OMP_TOOL** is not defined, the default value for *tool-var* is **enabled**.

Example:

```
% setenv OMP_TOOL enabled
```

Cross References

- *tool-var* ICV, see Section 2.5 on page 63.
- OMPT Interface, see Chapter 4 on page 419.

1 6.19 OMP_TOOL_LIBRARIES

2 The **OMP_TOOL_LIBRARIES** environment variable sets the *tool-libraries-var* ICV to a list of tool
3 libraries that are considered for use on a device on which an OpenMP implementation is being
4 initialized. The value of this environment variable must be a list of names of dynamically-loadable
5 libraries, separated by an implementation specific, platform typical separator.

6 If the *tool-var* ICV is not enabled, the value of *tool-libraries-var* is ignored. Otherwise, if
7 **ompt_start_tool** is not visible in the address space on a device where OpenMP is being
8 initialized or if **ompt_start_tool** returns **NULL**, an OpenMP implementation will consider
9 libraries in the *tool-libraries-var* list in a left to right order. The OpenMP implementation will
10 search the list for a library that meets two criteria: it can be dynamically loaded on the current
11 device and it defines the symbol **ompt_start_tool**. If an OpenMP implementation finds a
12 suitable library, no further libraries in the list will be considered.

13 Example:

```
14 % setenv OMP_TOOL_LIBRARIES libtoolXY64.so:/usr/local/lib/  
15 libtoolXY32.so
```

16 Cross References

- 17 • *tool-libraries-var* ICV, see Section 2.5 on page 63.
- 18 • OMPT Interface, see Chapter 4 on page 419.
- 19 • **ompt_start_tool** routine, see Section 4.2.1 on page 420.

20 6.20 OMP_DEBUG

21 The **OMP_DEBUG** environment variable sets the *debug-var* ICV, which controls whether an
22 OpenMP runtime collects information that an OMPD library may need to support a tool.

23 The value of this environment variable must be one of the following:

24 **enabled** | **disabled**

25 If **OMP_DEBUG** is set to any value other than **enabled** or **disabled** then the behavior is
26 implementation defined.

27 Example:

```
28 % setenv OMP_DEBUG enabled
```

Cross References

- *debug-var* ICV, see Section 2.5 on page 63.
- OMPD Interface, see Chapter 5 on page 533.
- Enabling the Runtime for OMPD, see Section 5.2.1 on page 534.

6.21 OMP_ALLOCATOR

OMP_ALLOCATOR sets the *def-allocator-var* ICV that specifies the default allocator for allocation calls, directives and clauses that do not specify an allocator. The value of this environment variable is a predefined allocator from Table 2.10 on page 155. The value of this environment variable is not case sensitive.

Cross References

- *def-allocator-var* ICV, see Section 2.5 on page 63.
- Memory allocators, see Section 2.11.2 on page 152.
- **omp_set_default_allocator** routine, see Section 3.7.4 on page 411.
- **omp_get_default_allocator** routine, see Section 3.7.5 on page 412.

APPENDIX A

OpenMP Implementation-Defined Behaviors

This appendix summarizes the behaviors that are described as implementation defined in this API. Each behavior is cross-referenced back to its description in the main specification. An implementation is required to define and to document its behavior in these cases.

- **Processor**: a hardware unit that is implementation defined (see Section 1.2.1 on page 2).
- **Device**: an implementation defined logical execution engine (see Section 1.2.1 on page 2).
- **Device address**: reference to an address in a *device data environment* (see Section 1.2.6 on page 12).
- **Memory model**: the minimum size at which a memory update may also read and write back adjacent variables that are part of another variable (as array or structure elements) is implementation defined but is no larger than required by the base language (see Section 1.4.1 on page 23).
- **requires directive**: support of requirements is implementation defined. All implementation-defined requirements should begin with **ext_** (see Section 2.4 on page 60).
- **Requires directive**: Support for any feature specified by a requirement clause on a **requires** directive is implementation defined (see Section 2.4 on page 60).
- **Internal control variables**: the initial values of *dyn-var*, *nthreads-var*, *run-sched-var*, *def-sched-var*, *bind-var*, *stacksize-var*, *wait-policy-var*, *thread-limit-var*, *max-active-levels-var*, *place-partition-var*, *affinity-format-var*, *default-device-var* and *def-allocator-var* are implementation defined. The method for initializing a target device's internal control variable is implementation defined (see Section 2.5.2 on page 66).
- **OpenMP context**: the accepted *isa-name* values for the *isa* trait, the accepted *arch-name* values for the *arch* trait, and the accepted *extension-name* values for the *extension* trait are implementation defined (see Section 2.3.1 on page 51).

- 1 • **declare variant directive**: whether, for some specific OpenMP context, the prototype of
2 the variant should differ from that of the base function, and if so how it should differ, is
3 implementation defined (see Section 2.3.5 on page 58).
- 4 • **Dynamic adjustment of threads**: providing the ability to adjust the number of threads
5 dynamically is implementation defined. Implementations are allowed to deliver fewer threads
6 (but at least one) than indicated in Algorithm 2.1 even if dynamic adjustment is disabled (see
7 Section 2.6.1 on page 78).
- 8 • **Thread affinity**: For the **close** thread affinity policy, if $T > P$ and P does not divide T evenly,
9 the exact number of threads in a particular place is implementation defined. For the **spread**
10 thread affinity, if $T > P$ and P does not divide T evenly, the exact number of threads in a
11 particular subpartition is implementation defined. The determination of whether the affinity
12 request can be fulfilled is implementation defined. If not, the mapping of threads in the team to
13 places is implementation defined (see Section 2.6.2 on page 80).
- 14 • **teams construct**: the number of teams that are created is implementation defined but less than
15 or equal to the value of the **num_teams** clause if specified. The maximum number of threads
16 that participate in the contention group that each team initiates is implementation defined but less
17 than or equal to the value of the **thread_limit** clause if specified. The assignment of the
18 initial threads to places and the values of the *place-partition-var* and *default-device-var* ICVs for
19 each initial thread are implementation defined (see Section 2.7 on page 82).
- 20 • **sections construct**: the method of scheduling the structured blocks among threads in the
21 team is implementation defined (see Section 2.8.1 on page 86).
- 22 • **single construct**: the method of choosing a thread to execute the structured block is
23 implementation defined (see Section 2.8.2 on page 89).
- 24 • **Worksharing-Loop directive**: the integer type (or kind, for Fortran) used to compute the
25 iteration count of a collapsed loop is implementation defined. The effect of the
26 **schedule(runtime)** clause when the *run-sched-var* ICV is set to **auto** is implementation
27 defined. The value of *simd_width* for the **simd** schedule modifier is implementation defined (see
28 Section 2.9.2 on page 101).
- 29 • **simd construct**: the integer type (or kind, for Fortran) used to compute the iteration count for
30 the collapsed loop is implementation defined. The number of iterations that are executed
31 concurrently at any given time is implementation defined. If the *alignment* parameter is not
32 specified in the **aligned** clause, the default alignments for the SIMD instructions are
33 implementation defined (see Section 2.9.3.1 on page 110).
- 34 • **declare simd directive**: if the parameter of the **simdlen** clause is not a constant positive
35 integer expression, the number of concurrent arguments for the function is implementation
36 defined. If the *alignment* parameter of the **aligned** clause is not specified, the default
37 alignments for SIMD instructions are implementation defined (see Section 2.9.3.3 on page 116).
- 38 • **distribute construct**: the integer type (or kind, for Fortran) used to compute the iteration
39 count for the collapsed loop is implementation defined. If no **dist_schedule** clause is

specified then the schedule for the **distribute** construct is implementation defined (see Section 2.9.4.1 on page 120).

- **taskloop construct**: The number of loop iterations assigned to a task created from a **taskloop** construct is implementation defined, unless the **grainsize** or **num_tasks** clause is specified. The integer type (or kind, for Fortran) used to compute the iteration count for the collapsed loop is implementation defined (see Section 2.10.2 on page 140).

C++

- **taskloop construct**: For **firstprivate** variables of class type, the number of invocations of copy constructors to perform the initialization is implementation defined (see Section 2.10.2 on page 140).

C++

- **Memory spaces**: The actual storage resource that each memory space defined in Table 2.8 on page 152 represents is implementation defined.
- **Memory allocators**: The minimum partitioning size for partitioning of allocated memory over the storage resources is implementation defined (see Section 2.11.2 on page 152). The default value for the **pool_size** allocator trait is implementation defined (see Table 2.9 on page 153). The associated memory space for each of the predefined **omp_cgroup_mem_alloc**, **omp_pteam_mem_alloc** and **omp_thread_mem_alloc** allocators is implementation defined (see Table 2.10 on page 155).
- **is_device_ptr clause**: Support for pointers created outside of the OpenMP device data management routines is implementation defined (see Section 2.12.5 on page 170).
- **target construct**: the effect of invoking a virtual member function of an object on a device other than the device on which the object was constructed is implementation defined (see Section 2.12.5 on page 170).
- **atomic construct**: a compliant implementation may enforce exclusive access between **atomic** regions that update different storage locations. The circumstances under which this occurs are implementation defined. If the storage location designated by x is not size-aligned (that is, if the byte alignment of x is not a multiple of the size of x), then the behavior of the atomic region is implementation defined (see Section 2.17.7 on page 234).

Fortran

- **Data-sharing attributes**: The data-sharing attributes of dummy arguments without the **VALUE** attribute are implementation-defined if the associated actual argument is shared, except for the conditions specified (see Section 2.19.1.2 on page 273).
- **threadprivate directive**: if the conditions for values of data in the threadprivate objects of threads (other than an initial thread) to persist between two consecutive active parallel regions do not all hold, the allocation status of an allocatable variable in the second region is implementation defined (see Section 2.19.2 on page 274).

- **Runtime library definitions:** it is implementation defined whether the include file `omp_lib.h` or the module `omp_lib` (or both) is provided. It is implementation defined whether any of the OpenMP runtime library routines that take an argument are extended with a generic interface so arguments of different **KIND** type can be accommodated (see Section 3.1 on page 332).

Fortran

- **omp_set_num_threads routine:** if the argument is not a positive integer the behavior is implementation defined (see Section 3.2.1 on page 334).
- **omp_set_schedule routine:** for implementation specific schedule kinds, the values and associated meanings of the second argument are implementation defined (see Section 3.2.12 on page 345).
- **omp_get_supported_active_levels routine:** the number of active levels of parallelism supported by the implementation is implementation defined, but must be greater than 0 (see Section 3.2.15 on page 349).
- **omp_set_max_active_levels routine:** when called from within any explicit `parallel` region the binding thread set (and binding region, if required) for the `omp_set_max_active_levels` region is implementation defined and the behavior is implementation defined. If the argument is not a non-negative integer then the behavior is implementation defined (see Section 3.2.16 on page 350).
- **omp_get_max_active_levels routine:** when called from within any explicit `parallel` region the binding thread set (and binding region, if required) for the `omp_get_max_active_levels` region is implementation defined (see Section 3.2.17 on page 351).
- **omp_get_place_proc_ids routine:** the meaning of the non-negative numerical identifiers returned by the `omp_get_place_proc_ids` routine is implementation defined. The order of the numerical identifiers returned in the array `ids` is implementation defined (see Section 3.2.26 on page 360).
- **omp_set_affinity_format routine:** when called from within any explicit `parallel` region, the binding thread set (and binding region, if required) for the `omp_set_affinity_format` region is implementation defined and the behavior is implementation defined. If the argument does not conform to the specified format then the result is implementation defined (see Section 3.2.30 on page 364).
- **omp_get_affinity_format routine:** when called from within any explicit `parallel` region the binding thread set (and binding region, if required) for the `omp_get_affinity_format` region is implementation defined (see Section 3.2.31 on page 366).
- **omp_display_affinity routine:** if the argument does not conform to the specified format then the result is implementation defined (see Section 3.2.32 on page 367).

- 1 • **omp_capture_affinity routine:** if the *format* argument does not conform to the specified
2 format then the result is implementation defined (see Section 3.2.33 on page 368).
- 3 • **omp_get_initial_device routine:** the value of the device number of the host device is
4 implementation defined (see Section 3.2.41 on page 376).
- 5 • **omp_target_memcpy_rect routine:** the maximum number of dimensions supported is
6 implementation defined, but must be at least three (see Section 3.6.5 on page 402).
- 7 • **ompt_callback_sync_region_wait, ompt_callback_mutex_released,**
8 **ompt_callback_dependences, ompt_callback_task_dependence,**
9 **ompt_callback_work, ompt_callback_master, ompt_callback_target_map,**
10 **ompt_callback_sync_region, ompt_callback_lock_init,**
11 **ompt_callback_lock_destroy, ompt_callback_mutex_acquire,**
12 **ompt_callback_mutex_acquired, ompt_callback_nest_lock,**
13 **ompt_callback_flush, ompt_callback_cancel** and
14 **ompt_callback_dispatch** tool callbacks: if a tool attempts to register a callback with the
15 string name using the runtime entry point **ompt_set_callback**, it is implementation defined
16 whether the registered callback may never or sometimes invoke this callback for the associated
17 events (see Table 4.2 on page 428)
- 18 • **Device tracing:** Whether a target device supports tracing or not is implementation defined; if a
19 target device does not support tracing, a **NULL** may be supplied for the *lookup* function to a
20 tool's device initializer (see Section 4.2.5 on page 427).
- 21 • **ompt_set_trace_ompt and ompt_buffer_get_record_ompt runtime entry**
22 **points:** it is implementation defined whether a device-specific tracing interface will define this
23 runtime entry point, indicating that it can collect traces in OMPT format. The kinds of trace
24 records available for a device is implementation defined (see Section 4.2.5 on page 427).
- 25 • **ompt_callback_target_data_op_t callback type:** it is implementation defined
26 whether in some operations *src_addr* or *dest_addr* might point to an intermediate buffer (see
27 Section 4.5.2.25 on page 488).
- 28 • **ompt_set_callback_t entry point type:** the subset of the associated event in which the
29 callback is invoked is implementation defined (see Section 4.6.1.3 on page 500).
- 30 • **ompt_get_place_proc_ids_t entry point type:** the meaning of the numerical identifiers
31 returned is implementation defined. The order of *ids* returned in the array is implementation
32 defined (see Section 4.6.1.8 on page 505).
- 33 • **ompt_get_partition_place_nums_t entry point type:** the order of the identifiers
34 returned in the array *place_nums* is implementation defined (see Section 4.6.1.10 on page 507).
- 35 • **ompt_get_proc_id_t entry point type:** the meaning of the numerical identifier returned is
36 implementation defined (see Section 4.6.1.11 on page 508).
- 37 • **ompd_callback_print_string_fn_t callback function:** the value of *category* is
38 implementation defined (see Section 5.4.5 on page 556).

- **ompd_parallel_handle_compare operation:** the means by which parallel region handles are ordered is implementation defined (see Section 5.5.6.5 on page 575).
- **ompd_task_handle_compare operation:** the means by which task handles are ordered is implementation defined (see Section 5.5.7.6 on page 580).
- **OMPT thread states:** The set of OMPT thread states supported is implementation defined (see Section 4.4.4.26 on page 452).
- **OMP_SCHEDULE environment variable:** if the value does not conform to the specified format then the result is implementation defined (see Section 6.1 on page 601).
- **OMP_NUM_THREADS environment variable:** if any value of the list specified leads to a number of threads that is greater than the implementation can support, or if any value is not a positive integer, then the result is implementation defined (see Section 6.2 on page 602).
- **OMP_DYNAMIC environment variable:** if the value is neither **true** nor **false** the behavior is implementation defined (see Section 6.3 on page 603).
- **OMP_PROC_BIND environment variable:** if the value is not **true**, **false**, or a comma separated list of **master**, **close**, or **spread**, the behavior is implementation defined. The behavior is also implementation defined if an initial thread cannot be bound to the first place in the OpenMP place list (see Section 6.4 on page 604).
- **OMP_PLACES environment variable:** the meaning of the numbers specified in the environment variable and how the numbering is done are implementation defined. The precise definitions of the abstract names are implementation defined. An implementation may add implementation-defined abstract names as appropriate for the target platform. When creating a place list of n elements by appending the number n to an abstract name, the determination of which resources to include in the place list is implementation defined. When requesting more resources than available, the length of the place list is also implementation defined. The behavior of the program is implementation defined when the execution environment cannot map a numerical value (either explicitly defined or implicitly derived from an interval) within the **OMP_PLACES** list to a processor on the target platform, or if it maps to an unavailable processor. The behavior is also implementation defined when the **OMP_PLACES** environment variable is defined using an abstract name (see Section 6.5 on page 605).
- **OMP_STACKSIZE environment variable:** if the value does not conform to the specified format or the implementation cannot provide a stack of the specified size then the behavior is implementation defined (see Section 6.6 on page 607).
- **OMP_WAIT_POLICY environment variable:** the details of the **ACTIVE** and **PASSIVE** behaviors are implementation defined (see Section 6.7 on page 608).
- **OMP_MAX_ACTIVE_LEVELS environment variable:** if the value is not a non-negative integer or is greater than the number of parallel levels an implementation can support then the behavior is implementation defined (see Section 6.8 on page 608).

- 1 ● **OMP_NESTED environment variable:** if the value is neither **true** nor **false** the behavior is
2 implementation defined (see Section 6.9 on page 609).
- 3 ● **Conflicting OMP_NESTED and OMP_MAX_ACTIVE_LEVELS environment variables:** if
4 both environment variables are set, the value of **OMP_NESTED** is **false**, and the value of
5 **OMP_MAX_ACTIVE_LEVELS** is greater than 1, the behavior is implementation defined (see
6 Section 6.9 on page 609).
- 7 ● **OMP_THREAD_LIMIT environment variable:** if the requested value is greater than the number
8 of threads an implementation can support, or if the value is not a positive integer, the behavior of
9 the program is implementation defined (see Section 6.10 on page 610).
- 10 ● **OMP_DISPLAY_AFFINITY environment variable:** for all values of the environment variables
11 other than **TRUE** or **FALSE**, the display action is implementation defined (see Section 6.13 on
12 page 612).
- 13 ● **OMP_AFFINITY_FORMAT environment variable:** if the value does not conform to the
14 specified format then the result is implementation defined (see Section 6.14 on page 613).
- 15 ● **OMP_TARGET_OFFLOAD environment variable:** the support of **disabled** is
16 implementation defined (see Section 6.17 on page 615).
- 17 ● **OMP_DEBUG environment variable:** if the value is neither **disabled** nor **enabled** the
18 behavior is implementation defined (see Section 6.20 on page 617).

This page intentionally left blank

APPENDIX B

Features History

This appendix summarizes the major changes between OpenMP API versions since version 2.5.

B.1 Deprecated Features

The following features have been deprecated in Version 5.0.

- The *nest-var* ICV, the **OMP_NESTED** environment variable, and the **omp_set_nested** and **omp_get_nested** routines were deprecated.
- Lock hints were renamed to synchronization hints. The following lock hint type and constants were deprecated:
 - the C/C++ type **omp_lock_hint_t** and the Fortran kind **omp_lock_hint_kind**;
 - the constants **omp_lock_hint_none**, **omp_lock_hint_uncontended**, **omp_lock_hint_contended**, **omp_lock_hint_nonspeculative**, and **omp_lock_hint_speculative**.

B.2 Version 4.5 to 5.0 Differences

- The memory model was extended to distinguish different types of flush operations according to specified flush properties (see Section 1.4.4 on page 25) and to define a happens before order based on synchronizing flush operations (see Section 1.4.5 on page 27).

- Various changes throughout the specification were made to provide initial support of C11, C++11, C++14, C++17 and Fortran 2008 (see Section 1.7 on page 31).
- Fortran 2003 is now fully supported (see Section 1.7 on page 31).
- The **requires** directive (see Section 2.4 on page 60) was added to support applications that require implementation-specific features.
- The *target-offload-var* internal control variable (see Section 2.5 on page 63) and the **OMP_TARGET_OFFLOAD** environment variable (see Section 6.17 on page 615) were added to support runtime control of the execution of device constructs.
- Control over whether nested parallelism is enabled or disabled was integrated into the *max-active-levels-var* internal control variable (see Section 2.5.2 on page 66), the default value of which is now implementation defined, unless determined according to the values of the **OMP_NUM_THREADS** (see Section 6.2 on page 602) or **OMP_PROC_BIND** (see Section 6.4 on page 604) environment variables.
- Support for array shaping (see Section 2.1.4 on page 43) and for array sections with non-unit strides in C and C++ (see Section 2.1.5 on page 44) was added to facilitate specification of discontinuous storage and the **target update** construct (see Section 2.12.6 on page 176) and the **depend** clause (see Section 2.17.11 on page 255) were extended to allow the use of shape-operators (see Section 2.1.4 on page 43).
- Iterators (see Section 2.1.6 on page 47) were added to support expressions in a list that expand to multiple expressions.
- The **metadirective** directive (see Section 2.3.4 on page 56) and **declare variant** directive (see Section 2.3.5 on page 58) were added to support selection of directive variants and declared function variants at a callsite, respectively, based on compile-time traits of the enclosing context.
- The **teams** construct (see Section 2.7 on page 82) was extended to support execution on the host device without an enclosing **target** construct (see Section 2.12.5 on page 170).
- The canonical loop form was defined for Fortran and, for all base languages, extended to permit non-rectangular loop nests (see Section 2.9.1 on page 95).
- The *relational-op* in the *canonical loop form* for C/C++ was extended to include **!=** (see Section 2.9.1 on page 95).
- The default loop schedule modifier for worksharing-loop constructs without the **static** schedule and the **ordered** clause was changed to **nonmonotonic** (see Section 2.9.2 on page 101).
- The collapse of associated loops that are imperfectly nested loops was defined for the worksharing-loop (see Section 2.9.2 on page 101), **simd** (see Section 2.9.3.1 on page 110), **taskloop** (see Section 2.10.2 on page 140) and **distribute** (see Section 2.9.4.2 on page 123) constructs.

- The **simd** construct (see Section 2.9.3.1 on page 110) was extended to accept the **if**, **nontemporal** and **order (concurrent)** clauses and to allow the use of **atomic** constructs within it.
- The **loop** construct and the **order (concurrent)** clause were added to support compiler optimization and parallelization of loops for which iterations may execute in any order, including concurrently (see Section 2.9.5 on page 128).
- The **scan** directive (see Section 2.9.6 on page 132) and the **inscan** modifier for the **reduction** clause (see Section 2.19.5.4 on page 300) were added to support inclusive and exclusive scan computations.
- To support task reductions, the **task** (see Section 2.10.1 on page 135) and **target** (see Section 2.12.5 on page 170) constructs were extended to accept the **in_reduction** clause (see Section 2.19.5.6 on page 303), the **taskgroup** construct (see Section 2.17.6 on page 232) was extended to accept the **task_reduction** clause Section 2.19.5.5 on page 303), and the **task** modifier was added to the **reduction** clause (see Section 2.19.5.4 on page 300).
- The **affinity** clause was added to the **task** construct (see Section 2.10.1 on page 135) to support hints that indicate data affinity of explicit tasks.
- The **detach** clause for the **task** construct (see Section 2.10.1 on page 135) and the **omp_fulfill_event** runtime routine (see Section 3.5.1 on page 396) were added to support execution of detachable tasks.
- To support taskloop reductions, the **taskloop** (see Section 2.10.2 on page 140) and **taskloop simd** (see Section 2.10.3 on page 146) constructs were extended to accept the **reduction** (see Section 2.19.5.4 on page 300) and **in_reduction** (see Section 2.19.5.6 on page 303) clauses.
- The **taskloop** construct (see Section 2.10.2 on page 140) was added to the list of constructs that can be canceled by the **cancel** construct (see Section 2.18.1 on page 263)).
- To support mutually exclusive inout sets, a **mutexinoutset dependence-type** was added to the **depend** clause (see Section 2.10.6 on page 149 and Section 2.17.11 on page 255).
- Predefined memory spaces (see Section 2.11.1 on page 152), predefined memory allocators and allocator traits (see Section 2.11.2 on page 152) and directives, clauses (see Section 2.11 on page 152 and API routines (see Section 3.7 on page 406) to use them were added to support different kinds of memories.
- The semantics of the **use_device_ptr** clause for pointer variables was clarified and the **use_device_addr** clause for using the device address of non-pointer variables inside the **target data** construct was added (see Section 2.12.2 on page 161).
- To support reverse offload, the **ancestor** modifier was added to the **device** clause for **target** constructs (see Section 2.12.5 on page 170).

- To reduce programmer effort implicit declare target directives for some functions (C, C++, Fortran) and subroutines (Fortran) were added (see Section 2.12.5 on page 170 and Section 2.12.7 on page 180).
- The **target update** construct (see Section 2.12.6 on page 176) was modified to allow array sections that specify discontinuous storage.
- The **to** and **from** clauses on the **target update** construct (see Section 2.12.6 on page 176), the **depend** clause on task generating constructs (see Section 2.17.11 on page 255), and the **map** clause (see Section 2.19.7.1 on page 315) were extended to allow any lvalue expression as a list item for C/C++.
- Support for nested **declare target** directives was added (see Section 2.12.7 on page 180).
- New combined constructs **master taskloop** (see Section 2.13.7 on page 192), **parallel master** (see Section 2.13.6 on page 191), **parallel master taskloop** (see Section 2.13.9 on page 195), **master taskloop simd** (see Section 2.13.8 on page 194), **parallel master taskloop simd** (see Section 2.13.10 on page 196) were added.
- The **depend** clause was added to the **taskwait** construct (see Section 2.17.5 on page 230).
- To support acquire and release semantics with weak memory ordering, the **acq_rel**, **acquire**, and **release** clauses were added to the **atomic** construct (see Section 2.17.7 on page 234) and **flush** construct (see Section 2.17.8 on page 242), and the memory ordering semantics of implicit flushes on various constructs and runtime routines were clarified (see Section 2.17.8.1 on page 246).
- The **atomic** construct was extended with the **hint** clause (see Section 2.17.7 on page 234).
- The **depend** clause (see Section 2.17.11 on page 255) was extended to support iterators and to support depend objects that can be created with the new **depobj** construct.
- Lock hints were renamed to synchronization hints, and the old names were deprecated (see Section 2.17.12 on page 260).
- To support conditional assignment to lastprivate variables, the **conditional** modifier was added to the **lastprivate** clause (see Section 2.19.4.5 on page 288).
- The description of the **map** clause was modified to clarify the mapping order when multiple *map-types* are specified for a variable or structure members of a variable on the same construct. The *close map-type-modifier* was added as a hint for the runtime to allocate memory close to the target device (see Section 2.19.7.1 on page 315).
- The capability to map C/C++ pointer variables and to assign the address of device memory that is mapped by an array section to them was added. Support for mapping of Fortran pointer and allocatable variables, including pointer and allocatable components of variables, was added (see Section 2.19.7.1 on page 315).
- The **defaultmap** clause (see Section 2.19.7.2 on page 324) was extended to allow selecting the data-mapping or data-sharing attributes for any of the scalar, aggregate, pointer or allocatable

- classes on a per-region basis. Additionally it accepts the **none** parameter to support the requirement that all variables referenced in the construct must be explicitly mapped or privatized.
- The **declare mapper** directive was added to support mapping of data types with direct and indirect members (see Section 2.19.7.3 on page 326).
 - The **omp_set_nested** (see Section 3.2.10 on page 343) and **omp_get_nested** (see Section 3.2.11 on page 344) routines and the **OMP_NESTED** environment variable (see Section 6.9 on page 609) were deprecated.
 - The **omp_get_supported_active_levels** routine was added to query the number of active levels of parallelism supported by the implementation (see Section 3.2.15 on page 349).
 - Runtime routines **omp_set_affinity_format** (see Section 3.2.30 on page 364), **omp_get_affinity_format** (see Section 3.2.31 on page 366), **omp_set_affinity** (see Section 3.2.32 on page 367), and **omp_capture_affinity** (see Section 3.2.33 on page 368) and environment variables **OMP_DISPLAY_AFFINITY** (see Section 6.13 on page 612) and **OMP_AFFINITY_FORMAT** (see Section 6.14 on page 613) were added to provide OpenMP runtime thread affinity information.
 - The **omp_get_device_num** runtime routine (see Section 3.2.37 on page 372) was added to support determination of the device on which a thread is executing.
 - The **omp_pause_resource** and **omp_pause_resource_all** runtime routines were added to allow the runtime to relinquish resources used by OpenMP (see Section 3.2.43 on page 378 and Section 3.2.44 on page 380).
 - Support for a first-party tool interface (see Section 4 on page 419) was added.
 - Support for a third-party tool interface (see Section 5 on page 533) was added.
 - Support for controlling offloading behavior with the **OMP_TARGET_OFFLOAD** environment variable was added (see Section 6.17 on page 615).
 - Stubs for Runtime Library Routines (previously Appendix A) were moved to a separate document.
 - Interface Declarations (previously Appendix B) were moved to a separate document.

B.3 Version 4.0 to 4.5 Differences

- Support for several features of Fortran 2003 was added (see Section 1.7 on page 31 for features that are still not supported).
- A parameter was added to the **ordered** clause of the worksharing-loop construct (see Section 2.9.2 on page 101) and clauses were added to the **ordered** construct (see

Section 2.17.9 on page 250) to support doacross loop nests and use of the **simd** construct on loops with loop-carried backward dependences.

- The **linear** clause was added to the worksharing-loop construct (see Section 2.9.2 on page 101).
- The **simdlen** clause was added to the **simd** construct (see Section 2.9.3.1 on page 110) to support specification of the exact number of iterations desired per SIMD chunk.
- The **priority** clause was added to the **task** construct (see Section 2.10.1 on page 135) to support hints that specify the relative execution priority of explicit tasks. The **omp_get_max_task_priority** routine was added to return the maximum supported priority value (see Section 3.2.42 on page 377) and the **OMP_MAX_TASK_PRIORITY** environment variable was added to control the maximum priority value allowed (see Section 6.16 on page 615).
- Taskloop constructs (see Section 2.10.2 on page 140 and Section 2.10.3 on page 146) were added to support nestable parallel loops that create OpenMP tasks.
- To support interaction with native device implementations, the **use_device_ptr** clause was added to the **target data** construct (see Section 2.12.2 on page 161) and the **is_device_ptr** clause was added to the **target** construct (see Section 2.12.5 on page 170).
- The **nowait** and **depend** clauses were added to the **target** construct (see Section 2.12.5 on page 170) to improve support for asynchronous execution of **target** regions.
- The **private**, **firstprivate** and **defaultmap** clauses were added to the **target** construct (see Section 2.12.5 on page 170).
- The **declare target** directive was extended to allow mapping of global variables to be deferred to specific device executions and to allow an *extended-list* to be specified in C/C++ (see Section 2.12.7 on page 180).
- To support unstructured data mapping for devices, the **target enter data** (see Section 2.12.3 on page 164) and **target exit data** (see Section 2.12.4 on page 166) constructs were added and the **map** clause (see Section 2.19.7.1 on page 315) was updated.
- To support a more complete set of device construct shortcuts, the **target parallel** (see Section 2.13.16 on page 203), **target parallel worksharing-loop** (see Section 2.13.17 on page 205), **target parallel worksharing-loop SIMD** (see Section 2.13.18 on page 206), and **target simd** (see Section 2.13.20 on page 209), combined constructs were added.
- The **if** clause was extended to take a *directive-name-modifier* that allows it to apply to combined constructs (see Section 2.15 on page 220).
- The **hint** clause was added to the **critical** construct (see Section 2.17.1 on page 223).
- The **source** and **sink** dependence types were added to the **depend** clause (see Section 2.17.11 on page 255) to support doacross loop nests.

- The implicit data-sharing attribute for scalar variables in **target** regions was changed to **firstprivate** (see Section 2.19.1.1 on page 270).
- Use of some C++ reference types was allowed in some data sharing attribute clauses (see Section 2.19.4 on page 282).
- Semantics for reductions on C/C++ array sections were added and restrictions on the use of arrays and pointers in reductions were removed (see Section 2.19.5.4 on page 300).
- The **ref**, **val**, and **uval** modifiers were added to the **linear** clause (see Section 2.19.4.6 on page 290).
- Support was added to the map clauses to handle structure elements (see Section 2.19.7.1 on page 315).
- Query functions for OpenMP thread affinity were added (see Section 3.2.24 on page 358 to Section 3.2.29 on page 363).
- The lock API was extended with lock routines that support storing a hint with a lock to select a desired lock implementation for a lock's intended usage by the application code (see Section 3.3.2 on page 385).
- Device memory routines were added to allow explicit allocation, deallocation, memory transfers and memory associations (see Section 3.6 on page 397).
- C/C++ Grammar (previously Appendix B) was moved to a separate document.

B.4 Version 3.1 to 4.0 Differences

- Various changes throughout the specification were made to provide initial support of Fortran 2003 (see Section 1.7 on page 31).
- C/C++ array syntax was extended to support array sections (see Section 2.1.5 on page 44).
- The **proc_bind** clause (see Section 2.6.2 on page 80), the **OMP_PLACES** environment variable (see Section 6.5 on page 605), and the **omp_get_proc_bind** runtime routine (see Section 3.2.23 on page 357) were added to support thread affinity policies.
- SIMD directives were added to support SIMD parallelism (see Section 2.9.3 on page 110).
- Implementation defined task scheduling points for untied tasks were removed (see Section 2.10.6 on page 149).
- Device directives (see Section 2.12 on page 160), the **OMP_DEFAULT_DEVICE** environment variable (see Section 6.15 on page 615), and the **omp_set_default_device**, **omp_get_default_device**, **omp_get_num_devices**, **omp_get_num_teams**,

`omp_get_team_num`, and `omp_is_initial_device` routines were added to support execution on devices.

- The **taskgroup** construct (see Section 2.17.6 on page 232) was added to support more flexible deep task synchronization.
- The **atomic** construct (see Section 2.17.7 on page 234) was extended to support atomic swap with the **capture** clause, to allow new atomic update and capture forms, and to support sequentially consistent atomic operations with a new **seq_cst** clause.
- The **depend** clause (see Section 2.17.11 on page 255) was added to support task dependences.
- The **cancel** construct (see Section 2.18.1 on page 263), the **cancellation point** construct (see Section 2.18.2 on page 267), the `omp_get_cancellation` runtime routine (see Section 3.2.9 on page 342) and the **OMP_CANCELLATION** environment variable (see Section 6.11 on page 610) were added to support the concept of cancellation.
- The **reduction** clause (see Section 2.19.5.4 on page 300) was extended and the **declare reduction** construct (see Section 2.19.5.7 on page 304) was added to support user defined reductions.
- The **OMP_DISPLAY_ENV** environment variable (see Section 6.12 on page 611) was added to display the value of ICVs associated with the OpenMP environment variables.
- Examples (previously Appendix A) were moved to a separate document.

B.5 Version 3.0 to 3.1 Differences

- The *bind-var* ICV has been added, which controls whether or not threads are bound to processors (see Section 2.5.1 on page 64). The value of this ICV can be set with the **OMP_PROC_BIND** environment variable (see Section 6.4 on page 604).
- The *nthreads-var* ICV has been modified to be a list of the number of threads to use at each nested parallel region level and the algorithm for determining the number of threads used in a parallel region has been modified to handle a list (see Section 2.6.1 on page 78).
- The **final** and **mergeable** clauses (see Section 2.10.1 on page 135) were added to the **task** construct to support optimization of task data environments.
- The **taskyield** construct (see Section 2.10.4 on page 147) was added to allow user-defined task scheduling points.
- The **atomic** construct (see Section 2.17.7 on page 234) was extended to include **read**, **write**, and **capture** forms, and an **update** clause was added to apply the already existing form of the **atomic** construct.

- Data environment restrictions were changed to allow **intent (in)** and **const**-qualified types for the **firstprivate** clause (see Section 2.19.4.4 on page 286).
- Data environment restrictions were changed to allow Fortran pointers in **firstprivate** (see Section 2.19.4.4 on page 286) and **lastprivate** (see Section 2.19.4.5 on page 288).
- New reduction operators **min** and **max** were added for C and C++ (see Section 2.19.5 on page 293).
- The nesting restrictions in Section 2.20 on page 328 were clarified to disallow closely-nested OpenMP regions within an **atomic** region. This allows an **atomic** region to be consistently defined with other OpenMP regions so that they include all code in the atomic construct.
- The **omp_in_final** runtime library routine (see Section 3.2.22 on page 356) was added to support specialization of final task regions.
- Descriptions of examples (previously Appendix A) were expanded and clarified.
- Replaced incorrect use of **omp_integer_kind** in Fortran interfaces with **selected_int_kind(8)**.

B.6 Version 2.5 to 3.0 Differences

- The definition of active **parallel** region has been changed: in Version 3.0 a **parallel** region is active if it is executed by a team consisting of more than one thread (see Section 1.2.2 on page 2).
- The concept of tasks has been added to the OpenMP execution model (see Section 1.2.5 on page 10 and Section 1.3 on page 20).
- The OpenMP memory model now covers atomicity of memory accesses (see Section 1.4.1 on page 23). The description of the behavior of **volatile** in terms of **flush** was removed.
- In Version 2.5, there was a single copy of the *nest-var*, *dyn-var*, *nthreads-var* and *run-sched-var* internal control variables (ICVs) for the whole program. In Version 3.0, there is one copy of these ICVs per task (see Section 2.5 on page 63). As a result, the **omp_set_num_threads**, **omp_set_nested** and **omp_set_dynamic** runtime library routines now have specified effects when called from inside a **parallel** region (see Section 3.2.1 on page 334, Section 3.2.7 on page 340 and Section 3.2.10 on page 343).
- The *thread-limit-var* ICV has been added, which controls the maximum number of threads participating in the OpenMP program. The value of this ICV can be set with the **OMP_THREAD_LIMIT** environment variable and retrieved with the **omp_get_thread_limit** runtime library routine (see Section 2.5.1 on page 64, Section 3.2.14 on page 348 and Section 6.10 on page 610).

- The *max-active-levels-var* ICV has been added, which controls the number of nested active **parallel** regions. The value of this ICV can be set with the **OMP_MAX_ACTIVE_LEVELS** environment variable and the **omp_set_max_active_levels** runtime library routine, and it can be retrieved with the **omp_get_max_active_levels** runtime library routine (see Section 2.5.1 on page 64, Section 3.2.16 on page 350, Section 3.2.17 on page 351 and Section 6.8 on page 608).
- The *stacksize-var* ICV has been added, which controls the stack size for threads that the OpenMP implementation creates. The value of this ICV can be set with the **OMP_STACKSIZE** environment variable (see Section 2.5.1 on page 64 and Section 6.6 on page 607).
- The *wait-policy-var* ICV has been added, which controls the desired behavior of waiting threads. The value of this ICV can be set with the **OMP_WAIT_POLICY** environment variable (see Section 2.5.1 on page 64 and Section 6.7 on page 608).
- The rules for determining the number of threads used in a **parallel** region have been modified (see Section 2.6.1 on page 78).
- In Version 3.0, the assignment of iterations to threads in a loop construct with a **static** schedule kind is deterministic (see Section 2.9.2 on page 101).
- In Version 3.0, a loop construct may be associated with more than one perfectly nested loop. The number of associated loops is controlled by the **collapse** clause (see Section 2.9.2 on page 101).
- Random access iterators, and variables of unsigned integer type, may now be used as loop iterators in loops associated with a loop construct (see Section 2.9.2 on page 101).
- The schedule kind **auto** has been added, which gives the implementation the freedom to choose any possible mapping of iterations in a loop construct to threads in the team (see Section 2.9.2 on page 101).
- The **task** construct (see Section 2.10 on page 135) has been added, which provides a mechanism for creating tasks explicitly.
- The **taskwait** construct (see Section 2.17.5 on page 230) has been added, which causes a task to wait for all its child tasks to complete.
- Fortran assumed-size arrays now have predetermined data-sharing attributes (see Section 2.19.1.1 on page 270).
- In Version 3.0, static class members variables may appear in a **threadprivate** directive (see Section 2.19.2 on page 274).
- Version 3.0 makes clear where, and with which arguments, constructors and destructors of private and threadprivate class type variables are called (see Section 2.19.2 on page 274, Section 2.19.4.3 on page 285, Section 2.19.4.4 on page 286, Section 2.19.6.1 on page 310 and Section 2.19.6.2 on page 312).

- In Version 3.0, Fortran allocatable arrays may appear in **private**, **firstprivate**, **lastprivate**, **reduction**, **copyin** and **copyprivate** clauses (see Section 2.19.2 on page 274, Section 2.19.4.3 on page 285, Section 2.19.4.4 on page 286, Section 2.19.4.5 on page 288, Section 2.19.5.4 on page 300, Section 2.19.6.1 on page 310 and Section 2.19.6.2 on page 312).
- In Fortran, **firstprivate** is now permitted as an argument to the **default** clause (see Section 2.19.4.1 on page 282).
- For list items in the **private** clause, implementations are no longer permitted to use the storage of the original list item to hold the new list item on the master thread. If no attempt is made to reference the original list item inside the **parallel** region, its value is well defined on exit from the **parallel** region (see Section 2.19.4.3 on page 285).
- The runtime library routines **omp_set_schedule** and **omp_get_schedule** have been added; these routines respectively set and retrieve the value of the *run-sched-var* ICV (see Section 3.2.12 on page 345 and Section 3.2.13 on page 347).
- The **omp_get_level** runtime library routine has been added, which returns the number of nested **parallel** regions enclosing the task that contains the call (see Section 3.2.18 on page 352).
- The **omp_get_ancestor_thread_num** runtime library routine has been added, which returns, for a given nested level of the current thread, the thread number of the ancestor (see Section 3.2.19 on page 353).
- The **omp_get_team_size** runtime library routine has been added, which returns, for a given nested level of the current thread, the size of the thread team to which the ancestor belongs (see Section 3.2.20 on page 354).
- The **omp_get_active_level** runtime library routine has been added, which returns the number of nested active **parallel** regions enclosing the task that contains the call (see Section 3.2.21 on page 355).
- In Version 3.0, locks are owned by tasks, not by threads (see Section 3.3 on page 381).

This page intentionally left blank

Index

Symbols

`_OPENMP` macro, [49](#), [611–613](#)

A

acquire flush, [27](#)
affinity, [80](#)
allocate, [156](#), [158](#)
array sections, [44](#)
array shaping, [43](#)
atomic, [234](#)
atomic construct, [621](#)
attribute clauses, [282](#)
attributes, data-mapping, [314](#)
attributes, data-sharing, [269](#)
auto, [105](#)

B

barrier, [226](#)
barrier, implicit, [228](#)

C

cancel, [263](#)
cancellation constructs, [263](#)
 cancel, [263](#)
 cancellation point, [267](#)
cancellation point, [267](#)
canonical loop form, [95](#)
capture, **atomic**, [234](#)
clauses
 allocate, [158](#)
 attribute data-sharing, [282](#)

collapse, [101](#), [102](#)
copyin, [310](#)
copyprivate, [312](#)
data copying, [309](#)
data-sharing, [282](#)
default, [282](#)
defaultmap, [324](#)
depend, [255](#)
firstprivate, [286](#)
hint, [260](#)
if Clause, [220](#)
in_reduction, [303](#)
lastprivate, [288](#)
linear, [290](#)
map, [315](#)
private, [285](#)
reduction, [300](#)
schedule, [103](#)
shared, [283](#)
task_reduction, [303](#)
combined constructs, [185](#)
 master taskloop, [192](#)
 master taskloop simd, [194](#)
 parallel loop, [186](#)
 parallel master, [191](#)
 parallel master taskloop, [195](#)
 parallel master taskloop simd,
 [196](#)
 parallel sections, [188](#)
 parallel workshare, [189](#)

- parallel worksharing-loop
 - construct, [185](#)
- parallel worksharing-loop SIMD
 - construct, [190](#)
- target parallel**, [203](#)
- target parallel loop**, [208](#)
- target parallel worksharing-loop
 - construct, [205](#)
- target parallel worksharing-loop SIMD
 - construct, [206](#)
- target simd**, [209](#)
- target teams**, [210](#)
- target teams distribute**, [211](#)
- target teams distribute parallel
 - worksharing-loop construct, [215](#)
- target teams distribute parallel
 - worksharing-loop SIMD
 - construct, [216](#)
- target teams distribute simd**, [213](#)
- target teams loop construct, [214](#)
- teams distribute**, [197](#)
- teams distribute parallel
 - worksharing-loop construct, [200](#)
- teams distribute parallel
 - worksharing-loop SIMD
 - construct, [201](#)
- teams distribute simd**, [198](#)
- teams loop**, [202](#)
- compilation sentinels, [50](#)
- compliance, [31](#)
- conditional compilation, [49](#)
- constructs
 - atomic**, [234](#)
 - barrier**, [226](#)
 - cancel**, [263](#)
 - cancellation constructs, [263](#)
 - cancellation point**, [267](#)
 - combined constructs, [185](#)
 - critical**, [223](#)
 - declare mapper**, [326](#)
 - declare target**, [180](#)
 - depobj**, [254](#)
 - device constructs, [160](#)
 - distribute**, [120](#)
 - distribute parallel do**, [125](#)
 - distribute parallel do simd**, [126](#)
 - distribute parallel for**, [125](#)
 - distribute parallel for simd**, [126](#)
 - distribute parallel worksharing-loop
 - construct, [125](#)
 - distribute parallel worksharing-loop
 - SIMD construct, [126](#)
 - distribute simd**, [123](#)
 - do Fortran**, [101](#)
 - flush**, [242](#)
 - for, C/C++**, [101](#)
 - loop**, [128](#)
 - master**, [221](#)
 - master taskloop**, [192](#)
 - master taskloop simd**, [194](#)
 - ordered**, [250](#)
 - parallel**, [74](#)
 - parallel do Fortran**, [185](#)
 - parallel for C/C++**, [185](#)
 - parallel loop**, [186](#)
 - parallel master**, [191](#)
 - parallel master taskloop**, [195](#)
 - parallel master taskloop simd**, [196](#)
 - parallel sections**, [188](#)
 - parallel workshare**, [189](#)
 - parallel worksharing-loop
 - construct, [185](#)
 - parallel worksharing-loop SIMD
 - construct, [190](#)
 - sections**, [86](#)
 - simd**, [110](#)
 - single**, [89](#)
 - target**, [170](#)
 - target data**, [161](#)
 - target enter data**, [164](#)
 - target exit data**, [166](#)
 - target parallel**, [203](#)

- target parallel do**, 205
- target parallel do simd**, 206
- target parallel for**, 205
- target parallel for simd**, 206
- target parallel loop**, 208
- target parallel worksharing-loop
 - construct, 205
- target parallel worksharing-loop SIMD
 - construct, 206
- target simd**, 209
- target teams**, 210
- target teams distribute**, 211
- target teams distribute parallel
 - worksharing-loop construct, 215
- target teams distribute parallel
 - worksharing-loop SIMD
 - construct, 216
- target teams distribute simd**, 213
- target teams loop**, 214
- target update**, 176
- task**, 135
- taskgroup**, 232
- tasking constructs, 135
- taskloop**, 140
- taskloop simd**, 146
- taskwait**, 230
- taskyield**, 147
- teams**, 82
- teams distribute**, 197
- teams distribute parallel
 - worksharing-loop construct, 200
- teams distribute parallel
 - worksharing-loop SIMD
 - construct, 201
- teams distribute simd**, 198
- teams loop**, 202
- workshare**, 92
- worksharing, 86
 - worksharing-loop construct, 101
 - worksharing-loop SIMD construct, 114
- controlling OpenMP thread affinity, 80
- copyin**, 310

- copyprivate**, 312
- critical**, 223

D

- data copying clauses, 309
- data environment, 269
- data terminology, 12
- data-mapping rules and clauses, 314
- data-sharing attribute clauses, 282
- data-sharing attribute rules, 269
- declare mapper**, 326
- declare reduction**, 304
- declare simd**, 116
- declare target**, 180
- declare variant**, 58
- default**, 282
- defaultmap**, 324
- depend**, 255
- depend object, 254
- depobj**, 254
- deprecated features, 627
- device constructs
 - declare mapper**, 326
 - declare target**, 180
 - device constructs, 160
 - distribute**, 120
 - distribute parallel worksharing-loop
 - construct, 125
 - distribute parallel worksharing-loop
 - SIMD construct, 126
 - distribute simd**, 123
 - target**, 170
 - target update**, 176
 - teams**, 82
- device data environments, 24, 164, 166
- device directives, 160
- device memory routines, 397
- directive format, 38
- directives, 37
 - allocate**, 156
 - declare mapper**, 326
 - declare reduction**, 304
 - declare simd**, 116
 - declare target**, 180

- declare variant**, 58
- memory management directives, 152
- metadirective**, 56
- requires**, 60
- scan** Directive, 132
- threadprivate**, 274
- variant directives, 51
- distribute**, 120
- distribute parallel worksharing-loop
 - construct, 125
- distribute parallel worksharing-loop SIMD
 - construct, 126
- distribute simd**, 123
- do**, *Fortran*, 101
- do simd**, 114
- dynamic**, 105
- dynamic thread adjustment, 620

E

- environment variables, 601
 - OMP_AFFINITY_FORMAT**, 613
 - OMP_ALLOCATOR**, 618
 - OMP_CANCELLATION**, 610
 - OMP_DEBUG**, 617
 - OMP_DEFAULT_DEVICE**, 615
 - OMP_DISPLAY_AFFINITY**, 612
 - OMP_DISPLAY_ENV**, 611
 - OMP_DYNAMIC**, 603
 - OMP_MAX_ACTIVE_LEVELS**, 608
 - OMP_MAX_TASK_PRIORITY**, 615
 - OMP_NESTED**, 609
 - OMP_NUM_THREADS**, 602
 - OMP_PLACES**, 605
 - OMP_PROC_BIND**, 604
 - OMP_SCHEDULE**, 601
 - OMP_STACKSIZE**, 607
 - OMP_TARGET_OFFLOAD**, 615
 - OMP_THREAD_LIMIT**, 610
 - OMP_TOOL**, 616
 - OMP_TOOL_LIBRARIES**, 617
 - OMP_WAIT_POLICY**, 608
- event, 396
- event callback registration, 425
- event callback signatures, 459

- event routines, 396
- execution environment routines, 334
- execution model, 20

F

- features history, 627
- firstprivate**, 286
- fixed source form conditional compilation
 - sentinels, 50
- fixed source form directives, 41
- flush**, 242
- flush operation, 25
- flush synchronization, 27
- flush-set, 25
- for**, *C/C++*, 101
- for simd**, 114
- frames, 454
- free source form conditional compilation
 - sentinel, 50
- free source form directives, 41

G

- glossary, 2
- guided**, 105

H

- happens before, 27
- header files, 332
- history of features, 627

I

- ICVs (internal control variables), 63
- if** Clause, 220
- implementation, 619
- implementation terminology, 16
- implicit barrier, 228
- implicit flushes, 246
- in_reduction**, 303
- include files, 332
- internal control variables, 619
- internal control variables (ICVs), 63
- introduction, 1
- iterators, 47

L

lastprivate, 288
linear, 290
list item privatization, 279
lock routines, 381
loop, 128
loop terminology, 8

M

map, 315
master, 221
master taskloop, 192
master taskloop simd, 194
memory allocators, 152
memory management, 152
memory management directives
 memory management directives, 152
memory management routines, 406
memory model, 23
memory spaces, 152
metadirective, 56
modifying and retrieving ICV values, 68
modifying ICVs, 66

N

nesting of regions, 328
normative references, 31

O

OMP_AFFINITY_FORMAT, 613
omp_alloc, 413
OMP_ALLOCATOR, 618
OMP_CANCELLATION, 610
omp_capture_affinity, 368
OMP_DEBUG, 617
OMP_DEFAULT_DEVICE, 615
omp_destroy_allocator, 410
omp_destroy_lock, 387
omp_destroy_nest_lock, 387
OMP_DISPLAY_AFFINITY, 612
omp_display_affinity, 367
OMP_DISPLAY_ENV, 611
OMP_DYNAMIC, 603
omp_free, 414

omp_fulfill_event, 396
omp_get_active_level, 355
omp_get_affinity_format, 366
omp_get_ancestor_thread_num, 353
omp_get_cancellation, 342
omp_get_default_allocator, 412
omp_get_default_device, 370
omp_get_device_num, 372
omp_get_dynamic, 341
omp_get_initial_device, 376
omp_get_level, 352
omp_get_max_active_levels, 351
omp_get_max_task_priority, 377
omp_get_max_threads, 336
omp_get_nested, 344
omp_get_num_devices, 371
omp_get_num_places, 358
omp_get_num_procs, 338
omp_get_num_teams, 373
omp_get_num_threads, 335
omp_get_partition_num_places,
 362
omp_get_partition_place_nums,
 363
omp_get_place_num, 362
omp_get_place_num_procs, 359
omp_get_place_proc_ids, 360
omp_get_proc_bind, 357
omp_get_schedule, 347
omp_get_supported_active
 _levels, 349
omp_get_team_num, 374
omp_get_team_size, 354
omp_get_thread_limit, 348
omp_get_thread_num, 337
omp_get_wtick, 395
omp_get_wtime, 394
omp_in_final, 356
omp_in_parallel, 339
omp_init_allocator, 409
omp_init_lock, 384, 385
omp_init_nest_lock, 384, 385
omp_is_initial_device, 375

<code>OMP_MAX_ACTIVE_LEVELS</code> , 608	<code>ompd_bp_thread_end</code> , 597
<code>OMP_MAX_TASK_PRIORITY</code> , 615	<code>ompd_callback_device_host</code>
<code>OMP_NESTED</code> , 609	<code>_fn_t</code> , 554
<code>OMP_NUM_THREADS</code> , 602	<code>ompd_callback_get_thread</code>
<code>omp_pause_resource</code> , 378	<code>_context_for_thread_id</code>
<code>omp_pause_resource_all</code> , 380	<code>_fn_t</code> , 547
<code>OMP_PLACES</code> , 605	<code>ompd_callback_memory_alloc</code>
<code>OMP_PROC_BIND</code> , 604	<code>_fn_t</code> , 546
<code>OMP_SCHEDULE</code> , 601	<code>ompd_callback_memory_free</code>
<code>omp_set_affinity_format</code> , 364	<code>_fn_t</code> , 546
<code>omp_set_default_allocator</code> , 411	<code>ompd_callback_memory_read</code>
<code>omp_set_default_device</code> , 369	<code>_fn_t</code> , 551
<code>omp_set_dynamic</code> , 340	<code>ompd_callback_memory_write</code>
<code>omp_set_lock</code> , 388	<code>_fn_t</code> , 553
<code>omp_set_max_active_levels</code> , 350	<code>ompd_callback_print_string</code>
<code>omp_set_nest_lock</code> , 388	<code>_fn_t</code> , 556
<code>omp_set_nested</code> , 343	<code>ompd_callback_sizeof_fn_t</code> , 549
<code>omp_set_num_threads</code> , 334	<code>ompd_callback_symbol_addr</code>
<code>omp_set_schedule</code> , 345	<code>_fn_t</code> , 550
<code>OMP_STACKSIZE</code> , 607	<code>ompd_callbacks_t</code> , 556
<code>omp_target_alloc</code> , 397	<code>ompd_dll_locations_valid</code> , 536
<code>omp_target_associate_ptr</code> , 403	<code>ompd_dll_locations</code> , 535
<code>omp_target_disassociate_ptr</code> , 405	<code>ompt_callback_buffer</code>
<code>omp_target_free</code> , 399	<code>_complete_t</code> , 487
<code>omp_target_is_present</code> , 400	<code>ompt_callback_buffer</code>
<code>omp_target_memcpy</code> , 400	<code>_request_t</code> , 486
<code>omp_target_memcpy_rect</code> , 402	<code>ompt_callback_cancel_t</code> , 481
<code>OMP_TARGET_OFFLOAD</code> , 615	<code>ompt_callback_control</code>
<code>omp_test_lock</code> , 392	<code>_tool_t</code> , 495
<code>omp_test_nest_lock</code> , 392	<code>ompt_callback_dependences_t</code> , 468
<code>OMP_THREAD_LIMIT</code> , 610	<code>ompt_callback_dispatch_t</code> , 465
<code>OMP_TOOL</code> , 616	<code>ompt_callback_device</code>
<code>OMP_TOOL_LIBRARIES</code> , 617	<code>_finalize_t</code> , 484
<code>omp_unset_lock</code> , 390	<code>ompt_callback_device</code>
<code>omp_unset_nest_lock</code> , 390	<code>_initialize_t</code> , 482
<code>OMP_WAIT_POLICY</code> , 608	<code>ompt_callback_flush_t</code> , 480
<code>ompd_bp_device_begin</code> , 598	<code>ompt_callback_implicit</code>
<code>ompd_bp_device_end</code> , 599	<code>_task_t</code> , 471
<code>ompd_bp_parallel_begin</code> , 594	<code>ompt_callback_master_t</code> , 473
<code>ompd_bp_parallel_end</code> , 595	<code>ompt_callback_mutex</code>
<code>ompd_bp_task_begin</code> , 595	<code>_acquire_t</code> , 476
<code>ompd_bp_task_end</code> , 596	<code>ompt_callback_mutex_t</code> , 477
<code>ompd_bp_thread_begin</code> , 597	<code>ompt_callback_nest_lock_t</code> , 479

- `ompt_callback_parallel`
 - `_begin_t`, 461
- `ompt_callback_parallel`
 - `_end_t`, 463
- `ompt_callback_sync_region_t`, 474
- `ompt_callback_device_load_t`, 484
- `ompt_callback_device`
 - `_unload_t`, 486
- `ompt_callback_target_data`
 - `_op_t`, 488
- `ompt_callback_target_map_t`, 492
- `ompt_callback_target`
 - `_submit_t`, 494
- `ompt_callback_target_t`, 490
- `ompt_callback_task_create_t`, 467
- `ompt_callback_task`
 - `_dependence_t`, 470
- `ompt_callback_task`
 - `_schedule_t`, 470
- `ompt_callback_thread`
 - `_begin_t`, 459
- `ompt_callback_thread_end_t`, 460
- `ompt_callback_work_t`, 464
- OpenMP compliance, 31
- ordered**, 250

P

- parallel**, 74
- parallel loop**, 186
- parallel master construct**, 191
- parallel master taskloop**, 195
- parallel master taskloop simd**, 196
- parallel sections**, 188
- parallel workshare**, 189
- parallel worksharing-loop construct**, 185
- parallel worksharing-loop SIMD**
 - construct, 190
- private**, 285

R

- read, atomic**, 234
- reduction**, 300
- reduction clauses**, 293
- release flush**, 27

- requires**, 60
- runtime**, 105
- runtime library definitions, 332
- runtime library routines, 331

S

- scan Directive**, 132
- scheduling**, 149
- sections**, 86
- shared**, 283
- simd**, 110
- SIMD Directives, 110
- Simple Lock Routines, 382
- single**, 89
- stand-alone directives, 42
- static**, 104
- strong flush, 25
- synchronization constructs, 223
- synchronization constructs and clauses, 223
- synchronization hints, 260
- synchronization terminology, 9

T

- target**, 170
- target data**, 161
- target memory routines, 397
- target parallel**, 203
- target parallel loop**, 208
- target parallel worksharing-loop construct
 - construct, 205
- target parallel worksharing-loop SIMD
 - construct, 206
- target simd**, 209
- target teams**, 210
- target teams distribute**, 211
- target teams distribute parallel
 - worksharing-loop construct, 215
- target teams distribute parallel
 - worksharing-loop SIMD
 - construct, 216
- target teams distribute simd**, 213
- target teams loop**, 214
- target update**, 176
- task**, 135

- task scheduling, [149](#)
- task_reduction**, [303](#)
- taskgroup**, [232](#)
- tasking constructs, [135](#)
- tasking terminology, [10](#)
- taskloop**, [140](#)
- taskloop simd**, [146](#)
- taskwait**, [230](#)
- taskyield**, [147](#)
- teams**, [82](#)
- teams distribute**, [197](#)
- teams distribute parallel worksharing-loop construct, [200](#)
- teams distribute parallel worksharing-loop SIMD construct, [201](#)
- teams distribute simd**, [198](#)
- teams loop**, [202](#)
- thread affinity, [80](#)
- threadprivate**, [274](#)
- timer, [394](#)
- timing routines, [394](#)
- tool control, [415](#)
- tool initialization, [423](#)
- tool interfaces definitions, [419](#), [534](#)
- tools header files, [419](#), [534](#)
- tracing device activity, [427](#)

U

- update, atomic**, [234](#)

V

- variables, environment, [601](#)
- variant directives, [51](#)

W

- wait identifier, [456](#)
- wall clock timer, [394](#)
- workshare**, [92](#)
- worksharing
 - constructs, [86](#)
 - parallel, [185](#)
 - scheduling, [109](#)
- worksharing constructs, [86](#)
- worksharing-loop construct, [101](#)
- worksharing-loop SIMD construct, [114](#)
- write, atomic**, [234](#)



OpenMP Application Programming Interface

Examples

Version 4.5.0 – November 2016

Source codes for OpenMP 4.5.0 Examples can be downloaded from [github](#).

Copyright © 1997-2016 OpenMP Architecture Review Board.

Permission to copy without fee all or part of this material is granted, provided the OpenMP Architecture Review Board copyright notice and the title of this document appear. Notice is given that copying is by permission of OpenMP Architecture Review Board.

This page intentionally left blank

Contents

Introduction	1
Examples	2
1. Parallel Execution	3
1.1. A Simple Parallel Loop	5
1.2. The parallel Construct	6
1.3. Controlling the Number of Threads on Multiple Nesting Levels	8
1.4. Interaction Between the num_threads Clause and omp_set_dynamic	11
1.5. Fortran Restrictions on the do Construct	13
1.6. The nowait Clause	15
1.7. The collapse Clause	18
1.8. linear Clause in Loop Constructs	22
1.9. The parallel sections Construct	24
1.10. The firstprivate Clause and the sections Construct	25
1.11. The single Construct	27
1.12. The workshare Construct	29
1.13. The master Construct	33
1.14. Parallel Random Access Iterator Loop	35
1.15. The omp_set_dynamic and omp_set_num_threads Routines	36
1.16. The omp_get_num_threads Routine	38
2. OpenMP Affinity	40
2.1. The proc_bind Clause	42
2.1.1. Spread Affinity Policy	42
2.1.2. Close Affinity Policy	44
2.1.3. Master Affinity Policy	47

2.2. Affinity Query Functions	48
3. Tasking	51
3.1. The task and taskwait Constructs	52
3.2. Task Priority	71
3.3. Task Dependences	73
3.3.1. Flow Dependence	73
3.3.2. Anti-dependence	74
3.3.3. Output Dependence	75
3.3.4. Concurrent Execution with Dependences	76
3.3.5. Matrix multiplication	78
3.4. The taskgroup Construct	80
3.5. The taskyield Construct	83
3.6. The taskloop Construct	85
4. Devices	87
4.1. target Construct	88
4.1.1. target Construct on parallel Construct	88
4.1.2. target Construct with map Clause	89
4.1.3. map Clause with to/from map-types	90
4.1.4. map Clause with Array Sections	91
4.1.5. target Construct with if Clause	93
4.2. target data Construct	96
4.2.1. Simple target data Construct	96
4.2.2. target data Region Enclosing Multiple target Regions	97
4.2.3. target data Construct with Orphaned Call	101
4.2.4. target data Construct with if Clause	104
4.3. target enter data and target exit data Constructs	108
4.4. target update Construct	111
4.4.1. Simple target data and target update Constructs	111
4.4.2. target update Construct with if Clause	113
4.5. declare target Construct	115
4.5.1. declare target and end declare target for a Function	115
4.5.2. declare target Construct for Class Type	117

4.5.3.	declare target and end declare target for Variables	117
4.5.4.	declare target and end declare target with declare simd	120
4.5.5.	declare target Directive with link Clause	123
4.6.	teams Constructs	126
4.6.1.	target and teams Constructs with omp_get_num_teams and omp_get_team_num Routines	126
4.6.2.	target , teams , and distribute Constructs	128
4.6.3.	target teams , and Distribute Parallel Loop Constructs	129
4.6.4.	target teams and Distribute Parallel Loop Constructs with Scheduling Clauses	131
4.6.5.	target teams and distribute simd Constructs	132
4.6.6.	target teams and Distribute Parallel Loop SIMD Constructs	134
4.7.	Asynchronous target Execution and Dependences	135
4.7.1.	Asynchronous target with Tasks	135
4.7.2.	nowait Clause on target Construct	139
4.7.3.	Asynchronous target with nowait and depend Clauses	141
4.8.	Array Sections in Device Constructs	144
4.9.	Device Routines	148
4.9.1.	omp_is_initial_device Routine	148
4.9.2.	omp_get_num_devices Routine	150
4.9.3.	omp_set_default_device and omp_get_default_device Routines	151
4.9.4.	Target Memory and Device Pointers Routines	152
5.	SIMD	154
5.1.	simd and declare simd Constructs	155
5.2.	inbranch and notinbranch Clauses	161
5.3.	Loop-Carried Lexical Forward Dependence	165
6.	Synchronization	169
6.1.	The critical Construct	171
6.2.	Worksharing Constructs Inside a critical Construct	174
6.3.	Binding of barrier Regions	176
6.4.	The atomic Construct	178

6.5. Restrictions on the atomic Construct	184
6.6. The flush Construct without a List	187
6.7. The ordered Clause and the ordered Construct	190
6.8. Doacross Loop Nest	194
6.9. Lock Routines	200
6.9.1. The omp_init_lock Routine	200
6.9.2. The omp_init_lock_with_hint Routine	201
6.9.3. Ownership of Locks	202
6.9.4. Simple Lock Routines	203
6.9.5. Nestable Lock Routines	206
7. Data Environment	209
7.1. The threadprivate Directive	211
7.2. The default (none) Clause	217
7.3. The private Clause	219
7.4. Fortran Private Loop Iteration Variables	223
7.5. Fortran Restrictions on shared and private Clauses with Common Blocks	225
7.6. Fortran Restrictions on Storage Association with the private Clause	227
7.7. C/C++ Arrays in a firstprivate Clause	230
7.8. The lastprivate Clause	232
7.9. The reduction Clause	233
7.10. The copyin Clause	240
7.11. The copyprivate Clause	242
7.12. C++ Reference in Data-Sharing Clauses	246
7.13. Fortran ASSOCIATE Construct	247
8. Memory Model	249
8.1. The OpenMP Memory Model	250
8.2. Race Conditions Caused by Implied Copies of Shared Variables in Fortran	256
9. Program Control	257
9.1. Conditional Compilation	259
9.2. Internal Control Variables (ICVs)	260
9.3. Placement of flush , barrier , taskwait and taskyield Directives	263
9.4. Cancellation Constructs	267

9.5. Nested Loop Constructs	272
9.6. Restrictions on Nesting of Regions	275
A. Document Revision History	281
A.1. Changes from 4.0.2 to 4.5.0	281
A.2. Changes from 4.0.1 to 4.0.2	282
A.3. Changes from 4.0 to 4.0.1	282
A.4. Changes from 3.1 to 4.0	282

Introduction

This collection of programming examples supplements the OpenMP API for Shared Memory Parallelization specifications, and is not part of the formal specifications. It assumes familiarity with the OpenMP specifications, and shares the typographical conventions used in that document.

Note – This first release of the OpenMP Examples reflects the OpenMP Version 4.5 specifications. Additional examples are being developed and will be published in future releases of this document.

The OpenMP API specification provides a model for parallel programming that is portable across shared memory architectures from different vendors. Compilers from numerous vendors support the OpenMP API.

The directives, library routines, and environment variables demonstrated in this document allow users to create and manage parallel programs while permitting portability. The directives extend the C, C++ and Fortran base languages with single program multiple data (SPMD) constructs, tasking constructs, device constructs, worksharing constructs, and synchronization constructs, and they provide support for sharing and privatizing data. The functionality to control the runtime environment is provided by library routines and environment variables. Compilers that support the OpenMP API often include a command line option to the compiler that activates and allows interpretation of all OpenMP directives.

The latest source codes for OpenMP Examples can be downloaded from the **sources** directory at <https://github.com/OpenMP/Examples>. The codes for this OpenMP 4.5.0 Examples document have the tag *v4.5.0*.

Complete information about the OpenMP API and a list of the compilers that support the OpenMP API can be found at the OpenMP.org web site

<http://www.openmp.org>

Examples

The following are examples of the OpenMP API directives, constructs, and routines.

C / C++

A statement following a directive is compound only when necessary, and a non-compound statement is indented with respect to a directive preceding it.

C / C++

Each example is labeled as *ename.seqno.ext*, where *ename* is the example name, *seqno* is the sequence number in a section, and *ext* is the source file extension to indicate the code type and source form. *ext* is one of the following:

- *c* – C code,
- *cpp* – C++ code,
- *f* – Fortran code in fixed form, and
- *f90* – Fortran code in free form.

Parallel Execution

A single thread, the *initial thread*, begins sequential execution of an OpenMP enabled program, as if the whole program is in an implicit parallel region consisting of an implicit task executed by the *initial thread*.

A **parallel** construct encloses code, forming a parallel region. An *initial thread* encountering a **parallel** region forks (creates) a team of threads at the beginning of the **parallel** region, and joins them (removes from execution) at the end of the region. The initial thread becomes the master thread of the team in a **parallel** region with a *thread* number equal to zero, the other threads are numbered from 1 to number of threads minus 1. A team may be comprised of just a single thread.

Each thread of a team is assigned an implicit task consisting of code within the parallel region. The task that creates a parallel region is suspended while the tasks of the team are executed. A thread is tied to its task; that is, only the thread assigned to the task can execute that task. After completion of the **parallel** region, the master thread resumes execution of the generating task.

Any task within a **parallel** region is allowed to encounter another **parallel** region to form a nested **parallel** region. The parallelism of a nested **parallel** region (whether it forks additional threads, or is executed serially by the encountering task) can be controlled by the **OMP_NESTED** environment variable or the **omp_set_nested()** API routine with arguments indicating true or false.

The number of threads of a **parallel** region can be set by the **OMP_NUM_THREADS** environment variable, the **omp_set_num_threads()** routine, or on the **parallel** directive with the **num_threads** clause. The routine overrides the environment variable, and the clause overrides all. Use the **OMP_DYNAMIC** or the **omp_set_dynamic()** function to specify that the OpenMP implementation dynamically adjust the number of threads for **parallel** regions. The default setting for dynamic adjustment is implementation defined. When dynamic adjustment is on and the number of threads is specified, the number of threads becomes an upper limit for the number of threads to be provided by the OpenMP runtime.

WORKSHARING CONSTRUCTS

A worksharing construct distributes the execution of the associated region among the members of the team that encounter it. There is an implied barrier at the end of the worksharing region (there is no barrier at the beginning). The worksharing constructs are:

- loop constructs: **for** and **do**
- **sections**
- **single**
- **workshare**

The **for** and **do** constructs (loop constructs) create a region consisting of a loop. A loop controlled by a loop construct is called an *associated* loop. Nested loops can form a single region when the **collapse** clause (with an integer argument) designates the number of *associated* loops to be executed in parallel, by forming a "single iteration space" for the specified number of nested loops. The **ordered** clause can also control multiple associated loops.

An associated loop must adhere to a "canonical form" (specified in the *Canonical Loop Form* of the OpenMP Specifications document) which allows the iteration count (of all associated loops) to be computed before the (outermost) loop is executed. Most common loops comply with the canonical form, including C++ iterators.

A **single** construct forms a region in which only one thread (any one of the team) executes the region. The other threads wait at the implied barrier at the end, unless the **nowait** clause is specified.

The **sections** construct forms a region that contains one or more structured blocks. Each block of a **sections** directive is constructed with a **section** construct, and executed once by one of the threads (any one) in the team. (If only one block is formed in the region, the **section** construct, which is used to separate blocks, is not required.) The other threads wait at the implied barrier at the end, unless the **nowait** clause is specified.

The **workshare** construct is a Fortran feature that consists of a region with a single structure block (section of code). Statements in the **workshare** region are divided into units of work, and executed (once) by threads of the team.

MASTER CONSTRUCT

The **master** construct is not a worksharing construct. The master region is executed only by the master thread. There is no implicit barrier (and flush) at the end of the **master** region; hence the other threads of the team continue execution beyond code statements beyond the **master** region.

1.1 A Simple Parallel Loop

The following example demonstrates how to parallelize a simple loop using the parallel loop construct. The loop iteration variable is private by default, so it is not necessary to specify it explicitly in a **private** clause.

C / C++

Example ploop.1.c

```
S-1 void simple(int n, float *a, float *b)
S-2 {
S-3     int i;
S-4
S-5     #pragma omp parallel for
S-6         for (i=1; i<n; i++) /* i is private by default */
S-7         b[i] = (a[i] + a[i-1]) / 2.0;
S-8 }
```

C / C++

Fortran

Example ploop.1.f

```
S-1     SUBROUTINE SIMPLE(N, A, B)
S-2
S-3     INTEGER I, N
S-4     REAL B(N), A(N)
S-5
S-6     !$OMP PARALLEL DO !I is private by default
S-7     DO I=2,N
S-8         B(I) = (A(I) + A(I-1)) / 2.0
S-9     ENDDO
S-10    !$OMP END PARALLEL DO
S-11
S-12    END SUBROUTINE SIMPLE
```

Fortran

1.2 The parallel Construct

The **parallel** construct can be used in coarse-grain parallel programs. In the following example, each thread in the **parallel** region decides what part of the global array *x* to work on, based on the thread number:

C / C++

Example parallel.1.c

```
S-1  #include <omp.h>
S-2
S-3  void subdomain(float *x, int istart, int ipoints)
S-4  {
S-5      int i;
S-6
S-7      for (i = 0; i < ipoints; i++)
S-8          x[istart+i] = 123.456;
S-9  }
S-10
S-11 void sub(float *x, int npoints)
S-12 {
S-13     int iam, nt, ipoints, istart;
S-14
S-15     #pragma omp parallel default(shared) private(iam,nt,ipoints,istart)
S-16     {
S-17         iam = omp_get_thread_num();
S-18         nt = omp_get_num_threads();
S-19         ipoints = npoints / nt;    /* size of partition */
S-20         istart = iam * ipoints;    /* starting array index */
S-21         if (iam == nt-1)           /* last thread may do more */
S-22             ipoints = npoints - istart;
S-23         subdomain(x, istart, ipoints);
S-24     }
S-25 }
S-26
S-27 int main()
S-28 {
S-29     float array[10000];
S-30
S-31     sub(array, 10000);
S-32
S-33     return 0;
S-34 }
```

C / C++

1

Example parallel.1.f

```

S-1      SUBROUTINE SUBDOMAIN(X, ISTART, IPOINTS)
S-2          INTEGER ISTART, IPOINTS
S-3          REAL X(*)
S-4
S-5          INTEGER I
S-6
S-7          DO 100 I=1, IPOINTS
S-8              X(ISTART+I) = 123.456
S-9      100    CONTINUE
S-10
S-11      END SUBROUTINE SUBDOMAIN
S-12
S-13      SUBROUTINE SUB(X, NPOINTS)
S-14          INCLUDE "omp_lib.h"      ! or USE OMP_LIB
S-15
S-16          REAL X(*)
S-17          INTEGER NPOINTS
S-18          INTEGER IAM, NT, IPOINTS, ISTART
S-19
S-20      !$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,NPOINTS)
S-21
S-22          IAM = OMP_GET_THREAD_NUM()
S-23          NT = OMP_GET_NUM_THREADS()
S-24          IPOINTS = NPOINTS/NT
S-25          ISTART = IAM * IPOINTS
S-26          IF (IAM .EQ. NT-1) THEN
S-27              IPOINTS = NPOINTS - ISTART
S-28          ENDIF
S-29          CALL SUBDOMAIN(X, ISTART, IPOINTS)
S-30
S-31      !$OMP END PARALLEL
S-32      END SUBROUTINE SUB
S-33
S-34      PROGRAM PAREXAMPLE
S-35          REAL ARRAY(10000)
S-36          CALL SUB(ARRAY, 10000)
S-37      END PROGRAM PAREXAMPLE

```


1.3 Controlling the Number of Threads on Multiple Nesting Levels

The following examples demonstrate how to use the **OMP_NUM_THREADS** environment variable to control the number of threads on multiple nesting levels:

C / C++

Example nthrs_nesting.1.c

```
S-1  #include <stdio.h>
S-2  #include <omp.h>
S-3  int main (void)
S-4  {
S-5      omp_set_nested(1);
S-6      omp_set_dynamic(0);
S-7      #pragma omp parallel
S-8      {
S-9          #pragma omp parallel
S-10         {
S-11             #pragma omp single
S-12             {
S-13                 /*
S-14                 * If OMP_NUM_THREADS=2,3 was set, the following should print:
S-15                 * Inner: num_thds=3
S-16                 * Inner: num_thds=3
S-17                 *
S-18                 * If nesting is not supported, the following should print:
S-19                 * Inner: num_thds=1
S-20                 * Inner: num_thds=1
S-21                 */
S-22                 printf ("Inner: num_thds=%d\n", omp_get_num_threads());
S-23             }
S-24         }
S-25         #pragma omp barrier
S-26         omp_set_nested(0);
S-27         #pragma omp parallel
S-28         {
S-29             #pragma omp single
S-30             {
S-31                 /*
S-32                 * Even if OMP_NUM_THREADS=2,3 was set, the following should
S-33                 * print, because nesting is disabled:
S-34                 * Inner: num_thds=1
S-35                 * Inner: num_thds=1
S-36                 */
S-37                 printf ("Inner: num_thds=%d\n", omp_get_num_threads());
```

```

S-38         }
S-39     }
S-40     #pragma omp barrier
S-41     #pragma omp single
S-42     {
S-43         /*
S-44         * If OMP_NUM_THREADS=2,3 was set, the following should print:
S-45         * Outer: num_thds=2
S-46         */
S-47         printf ("Outer: num_thds=%d\n", omp_get_num_threads());
S-48     }
S-49 }
S-50 return 0;
S-51 }

```

C / C++

Fortran

1

Example nthrs_nesting.1.f

```

S-1     program icv
S-2     use omp_lib
S-3     call omp_set_nested(.true.)
S-4     call omp_set_dynamic(.false.)
S-5     !$omp parallel
S-6     !$omp parallel
S-7     !$omp single
S-8         ! If OMP_NUM_THREADS=2,3 was set, the following should print:
S-9         ! Inner: num_thds= 3
S-10        ! Inner: num_thds= 3
S-11        ! If nesting is not supported, the following should print:
S-12        ! Inner: num_thds= 1
S-13        ! Inner: num_thds= 1
S-14        print *, "Inner: num_thds=", omp_get_num_threads()
S-15     !$omp end single
S-16     !$omp end parallel
S-17     !$omp barrier
S-18         call omp_set_nested(.false.)
S-19     !$omp parallel
S-20     !$omp single
S-21         ! Even if OMP_NUM_THREADS=2,3 was set, the following should print,
S-22         ! because nesting is disabled:
S-23         ! Inner: num_thds= 1
S-24         ! Inner: num_thds= 1
S-25         print *, "Inner: num_thds=", omp_get_num_threads()
S-26     !$omp end single
S-27     !$omp end parallel
S-28     !$omp barrier

```

```
S-29  !$omp single
S-30      ! If OMP_NUM_THREADS=2,3 was set, the following should print:
S-31      ! Outer: num_thds= 2
S-32      print *, "Outer: num_thds=", omp_get_num_threads()
S-33  !$omp end single
S-34  !$omp end parallel
S-35      end
```

Fortran

1.4 Interaction Between the num_threads Clause and omp_set_dynamic

The following example demonstrates the **num_threads** clause and the effect of the **omp_set_dynamic** routine on it.

The call to the **omp_set_dynamic** routine with argument **0** in C/C++, or **.FALSE.** in Fortran, disables the dynamic adjustment of the number of threads in OpenMP implementations that support it. In this case, 10 threads are provided. Note that in case of an error the OpenMP implementation is free to abort the program or to supply any number of threads available.

C / C++

Example nthrs_dynamic.1.c

```
S-1 #include <omp.h>
S-2 int main()
S-3 {
S-4     omp_set_dynamic(0);
S-5     #pragma omp parallel num_threads(10)
S-6     {
S-7         /* do work here */
S-8     }
S-9     return 0;
S-10 }
```

C / C++

Fortran

Example nthrs_dynamic.1.f

```
S-1      PROGRAM EXAMPLE
S-2      INCLUDE "omp_lib.h"      ! or USE OMP_LIB
S-3      CALL OMP_SET_DYNAMIC(.FALSE.)
S-4      !$OMP    PARALLEL NUM_THREADS(10)
S-5              ! do work here
S-6      !$OMP    END PARALLEL
S-7      END PROGRAM EXAMPLE
```

Fortran

The call to the **omp_set_dynamic** routine with a non-zero argument in C/C++, or **.TRUE.** in Fortran, allows the OpenMP implementation to choose any number of threads between 1 and 10.

C / C++

1 *Example nthrs_dynamic.2.c*

```
S-1  #include <omp.h>
S-2  int main()
S-3  {
S-4      omp_set_dynamic(1);
S-5      #pragma omp parallel num_threads(10)
S-6      {
S-7          /* do work here */
S-8      }
S-9      return 0;
S-10 }
```

C / C++

Fortran

2 *Example nthrs_dynamic.2.f*

```
S-1      PROGRAM EXAMPLE
S-2          INCLUDE "omp_lib.h"          ! or USE OMP_LIB
S-3          CALL OMP_SET_DYNAMIC(.TRUE.)
S-4      !$OMP      PARALLEL NUM_THREADS(10)
S-5          ! do work here
S-6      !$OMP      END PARALLEL
S-7      END PROGRAM EXAMPLE
```

Fortran

3 It is good practice to set the *dyn-var* ICV explicitly by calling the **omp_set_dynamic** routine, as
4 its default setting is implementation defined.

1.5 Fortran Restrictions on the do Construct

Fortran

If an **end do** directive follows a *do-construct* in which several **DO** statements share a **DO** termination statement, then a **do** directive can only be specified for the outermost of these **DO** statements. The following example contains correct usages of loop constructs:

Example fort_do.1.f

```
S-1      SUBROUTINE WORK(I, J)
S-2      INTEGER I, J
S-3      END SUBROUTINE WORK
S-4
S-5      SUBROUTINE DO_GOOD()
S-6      INTEGER I, J
S-7      REAL A(1000)
S-8
S-9      DO 100 I = 1, 10
S-10     !$OMP DO
S-11      DO 100 J = 1, 10
S-12      CALL WORK(I, J)
S-13     100 CONTINUE      ! !$OMP ENDDO implied here
S-14
S-15     !$OMP DO
S-16      DO 200 J = 1, 10
S-17     200      A(I) = I + 1
S-18     !$OMP ENDDO
S-19
S-20     !$OMP DO
S-21      DO 300 I = 1, 10
S-22      DO 300 J = 1, 10
S-23      CALL WORK(I, J)
S-24     300 CONTINUE
S-25     !$OMP ENDDO
S-26     END SUBROUTINE DO_GOOD
```

The following example is non-conforming because the matching **do** directive for the **end do** does not precede the outermost loop:

Example fort_do.2.f

```
S-1      SUBROUTINE WORK(I, J)
S-2      INTEGER I, J
S-3      END SUBROUTINE WORK
S-4
S-5      SUBROUTINE DO_WRONG
S-6      INTEGER I, J
```

```
S-7
S-8      DO 100 I = 1,10
S-9      !$OMP      DO
S-10      DO 100 J = 1,10
S-11      CALL WORK(I,J)
S-12      100      CONTINUE
S-13      !$OMP      ENDDO
S-14      END SUBROUTINE DO_WRONG
```

Fortran

1.6 The `nowait` Clause

If there are multiple independent loops within a **parallel** region, you can use the **nowait** clause to avoid the implied barrier at the end of the loop construct, as follows:

C / C++

Example nowait.1.c

```
S-1 #include <math.h>
S-2
S-3 void nowait_example(int n, int m, float *a, float *b, float *y, float *z)
S-4 {
S-5     int i;
S-6     #pragma omp parallel
S-7     {
S-8         #pragma omp for nowait
S-9         for (i=1; i<n; i++)
S-10             b[i] = (a[i] + a[i-1]) / 2.0;
S-11
S-12         #pragma omp for nowait
S-13         for (i=0; i<m; i++)
S-14             y[i] = sqrt(z[i]);
S-15     }
S-16 }
```

C / C++

Fortran

Example nowait.1.f

```
S-1      SUBROUTINE NOWAIT_EXAMPLE(N, M, A, B, Y, Z)
S-2
S-3      INTEGER N, M
S-4      REAL A(*), B(*), Y(*), Z(*)
S-5
S-6      INTEGER I
S-7
S-8      !$OMP PARALLEL
S-9
S-10     !$OMP DO
S-11         DO I=2,N
S-12             B(I) = (A(I) + A(I-1)) / 2.0
S-13         ENDDO
S-14     !$OMP END DO NOWAIT
S-15
S-16     !$OMP DO
S-17         DO I=1,M
S-18             Y(I) = SQRT(Z(I))
```



```

S-19         ENDDO
S-20     !$OMP END DO NOWAIT
S-21
S-22     !$OMP END PARALLEL
S-23
S-24         END SUBROUTINE NOWAIT_EXAMPLE

```

Fortran

In the following example, static scheduling distributes the same logical iteration numbers to the threads that execute the three loop regions. This allows the **nowait** clause to be used, even though there is a data dependence between the loops. The dependence is satisfied as long the same thread executes the same logical iteration numbers in each loop.

Note that the iteration count of the loops must be the same. The example satisfies this requirement, since the iteration space of the first two loops is from **0** to **n-1** (from **1** to **N** in the Fortran version), while the iteration space of the last loop is from **1** to **n** (**2** to **N+1** in the Fortran version).

C / C++

Example nowait.2.c

```

S-1
S-2     #include <math.h>
S-3     void nowait_example2(int n, float *a, float *b, float *c, float *y, float
S-4     *z)
S-5     {
S-6         int i;
S-7         #pragma omp parallel
S-8         {
S-9             #pragma omp for schedule(static) nowait
S-10            for (i=0; i<n; i++)
S-11                c[i] = (a[i] + b[i]) / 2.0f;
S-12            #pragma omp for schedule(static) nowait
S-13            for (i=0; i<n; i++)
S-14                z[i] = sqrtf(c[i]);
S-15            #pragma omp for schedule(static) nowait
S-16            for (i=1; i<=n; i++)
S-17                y[i] = z[i-1] + a[i];
S-18        }
S-19    }

```

C / C++

1

Example nowait.2.f90

```

S-1      SUBROUTINE NOWAIT_EXAMPLE2(N, A, B, C, Y, Z)
S-2      INTEGER N
S-3      REAL A(*), B(*), C(*), Y(*), Z(*)
S-4      INTEGER I
S-5      !$OMP PARALLEL
S-6      !$OMP DO SCHEDULE(STATIC)
S-7      DO I=1,N
S-8      C(I) = (A(I) + B(I)) / 2.0
S-9      ENDDO
S-10     !$OMP END DO NOWAIT
S-11     !$OMP DO SCHEDULE(STATIC)
S-12     DO I=1,N
S-13     Z(I) = SQRT(C(I))
S-14     ENDDO
S-15     !$OMP END DO NOWAIT
S-16     !$OMP DO SCHEDULE(STATIC)
S-17     DO I=2,N+1
S-18     Y(I) = Z(I-1) + A(I)
S-19     ENDDO
S-20     !$OMP END DO NOWAIT
S-21     !$OMP END PARALLEL
S-22     END SUBROUTINE NOWAIT_EXAMPLE2

```

1 1.7 The collapse Clause

2 In the following example, the **k** and **j** loops are associated with the loop construct. So the iterations
3 of the **k** and **j** loops are collapsed into one loop with a larger iteration space, and that loop is then
4 divided among the threads in the current team. Since the **i** loop is not associated with the loop
5 construct, it is not collapsed, and the **i** loop is executed sequentially in its entirety in every iteration
6 of the collapsed **k** and **j** loop.

7 The variable **j** can be omitted from the **private** clause when the **collapse** clause is used since
8 it is implicitly private. However, if the **collapse** clause is omitted then **j** will be shared if it is
9 omitted from the **private** clause. In either case, **k** is implicitly private and could be omitted from
10 the **private** clause.

▼ C / C++ ▼

11 *Example collapse.1.c*

```
S-1 void bar(float *a, int i, int j, int k);  
S-2 int kl, ku, ks, jl, ju, js, il, iu, is;  
S-3 void sub(float *a)  
S-4 {  
S-5     int i, j, k;  
S-6     #pragma omp for collapse(2) private(i, k, j)  
S-7     for (k=kl; k<=ku; k+=ks)  
S-8         for (j=jl; j<=ju; j+=js)  
S-9             for (i=il; i<=iu; i+=is)  
S-10                 bar(a,i,j,k);  
S-11 }
```

▲ C / C++ ▲
▼ Fortran ▼

12 *Example collapse.1.f*

```
S-1     subroutine sub(a)  
S-2     real a(*)  
S-3     integer kl, ku, ks, jl, ju, js, il, iu, is  
S-4     common /csub/ kl, ku, ks, jl, ju, js, il, iu, is  
S-5     integer i, j, k  
S-6     !$omp do collapse(2) private(i,j,k)  
S-7         do k = kl, ku, ks  
S-8             do j = jl, ju, js  
S-9                 do i = il, iu, is  
S-10                     call bar(a,i,j,k)  
S-11                 enddo  
S-12             enddo  
S-13         enddo  
S-14     !$omp end do  
S-15     end subroutine
```

Fortran

In the next example, the **k** and **j** loops are associated with the loop construct. So the iterations of the **k** and **j** loops are collapsed into one loop with a larger iteration space, and that loop is then divided among the threads in the current team.

The sequential execution of the iterations in the **k** and **j** loops determines the order of the iterations in the collapsed iteration space. This implies that in the sequentially last iteration of the collapsed iteration space, **k** will have the value 2 and **j** will have the value 3. Since **klast** and **jlast** are **lastprivate**, their values are assigned by the sequentially last iteration of the collapsed **k** and **j** loop. This example prints: 2 3.

C / C++

Example collapse.2.c

```
S-1  #include <stdio.h>
S-2  void test()
S-3  {
S-4      int j, k, jlast, klast;
S-5      #pragma omp parallel
S-6      {
S-7          #pragma omp for collapse(2) lastprivate(jlast, klast)
S-8          for (k=1; k<=2; k++)
S-9              for (j=1; j<=3; j++)
S-10             {
S-11                 jlast=j;
S-12                 klast=k;
S-13             }
S-14         #pragma omp single
S-15         printf("%d %d\n", klast, jlast);
S-16     }
S-17 }
```

C / C++

Example collapse.2.f

```

S-1      program test
S-2      !$omp parallel
S-3      !$omp do private(j,k) collapse(2) lastprivate(jlast, klast)
S-4          do k = 1,2
S-5              do j = 1,3
S-6                  jlast=j
S-7                  klast=k
S-8              enddo
S-9          enddo
S-10     !$omp end do
S-11     !$omp single
S-12         print *, klast, jlast
S-13     !$omp end single
S-14     !$omp end parallel
S-15     end program test

```

The next example illustrates the interaction of the **collapse** and **ordered** clauses.

In the example, the loop construct has both a **collapse** clause and an **ordered** clause. The **collapse** clause causes the iterations of the **k** and **j** loops to be collapsed into one loop with a larger iteration space, and that loop is divided among the threads in the current team. An **ordered** clause is added to the loop construct, because an ordered region binds to the loop region arising from the loop construct.

According to Section 2.12.8 of the OpenMP 4.0 specification, a thread must not execute more than one ordered region that binds to the same loop region. So the **collapse** clause is required for the example to be conforming. With the **collapse** clause, the iterations of the **k** and **j** loops are collapsed into one loop, and therefore only one ordered region will bind to the collapsed **k** and **j** loop. Without the **collapse** clause, there would be two ordered regions that bind to each iteration of the **k** loop (one arising from the first iteration of the **j** loop, and the other arising from the second iteration of the **j** loop).

The code prints

```

0 1 1
0 1 2
0 2 1
1 2 2
1 3 1
1 3 2

```

1

Example collapse.3.c

```

S-1  #include <omp.h>
S-2  #include <stdio.h>
S-3  void work(int a, int j, int k);
S-4  void sub()
S-5  {
S-6      int j, k, a;
S-7      #pragma omp parallel num_threads(2)
S-8      {
S-9          #pragma omp for collapse(2) ordered private(j,k) schedule(static,3)
S-10         for (k=1; k<=3; k++)
S-11             for (j=1; j<=2; j++)
S-12                 {
S-13                     #pragma omp ordered
S-14                     printf("%d %d %d\n", omp_get_thread_num(), k, j);
S-15                     /* end ordered */
S-16                     work(a, j, k);
S-17                 }
S-18     }
S-19 }

```

2

Example collapse.3.f

```

S-1      program test
S-2      include 'omp_lib.h'
S-3      !$omp parallel num_threads(2)
S-4      !$omp do collapse(2) ordered private(j,k) schedule(static,3)
S-5          do k = 1,3
S-6              do j = 1,2
S-7                  !$omp ordered
S-8                      print *, omp_get_thread_num(), k, j
S-9                  !$omp end ordered
S-10                     call work(a, j, k)
S-11                 enddo
S-12             enddo
S-13         !$omp end do
S-14     !$omp end parallel
S-15     end program test

```

1 1.8 linear Clause in Loop Constructs

2 The following example shows the use of the **linear** clause in a loop construct to allow the proper
3 parallelization of a loop that contains an induction variable (j). At the end of the execution of the
4 loop construct, the original variable j is updated with the value $N/2$ from the last iteration of the
5 loop.

▼ C / C++ ▼

6 *Example linear_in_loop.1.c*

```
S-1 #include <stdio.h>
S-2
S-3 #define N 100
S-4 int main(void)
S-5 {
S-6     float a[N], b[N/2];
S-7     int i, j;
S-8
S-9     for ( i = 0; i < N; i++ )
S-10         a[i] = i + 1;
S-11
S-12     j = 0;
S-13     #pragma omp parallel
S-14     #pragma omp for linear(j:1)
S-15     for ( i = 0; i < N; i += 2 ) {
S-16         b[j] = a[i] * 2.0f;
S-17         j++;
S-18     }
S-19
S-20     printf( "%d %f %f\n", j, b[0], b[j-1] );
S-21     /* print out: 50 2.0 198.0 */
S-22
S-23     return 0;
S-24 }
```

▲ C / C++ ▲

1

Example linear_in_loop.1.f90

```

S-1  program linear_loop
S-2      implicit none
S-3      integer, parameter :: N = 100
S-4      real :: a(N), b(N/2)
S-5      integer :: i, j
S-6
S-7      do i = 1, N
S-8          a(i) = i
S-9      end do
S-10
S-11      j = 0
S-12      !$omp parallel
S-13      !$omp do linear(j:1)
S-14      do i = 1, N, 2
S-15          j = j + 1
S-16          b(j) = a(i) * 2.0
S-17      end do
S-18      !$omp end parallel
S-19
S-20      print *, j, b(1), b(j)
S-21      ! print out: 50 2.0 198.0
S-22
S-23  end program

```


1.9 The parallel sections Construct

In the following example routines **XAXIS**, **YAXIS**, and **ZAXIS** can be executed concurrently. The first **section** directive is optional. Note that all **section** directives need to appear in the **parallel sections** construct.

C / C++

Example psections.1.c

```
S-1 void XAXIS();
S-2 void YAXIS();
S-3 void ZAXIS();
S-4
S-5 void sect_example()
S-6 {
S-7     #pragma omp parallel sections
S-8     {
S-9         #pragma omp section
S-10        XAXIS();
S-11
S-12        #pragma omp section
S-13        YAXIS();
S-14
S-15        #pragma omp section
S-16        ZAXIS();
S-17    }
S-18 }
```

C / C++

Fortran

Example psections.1.f

```
S-1 SUBROUTINE SECT_EXAMPLE()
S-2 !$OMP PARALLEL SECTIONS
S-3 !$OMP SECTION
S-4     CALL XAXIS()
S-5 !$OMP SECTION
S-6     CALL YAXIS()
S-7
S-8 !$OMP SECTION
S-9     CALL ZAXIS()
S-10
S-11 !$OMP END PARALLEL SECTIONS
S-12 END SUBROUTINE SECT_EXAMPLE
```

Fortran

1.10 The `firstprivate` Clause and the `sections` Construct

In the following example of the `sections` construct the `firstprivate` clause is used to initialize the private copy of `section_count` of each thread. The problem is that the `section` constructs modify `section_count`, which breaks the independence of the `section` constructs. When different threads execute each section, both sections will print the value 1. When the same thread executes the two sections, one section will print the value 1 and the other will print the value 2. Since the order of execution of the two sections in this case is unspecified, it is unspecified which section prints which value.

C / C++

Example `fpriv_sections.1.c`

```
S-1  #include <omp.h>
S-2  #include <stdio.h>
S-3  #define NT 4
S-4  int main( ) {
S-5      int section_count = 0;
S-6      omp_set_dynamic(0);
S-7      omp_set_num_threads(NT);
S-8      #pragma omp parallel
S-9      #pragma omp sections firstprivate( section_count )
S-10     {
S-11     #pragma omp section
S-12     {
S-13         section_count++;
S-14         /* may print the number one or two */
S-15         printf( "section_count %d\n", section_count );
S-16     }
S-17     #pragma omp section
S-18     {
S-19         section_count++;
S-20         /* may print the number one or two */
S-21         printf( "section_count %d\n", section_count );
S-22     }
S-23 }
S-24     return 0;
S-25 }
```

C / C++

1

Example fpriv_sections.1.f90

```

S-1  program section
S-2      use omp_lib
S-3      integer :: section_count = 0
S-4      integer, parameter :: NT = 4
S-5      call omp_set_dynamic(.false.)
S-6      call omp_set_num_threads(NT)
S-7      !$omp parallel
S-8      !$omp sections firstprivate ( section_count )
S-9      !$omp section
S-10         section_count = section_count + 1
S-11      ! may print the number one or two
S-12         print *, 'section_count', section_count
S-13      !$omp section
S-14         section_count = section_count + 1
S-15      ! may print the number one or two
S-16         print *, 'section_count', section_count
S-17      !$omp end sections
S-18      !$omp end parallel
S-19  end program section

```

1 1.11 The single Construct

2 The following example demonstrates the **single** construct. In the example, only one thread prints
3 each of the progress messages. All other threads will skip the **single** region and stop at the
4 barrier at the end of the **single** construct until all threads in the team have reached the barrier. If
5 other threads can proceed without waiting for the thread executing the **single** region, a **nowait**
6 clause can be specified, as is done in the third **single** construct in this example. The user must
7 not make any assumptions as to which thread will execute a **single** region.

C / C++

8 *Example single.1.c*

```
S-1 #include <stdio.h>
S-2
S-3 void work1() {}
S-4 void work2() {}
S-5
S-6 void single_example()
S-7 {
S-8     #pragma omp parallel
S-9     {
S-10         #pragma omp single
S-11             printf("Beginning work1.\n");
S-12
S-13         work1();
S-14
S-15         #pragma omp single
S-16             printf("Finishing work1.\n");
S-17
S-18         #pragma omp single nowait
S-19             printf("Finished work1 and beginning work2.\n");
S-20
S-21         work2();
S-22     }
S-23 }
```

C / C++

1

Example single.1.f

```

S-1      SUBROUTINE WORK1 ()
S-2      END SUBROUTINE WORK1
S-3
S-4      SUBROUTINE WORK2 ()
S-5      END SUBROUTINE WORK2
S-6
S-7      PROGRAM SINGLE_EXAMPLE
S-8      !$OMP PARALLEL
S-9
S-10     !$OMP SINGLE
S-11         print *, "Beginning work1."
S-12     !$OMP END SINGLE
S-13
S-14         CALL WORK1 ()
S-15
S-16     !$OMP SINGLE
S-17         print *, "Finishing work1."
S-18     !$OMP END SINGLE
S-19
S-20     !$OMP SINGLE
S-21         print *, "Finished work1 and beginning work2."
S-22     !$OMP END SINGLE NOWAIT
S-23
S-24         CALL WORK2 ()
S-25
S-26     !$OMP END PARALLEL
S-27
S-28     END PROGRAM SINGLE_EXAMPLE

```

1.12 The workshare Construct

Fortran

The following are examples of the **workshare** construct.

In the following example, **workshare** spreads work across the threads executing the **parallel** region, and there is a barrier after the last statement. Implementations must enforce Fortran execution rules inside of the **workshare** block.

Example workshare.1.f

```
S-1      SUBROUTINE WSHARE1(AA, BB, CC, DD, EE, FF, N)
S-2      INTEGER N
S-3      REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N), EE(N,N), FF(N,N)
S-4
S-5      !$OMP    PARALLEL
S-6      !$OMP    WORKSHARE
S-7          AA = BB
S-8          CC = DD
S-9          EE = FF
S-10     !$OMP    END WORKSHARE
S-11     !$OMP    END PARALLEL
S-12
S-13     END SUBROUTINE WSHARE1
```

In the following example, the barrier at the end of the first **workshare** region is eliminated with a **nowait** clause. Threads doing **CC = DD** immediately begin work on **EE = FF** when they are done with **CC = DD**.

Example workshare.2.f

```
S-1      SUBROUTINE WSHARE2(AA, BB, CC, DD, EE, FF, N)
S-2      INTEGER N
S-3      REAL AA(N,N), BB(N,N), CC(N,N)
S-4      REAL DD(N,N), EE(N,N), FF(N,N)
S-5
S-6      !$OMP    PARALLEL
S-7      !$OMP    WORKSHARE
S-8          AA = BB
S-9          CC = DD
S-10     !$OMP    END WORKSHARE NOWAIT
S-11     !$OMP    WORKSHARE
S-12          EE = FF
S-13     !$OMP    END WORKSHARE
S-14     !$OMP    END PARALLEL
S-15     END SUBROUTINE WSHARE2
```

The following example shows the use of an **atomic** directive inside a **workshare** construct. The computation of **SUM(AA)** is workshared, but the update to **R** is atomic.

Example workshare.3.f

```

S-1      SUBROUTINE WSHARE3(AA, BB, CC, DD, N)
S-2      INTEGER N
S-3      REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
S-4      REAL R
S-5      R=0
S-6      !$OMP PARALLEL
S-7      !$OMP WORKSHARE
S-8          AA = BB
S-9      !$OMP ATOMIC UPDATE
S-10         R = R + SUM(AA)
S-11         CC = DD
S-12      !$OMP END WORKSHARE
S-13      !$OMP END PARALLEL
S-14      END SUBROUTINE WSHARE3

```

Fortran **WHERE** and **FORALL** statements are *compound statements*, made up of a *control* part and a *statement* part. When **workshare** is applied to one of these compound statements, both the control and the statement parts are workshared. The following example shows the use of a **WHERE** statement in a **workshare** construct.

Each task gets worked on in order by the threads:

```

AA = BB then
CC = DD then
EE .ne. 0 then
FF = 1 / EE then
GG = HH

```

Example workshare.4.f

```

S-1      SUBROUTINE WSHARE4(AA, BB, CC, DD, EE, FF, GG, HH, N)
S-2      INTEGER N
S-3      REAL AA(N,N), BB(N,N), CC(N,N)
S-4      REAL DD(N,N), EE(N,N), FF(N,N)
S-5      REAL GG(N,N), HH(N,N)
S-6
S-7      !$OMP PARALLEL
S-8      !$OMP WORKSHARE
S-9          AA = BB
S-10         CC = DD
S-11         WHERE (EE .ne. 0) FF = 1 / EE

```

```

S-12          GG = HH
S-13  !$OMP    END WORKSHARE
S-14  !$OMP    END PARALLEL
S-15
S-16          END SUBROUTINE WSHARE4

```

In the following example, an assignment to a shared scalar variable is performed by one thread in a **workshare** while all other threads in the team wait.

Example workshare.5.f

```

S-1          SUBROUTINE WSHARE5(AA, BB, CC, DD, N)
S-2          INTEGER N
S-3          REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
S-4
S-5          INTEGER SHR
S-6
S-7  !$OMP    PARALLEL SHARED(SHR)
S-8  !$OMP    WORKSHARE
S-9          AA = BB
S-10         SHR = 1
S-11         CC = DD * SHR
S-12  !$OMP    END WORKSHARE
S-13  !$OMP    END PARALLEL
S-14
S-15          END SUBROUTINE WSHARE5

```

The following example contains an assignment to a private scalar variable, which is performed by one thread in a **workshare** while all other threads wait. It is non-conforming because the private scalar variable is undefined after the assignment statement.

Example workshare.6.f

```

S-1          SUBROUTINE WSHARE6_WRONG(AA, BB, CC, DD, N)
S-2          INTEGER N
S-3          REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
S-4
S-5          INTEGER PRI
S-6
S-7  !$OMP    PARALLEL PRIVATE(PRI)
S-8  !$OMP    WORKSHARE
S-9          AA = BB
S-10         PRI = 1
S-11         CC = DD * PRI
S-12  !$OMP    END WORKSHARE
S-13  !$OMP    END PARALLEL

```


S-14

S-15 **END SUBROUTINE WSHARE6_WRONG**

1 Fortran execution rules must be enforced inside a **workshare** construct. In the following
2 example, the same result is produced in the following program fragment regardless of whether the
3 code is executed sequentially or inside an OpenMP program with multiple threads:

4 *Example workshare.7.f*

S-1 **SUBROUTINE WSHARE7(AA, BB, CC, N)**

S-2 **INTEGER N**

S-3 **REAL AA(N), BB(N), CC(N)**

S-4

S-5 **!\$OMP PARALLEL**

S-6 **!\$OMP WORKSHARE**

S-7 **AA(1:50) = BB(11:60)**

S-8 **CC(11:20) = AA(1:10)**

S-9 **!\$OMP END WORKSHARE**

S-10 **!\$OMP END PARALLEL**

S-11

S-12 **END SUBROUTINE WSHARE7**

Fortran

1.13 The master Construct

The following example demonstrates the master construct . In the example, the master keeps track of how many iterations have been executed and prints out a progress report. The other threads skip the master region without waiting.

C / C++

Example master.1.c

```
S-1  #include <stdio.h>
S-2
S-3  extern float average(float,float,float);
S-4
S-5  void master_example( float* x, float* xold, int n, float tol )
S-6  {
S-7      int c, i, toobig;
S-8      float error, y;
S-9      c = 0;
S-10     #pragma omp parallel
S-11     {
S-12         do{
S-13             #pragma omp for private(i)
S-14             for( i = 1; i < n-1; ++i ){
S-15                 xold[i] = x[i];
S-16             }
S-17             #pragma omp single
S-18             {
S-19                 toobig = 0;
S-20             }
S-21             #pragma omp for private(i,y,error) reduction(+:toobig)
S-22             for( i = 1; i < n-1; ++i ){
S-23                 y = x[i];
S-24                 x[i] = average( xold[i-1], x[i], xold[i+1] );
S-25                 error = y - x[i];
S-26                 if( error > tol || error < -tol ) ++toobig;
S-27             }
S-28             #pragma omp master
S-29             {
S-30                 ++c;
S-31                 printf( "iteration %d, toobig=%d\n", c, toobig );
S-32             }
S-33         }while( toobig > 0 );
S-34     }
S-35 }
```

C / C++

1

Example master.1.f

```

S-1      SUBROUTINE MASTER_EXAMPLE( X, XOLD, N, TOL )
S-2      REAL X(*), XOLD(*), TOL
S-3      INTEGER N
S-4      INTEGER C, I, TOOBIG
S-5      REAL ERROR, Y, AVERAGE
S-6      EXTERNAL AVERAGE
S-7      C = 0
S-8      TOOBIG = 1
S-9      !$OMP PARALLEL
S-10         DO WHILE( TOOBIG > 0 )
S-11      !$OMP      DO PRIVATE(I)
S-12                 DO I = 2, N-1
S-13                     XOLD(I) = X(I)
S-14                 ENDDO
S-15      !$OMP      SINGLE
S-16                 TOOBIG = 0
S-17      !$OMP      END SINGLE
S-18      !$OMP      DO PRIVATE(I,Y,ERROR), REDUCTION(+:TOOBIG)
S-19                 DO I = 2, N-1
S-20                     Y = X(I)
S-21                     X(I) = AVERAGE( XOLD(I-1), X(I), XOLD(I+1) )
S-22                     ERROR = Y-X(I)
S-23                     IF( ERROR > TOL .OR. ERROR < -TOL ) TOOBIG = TOOBIG+1
S-24                 ENDDO
S-25      !$OMP      MASTER
S-26                 C = C + 1
S-27                 PRINT *, 'Iteration ', C, 'TOOBIG=', TOOBIG
S-28      !$OMP      END MASTER
S-29      ENDDO
S-30      !$OMP END PARALLEL
S-31      END SUBROUTINE MASTER_EXAMPLE

```

1 1.14 Parallel Random Access Iterator Loop

C++

2 The following example shows a parallel random access iterator loop.

3 *Example pra_iterator.1.cpp*

```
S-1 #include <vector>
S-2 void iterator_example()
S-3 {
S-4     std::vector<int> vec(23);
S-5     std::vector<int>::iterator it;
S-6     #pragma omp parallel for default(none) shared(vec)
S-7     for (it = vec.begin(); it < vec.end(); it++)
S-8     {
S-9         // do work with *it //
S-10    }
S-11 }
```

C++

1 1.15 The `omp_set_dynamic` and 2 `omp_set_num_threads` Routines

3 Some programs rely on a fixed, prespecified number of threads to execute correctly. Because the
4 default setting for the dynamic adjustment of the number of threads is implementation defined, such
5 programs can choose to turn off the dynamic threads capability and set the number of threads
6 explicitly to ensure portability. The following example shows how to do this using
7 `omp_set_dynamic`, and `omp_set_num_threads`.

8 In this example, the program executes correctly only if it is executed by 16 threads. If the
9 implementation is not capable of supporting 16 threads, the behavior of this example is
10 implementation defined. Note that the number of threads executing a **parallel** region remains
11 constant during the region, regardless of the dynamic threads setting. The dynamic threads
12 mechanism determines the number of threads to use at the start of the **parallel** region and keeps
13 it constant for the duration of the region.

▼ C / C++ ▼

14 *Example `set_dynamic_nthrs.1.c`*

```
S-1 #include <omp.h>
S-2 #include <stdlib.h>
S-3
S-4 void do_by_16(float *x, int iam, int ipoints) {}
S-5
S-6 void dynthreads(float *x, int npoints)
S-7 {
S-8     int iam, ipoints;
S-9
S-10    omp_set_dynamic(0);
S-11    omp_set_num_threads(16);
S-12
S-13    #pragma omp parallel shared(x, npoints) private(iam, ipoints)
S-14    {
S-15        if (omp_get_num_threads() != 16)
S-16            abort();
S-17
S-18        iam = omp_get_thread_num();
S-19        ipoints = npoints/16;
S-20        do_by_16(x, iam, ipoints);
S-21    }
S-22 }
```

▲ C / C++ ▲

1

Example set_dynamic_nthr.f

```

S-1      SUBROUTINE DO_BY_16(X, IAM, IPOINTS)
S-2          REAL X(*)
S-3          INTEGER IAM, IPOINTS
S-4      END SUBROUTINE DO_BY_16

S-5
S-6      SUBROUTINE DYNTHREADS(X, NPOINTS)
S-7
S-8          INCLUDE "omp_lib.h"          ! or USE OMP_LIB
S-9
S-10         INTEGER NPOINTS
S-11         REAL X(NPOINTS)
S-12
S-13         INTEGER IAM, IPOINTS
S-14
S-15         CALL OMP_SET_DYNAMIC(.FALSE.)
S-16         CALL OMP_SET_NUM_THREADS(16)
S-17
S-18     !$OMP  PARALLEL SHARED(X,NPOINTS) PRIVATE(IAM, IPOINTS)
S-19
S-20         IF (OMP_GET_NUM_THREADS() .NE. 16) THEN
S-21             STOP
S-22         ENDIF
S-23
S-24         IAM = OMP_GET_THREAD_NUM()
S-25         IPOINTS = NPOINTS/16
S-26         CALL DO_BY_16(X, IAM, IPOINTS)
S-27
S-28     !$OMP  END PARALLEL
S-29
S-30     END SUBROUTINE DYNTHREADS

```

1 1.16 The `omp_get_num_threads` Routine

2 In the following example, the `omp_get_num_threads` call returns 1 in the sequential part of
3 the code, so `np` will always be equal to 1. To determine the number of threads that will be deployed
4 for the `parallel` region, the call should be inside the `parallel` region.

▼ C / C++ ▼

5 *Example `get_nthrs.l.c`*

```
S-1 #include <omp.h>
S-2 void work(int i);
S-3
S-4 void incorrect()
S-5 {
S-6     int np, i;
S-7
S-8     np = omp_get_num_threads(); /* misplaced */
S-9
S-10    #pragma omp parallel for schedule(static)
S-11    for (i=0; i < np; i++)
S-12        work(i);
S-13 }
```

▲ C / C++ ▲
▼ Fortran ▼

6 *Example `get_nthrs.l.f`*

```
S-1      SUBROUTINE WORK(I)
S-2      INTEGER I
S-3      I = I + 1
S-4      END SUBROUTINE WORK
S-5
S-6      SUBROUTINE INCORRECT()
S-7          INCLUDE "omp_lib.h"      ! or USE OMP_LIB
S-8          INTEGER I, NP
S-9
S-10         NP = OMP_GET_NUM_THREADS() !misplaced: will return 1
S-11     !$OMP  PARALLEL DO SCHEDULE(STATIC)
S-12         DO I = 0, NP-1
S-13             CALL WORK(I)
S-14         ENDDO
S-15     !$OMP  END PARALLEL DO
S-16     END SUBROUTINE INCORRECT
```

Fortran

The following example shows how to rewrite this program without including a query for the number of threads:

C / C++

Example get_nthrs.2.c

```
S-1  #include <omp.h>
S-2  void work(int i);
S-3
S-4  void correct()
S-5  {
S-6      int i;
S-7
S-8      #pragma omp parallel private(i)
S-9      {
S-10         i = omp_get_thread_num();
S-11         work(i);
S-12     }
S-13 }
```

C / C++

Fortran

Example get_nthrs.2.f

```
S-1      SUBROUTINE WORK(I)
S-2          INTEGER I
S-3
S-4          I = I + 1
S-5
S-6      END SUBROUTINE WORK
S-7
S-8      SUBROUTINE CORRECT()
S-9          INCLUDE "omp_lib.h"      ! or USE OMP_LIB
S-10         INTEGER I
S-11
S-12      !$OMP    PARALLEL PRIVATE(I)
S-13          I = OMP_GET_THREAD_NUM()
S-14          CALL WORK(I)
S-15      !$OMP    END PARALLEL
S-16
S-17      END SUBROUTINE CORRECT
```

Fortran

2 OpenMP Affinity

3 OpenMP Affinity consists of a **proc_bind** policy (thread affinity policy) and a specification of
4 places ("location units" or *processors* that may be cores, hardware threads, sockets, etc.). OpenMP
5 Affinity enables users to bind computations on specific places. The placement will hold for the
6 duration of the parallel region. However, the runtime is free to migrate the OpenMP threads to
7 different cores (hardware threads, sockets, etc.) prescribed within a given place, if two or more
8 cores (hardware threads, sockets, etc.) have been assigned to a given place.

9 Often the binding can be managed without resorting to explicitly setting places. Without the
10 specification of places in the **OMP_PLACES** variable, the OpenMP runtime will distribute and bind
11 threads using the entire range of processors for the OpenMP program, according to the
12 **OMP_PROC_BIND** environment variable or the **proc_bind** clause. When places are specified,
13 the OMP runtime binds threads to the places according to a default distribution policy, or those
14 specified in the **OMP_PROC_BIND** environment variable or the **proc_bind** clause.

15 In the OpenMP Specifications document a processor refers to an execution unit that is enabled for
16 an OpenMP thread to use. A processor is a core when there is no SMT (Simultaneous
17 Multi-Threading) support or SMT is disabled. When SMT is enabled, a processor is a hardware
18 thread (HW-thread). (This is the usual case; but actually, the execution unit is implementation
19 defined.) Processor numbers are numbered sequentially from 0 to the number of cores less one
20 (without SMT), or 0 to the number HW-threads less one (with SMT). OpenMP places use the
21 processor number to designate binding locations (unless an "abstract name" is used.)

22 The processors available to a process may be a subset of the system's processors. This restriction
23 may be the result of a wrapper process controlling the execution (such as **numactl** on Linux
24 systems), compiler options, library-specific environment variables, or default kernel settings. For
25 instance, the execution of multiple MPI processes, launched on a single compute node, will each
26 have a subset of processors as determined by the MPI launcher or set by MPI affinity environment
27 variables for the MPI library.

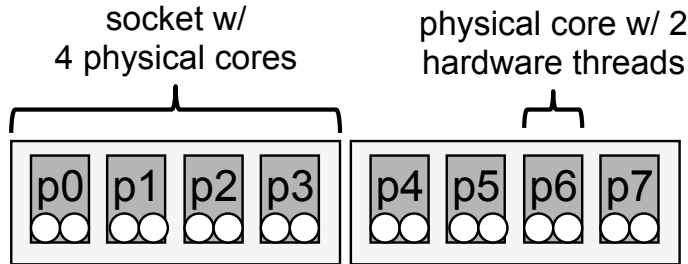
28 Threads of a team are positioned onto places in a compact manner, a scattered distribution, or onto
29 the master's place, by setting the **OMP_PROC_BIND** environment variable or the **proc_bind**

1 clause to *close*, *spread*, or *master*, respectively. When **OMP_PROC_BIND** is set to FALSE no
2 binding is enforced; and when the value is TRUE, the binding is implementation defined to a set of
3 places in the **OMP_PLACES** variable or to places defined by the implementation if the
4 **OMP_PLACES** variable is not set.

5 The **OMP_PLACES** variable can also be set to an abstract name (*threads*, *cores*, *sockets*) to specify
6 that a place is either a single hardware thread, a core, or a socket, respectively. This description of
7 the **OMP_PLACES** is most useful when the number of threads is equal to the number of hardware
8 thread, cores or sockets. It can also be used with a *close* or *spread* distribution policy when the
9 equality doesn't hold.

1 2.1 The `proc_bind` Clause

2 The following examples demonstrate how to use the **`proc_bind`** clause to control the thread
3 binding for a team of threads in a **`parallel`** region. The machine architecture is depicted in the
4 figure below. It consists of two sockets, each equipped with a quad-core processor and configured
5 to execute two hardware threads simultaneously on each core. These examples assume a contiguous
6 core numbering starting from 0, such that the hardware threads 0,1 form the first physical core.



7 The following equivalent place list declarations consist of eight places (which we designate as p0 to
8 p7):

9 **`OMP_PLACES="{0,1},{2,3},{4,5},{6,7},{8,9},{10,11},{12,13},{14,15}"`**

10 or

11 **`OMP_PLACES="{0:2}:8:2"`**

12 2.1.1 Spread Affinity Policy

13 The following example shows the result of the **`spread`** affinity policy on the partition list when the
14 number of threads is less than or equal to the number of places in the parent's place partition, for
15 the machine architecture depicted above. Note that the threads are bound to the first place of each
16 subpartition.

17  C / C++

Example affinity.1.c

```
S-1 void work();  
S-2 int main()  
S-3 {  
S-4 #pragma omp parallel proc_bind(spread) num_threads(4)  
S-5 {  
S-6     work();  
S-7 }  
S-8 return 0;  
S-9 }
```

C / C++

Fortran

Example affinity.1.f

```
S-1      PROGRAM EXAMPLE
S-2      !$OMP PARALLEL PROC_BIND (SPREAD) NUM_THREADS (4)
S-3      CALL WORK ()
S-4      !$OMP END PARALLEL
S-5      END PROGRAM EXAMPLE
```

Fortran

It is unspecified on which place the master thread is initially started. If the master thread is initially started on p0, the following placement of threads will be applied in the parallel region:

- thread 0 executes on p0 with the place partition p0,p1
- thread 1 executes on p2 with the place partition p2,p3
- thread 2 executes on p4 with the place partition p4,p5
- thread 3 executes on p6 with the place partition p6,p7

If the master thread would initially be started on p2, the placement of threads and distribution of the place partition would be as follows:

- thread 0 executes on p2 with the place partition p2,p3
- thread 1 executes on p4 with the place partition p4,p5
- thread 2 executes on p6 with the place partition p6,p7
- thread 3 executes on p0 with the place partition p0,p1

The following example illustrates the **spread** thread affinity policy when the number of threads is greater than the number of places in the parent's place partition.

Let T be the number of threads in the team, and P be the number of places in the parent's place partition. The first T/P threads of the team (including the master thread) execute on the parent's place. The next T/P threads execute on the next place in the place partition, and so on, with wrap around.

C / C++

Example affinity.2.c

```
S-1      void work();
S-2      void foo()
S-3      {
S-4          #pragma omp parallel num_threads(16) proc_bind(spread)
S-5          {
S-6              work();
S-7          }
S-8      }
```

C / C++

Fortran

Example affinity.2.f90

```

S-1  subroutine foo
S-2  !$omp parallel num_threads(16) proc_bind(spread)
S-3      call work()
S-4  !$omp end parallel
S-5  end subroutine

```

Fortran

It is unspecified on which place the master thread is initially started. If the master thread is initially started on p0, the following placement of threads will be applied in the parallel region:

- threads 0,1 execute on p0 with the place partition p0
- threads 2,3 execute on p1 with the place partition p1
- threads 4,5 execute on p2 with the place partition p2
- threads 6,7 execute on p3 with the place partition p3
- threads 8,9 execute on p4 with the place partition p4
- threads 10,11 execute on p5 with the place partition p5
- threads 12,13 execute on p6 with the place partition p6
- threads 14,15 execute on p7 with the place partition p7

If the master thread would initially be started on p2, the placement of threads and distribution of the place partition would be as follows:

- threads 0,1 execute on p2 with the place partition p2
- threads 2,3 execute on p3 with the place partition p3
- threads 4,5 execute on p4 with the place partition p4
- threads 6,7 execute on p5 with the place partition p5
- threads 8,9 execute on p6 with the place partition p6
- threads 10,11 execute on p7 with the place partition p7
- threads 12,13 execute on p0 with the place partition p0
- threads 14,15 execute on p1 with the place partition p1

2.1.2 Close Affinity Policy

The following example shows the result of the **close** affinity policy on the partition list when the number of threads is less than or equal to the number of places in parent's place partition, for the machine architecture depicted above. The place partition is not changed by the **close** policy.

1 *Example affinity.3.c*

```

S-1 void work();
S-2 int main()
S-3 {
S-4 #pragma omp parallel proc_bind(close) num_threads(4)
S-5 {
S-6     work();
S-7 }
S-8 return 0;
S-9 }

```

2 *Example affinity.3.f*

```

S-1      PROGRAM EXAMPLE
S-2      !$OMP PARALLEL PROC_BIND(CLOSE) NUM_THREADS(4)
S-3      CALL WORK()
S-4      !$OMP END PARALLEL
S-5      END PROGRAM EXAMPLE

```

3 It is unspecified on which place the master thread is initially started. If the master thread is initially
 4 started on p0, the following placement of threads will be applied in the **parallel** region:

- 5 • thread 0 executes on p0 with the place partition p0-p7
- 6 • thread 1 executes on p1 with the place partition p0-p7
- 7 • thread 2 executes on p2 with the place partition p0-p7
- 8 • thread 3 executes on p3 with the place partition p0-p7

9 If the master thread would initially be started on p2, the placement of threads and distribution of the
 10 place partition would be as follows:

- 11 • thread 0 executes on p2 with the place partition p0-p7
- 12 • thread 1 executes on p3 with the place partition p0-p7
- 13 • thread 2 executes on p4 with the place partition p0-p7
- 14 • thread 3 executes on p5 with the place partition p0-p7

15 The following example illustrates the **close** thread affinity policy when the number of threads is
 16 greater than the number of places in the parent's place partition.

17 Let T be the number of threads in the team, and P be the number of places in the parent's place
 18 partition. The first T/P threads of the team (including the master thread) execute on the parent's
 19 place. The next T/P threads execute on the next place in the place partition, and so on, with wrap
 20 around. The place partition is not changed by the **close** policy.

1 *Example affinity.4.c*

```

S-1 void work();
S-2 void foo()
S-3 {
S-4     #pragma omp parallel num_threads(16) proc_bind(close)
S-5     {
S-6         work();
S-7     }
S-8 }

```

2 *Example affinity.4.f90*

```

S-1 subroutine foo
S-2 !$omp parallel num_threads(16) proc_bind(close)
S-3     call work()
S-4 !$omp end parallel
S-5 end subroutine

```

It is unspecified on which place the master thread is initially started. If the master thread is initially running on p0, the following placement of threads will be applied in the parallel region:

- threads 0,1 execute on p0 with the place partition p0-p7
- threads 2,3 execute on p1 with the place partition p0-p7
- threads 4,5 execute on p2 with the place partition p0-p7
- threads 6,7 execute on p3 with the place partition p0-p7
- threads 8,9 execute on p4 with the place partition p0-p7
- threads 10,11 execute on p5 with the place partition p0-p7
- threads 12,13 execute on p6 with the place partition p0-p7
- threads 14,15 execute on p7 with the place partition p0-p7

If the master thread would initially be started on p2, the placement of threads and distribution of the place partition would be as follows:

- threads 0,1 execute on p2 with the place partition p0-p7
- threads 2,3 execute on p3 with the place partition p0-p7
- threads 4,5 execute on p4 with the place partition p0-p7
- threads 6,7 execute on p5 with the place partition p0-p7
- threads 8,9 execute on p6 with the place partition p0-p7
- threads 10,11 execute on p7 with the place partition p0-p7
- threads 12,13 execute on p0 with the place partition p0-p7
- threads 14,15 execute on p1 with the place partition p0-p7

2.1.3 Master Affinity Policy

The following example shows the result of the **master** affinity policy on the partition list for the machine architecture depicted above. The place partition is not changed by the master policy.

C / C++

Example affinity.5.c

```
S-1 void work();
S-2 int main()
S-3 {
S-4 #pragma omp parallel proc_bind(master) num_threads(4)
S-5 {
S-6     work();
S-7 }
S-8 return 0;
S-9 }
```

C / C++

Fortran

Example affinity.5.f

```
S-1 PROGRAM EXAMPLE
S-2 !$OMP PARALLEL PROC_BIND(MASTER) NUM_THREADS(4)
S-3 CALL WORK()
S-4 !$OMP END PARALLEL
S-5 END PROGRAM EXAMPLE
```

Fortran

It is unspecified on which place the master thread is initially started. If the master thread is initially running on p0, the following placement of threads will be applied in the parallel region:

- threads 0-3 execute on p0 with the place partition p0-p7

If the master thread would initially be started on p2, the placement of threads and distribution of the place partition would be as follows:

- threads 0-3 execute on p2 with the place partition p0-p7

2.2 Affinity Query Functions

In the example below a team of threads is generated on each socket of the system, using nested parallelism. Several query functions are used to gather information to support the creation of the teams and to obtain socket and thread numbers.

For proper execution of the code, the user must create a place partition, such that each place is a listing of the core numbers for a socket. For example, in a 2 socket system with 8 cores in each socket, and sequential numbering in the socket for the core numbers, the **OMP_PLACES** variable would be set to "{0:8},{8:8}", using the place syntax {*lower_bound:length:stride*}, and the default stride of 1.

The code determines the number of sockets (*n_sockets*) using the **omp_get_num_places()** query function. In this example each place is constructed with a list of each socket's core numbers, hence the number of places is equal to the number of sockets.

The outer parallel region forms a team of threads, and each thread executes on a socket (place) because the **proc_bind** clause uses **spread** in the outer **parallel** construct. Next, in the *socket_init* function, an inner parallel region creates a team of threads equal to the number of elements (core numbers) from the place of the parent thread. Because the outer **parallel** construct uses a **spread** affinity policy, each of its threads inherits a subpartition of the original partition. Hence, the **omp_get_place_num_procs** query function returns the number of elements (here procs = cores) in the subpartition of the thread. After each parent thread creates its nested parallel region on the section, the socket number and thread number are reported.

Note: Portable tools like hwloc (Portable HardWare LOcality package), which support many common operating systems, can be used to determine the configuration of a system. On some systems there are utilities, files or user guides that provide configuration information. For instance, the socket number and *proc_id*'s for a socket can be found in the */proc/cpuinfo* text file on Linux systems.

C / C++

Example affinity.6.c

```
S-1
S-2  #include <stdio.h>
S-3  #include <omp.h>
S-4
S-5  void socket_init(int socket_num)
S-6  {
S-7      int n_procs;
S-8
S-9      n_procs = omp_get_place_num_procs(socket_num);
S-10     #pragma omp parallel num_threads(n_procs) proc_bind(close)
S-11     {
S-12         printf("Reporting in from socket num, thread num:  %d %d\n",
```

```

S-13                                     socket_num,omp_get_thread_num() );
S-14     }
S-15 }
S-16
S-17 int main()
S-18 {
S-19     int n_sockets, socket_num;
S-20
S-21     omp_set_nested(1);                // or export OMP_NESTED=true
S-22     omp_set_max_active_levels(2);    // or export OMP_MAX_ACTIVE_LEVELS=2
S-23
S-24     n_sockets = omp_get_num_places();
S-25     #pragma omp parallel num_threads(n_sockets) private(socket_num) \
S-26                 proc_bind(spread)
S-27     {
S-28         socket_num = omp_get_place_num();
S-29         socket_init(socket_num);
S-30     }
S-31 }

```



1 *Example affinity.6.f90*

```

S-1
S-2 subroutine socket_init(socket_num)
S-3     use omp_lib
S-4     integer :: socket_num, n_procs
S-5
S-6     n_procs = omp_get_place_num_procs(socket_num)
S-7     !$omp parallel num_threads(n_procs) proc_bind(close)
S-8
S-9         print*, "Reporting in from socket num, thread num: ", &
S-10                                socket_num,omp_get_thread_num()
S-11     !$omp end parallel
S-12 end subroutine
S-13
S-14 program numa_teams
S-15     use omp_lib
S-16     integer :: n_sockets, socket_num
S-17
S-18     call omp_set_nested(.true.)          ! or export OMP_NESTED=true
S-19     call omp_set_max_active_levels(2) ! or export OMP_MAX_ACTIVE_LEVELS=2
S-20
S-21     n_sockets = omp_get_num_places()
S-22     !$omp parallel num_threads(n_sockets) private(socket_num) &
S-23     !$omp&         proc_bind(spread)

```

```
S-24
S-25      socket_num = omp_get_place_num()
S-26      call socket_init(socket_num)
S-27
S-28      !$omp end parallel
S-29 end program
```

Fortran

2

Tasking

3 Tasking constructs provide units of work to a thread for execution. Worksharing constructs do this,
4 too (e.g. **for**, **do**, **sections**, and **singles** constructs); but the work units are tightly controlled
5 by an iteration limit and limited scheduling, or a limited number of **sections** or **single**
6 regions. Worksharing was designed with "data parallel" computing in mind. Tasking was designed
7 for "task parallel" computing and often involves non-locality or irregularity in memory access.

8 The **task** construct can be used to execute work chunks: in a while loop; while traversing nodes in
9 a list; at nodes in a tree graph; or in a normal loop (with a **taskloop** construct). Unlike the
10 statically scheduled loop iterations of worksharing, a task is often enqueued, and then dequeued for
11 execution by any of the threads of the team within a parallel region. The generation of tasks can be
12 from a single generating thread (creating sibling tasks), or from multiple generators in a recursive
13 graph tree traversals. A **taskloop** construct bundles iterations of an associated loop into tasks,
14 and provides similar controls found in the **task** construct.

15 Sibling tasks are synchronized by the **taskwait** construct, and tasks and their descendent tasks
16 can be synchronized by containing them in a **taskgroup** region. Ordered execution is
17 accomplished by specifying dependences with a **depend** clause. Also, priorities can be specified
18 as hints to the scheduler through a **priority** clause.

19 Various clauses can be used to manage and optimize task generation, as well as reduce the overhead
20 of execution and to relinquish control of threads for work balance and forward progress.

21 Once a thread starts executing a task, it is the designated thread for executing the task to
22 completion, even though it may leave the execution at a scheduling point and return later. The
23 thread is tied to the task. Scheduling points can be introduced with the **taskyield** construct.
24 With an **untied** clause any other thread is allowed to continue the task. An **if** clause with a *true*
25 expression allows the generating thread to immediately execute the task as an undeferred task. By
26 including the data environment of the generating task into the generated task with the **mergeable**
27 and **final** clauses, task generation overhead can be reduced.

28 A complete list of the tasking constructs and details of their clauses can be found in the *Tasking*
29 *Constructs* chapter of the OpenMP Specifications, in the *OpenMP Application Programming*
30 *Interface* section.

1 3.1 The task and taskwait Constructs

2 The following example shows how to traverse a tree-like structure using explicit tasks. Note that the
3 **traverse** function should be called from within a parallel region for the different specified tasks
4 to be executed in parallel. Also note that the tasks will be executed in no specified order because
5 there are no synchronization directives. Thus, assuming that the traversal will be done in post order,
6 as in the sequential code, is wrong.

▼ C / C++ ▼

7 *Example tasking.1.c*

```
S-1 struct node {  
S-2     struct node *left;  
S-3     struct node *right;  
S-4 };  
S-5 extern void process(struct node *);  
S-6 void traverse( struct node *p ) {  
S-7     if (p->left)  
S-8     #pragma omp task    // p is firstprivate by default  
S-9         traverse(p->left);  
S-10    if (p->right)  
S-11    #pragma omp task    // p is firstprivate by default  
S-12        traverse(p->right);  
S-13    process(p);  
S-14 }
```

▲ C / C++ ▲
▼ Fortran ▼

8 *Example tasking.1.f90*

```
S-1 RECURSIVE SUBROUTINE traverse ( P )  
S-2     TYPE Node  
S-3         TYPE(Node), POINTER :: left, right  
S-4     END TYPE Node  
S-5     TYPE(Node) :: P  
S-6     IF (associated(P%left)) THEN  
S-7         !$OMP TASK    ! P is firstprivate by default  
S-8         CALL traverse(P%left)  
S-9         !$OMP END TASK  
S-10    ENDIF  
S-11    IF (associated(P%right)) THEN  
S-12        !$OMP TASK    ! P is firstprivate by default  
S-13        CALL traverse(P%right)  
S-14        !$OMP END TASK  
S-15    ENDIF  
S-16    CALL process ( P )  
S-17 END SUBROUTINE
```

Fortran

In the next example, we force a postorder traversal of the tree by adding a **taskwait** directive. Now, we can safely assume that the left and right sons have been executed before we process the current node.

C / C++

Example tasking.2.c

```
S-1 struct node {
S-2     struct node *left;
S-3     struct node *right;
S-4 };
S-5 extern void process(struct node *);
S-6 void postorder_traverse( struct node *p ) {
S-7     if (p->left)
S-8         #pragma omp task // p is firstprivate by default
S-9         postorder_traverse(p->left);
S-10    if (p->right)
S-11        #pragma omp task // p is firstprivate by default
S-12        postorder_traverse(p->right);
S-13    #pragma omp taskwait
S-14    process(p);
S-15 }
```

C / C++

Fortran

Example tasking.2.f90

```
S-1 RECURSIVE SUBROUTINE traverse ( P )
S-2     TYPE Node
S-3         TYPE(Node), POINTER :: left, right
S-4     END TYPE Node
S-5     TYPE(Node) :: P
S-6     IF (associated(P%left)) THEN
S-7         !$OMP TASK ! P is firstprivate by default
S-8         CALL traverse(P%left)
S-9         !$OMP END TASK
S-10    ENDIF
S-11    IF (associated(P%right)) THEN
S-12        !$OMP TASK ! P is firstprivate by default
S-13        CALL traverse(P%right)
S-14        !$OMP END TASK
S-15    ENDIF
S-16    !$OMP TASKWAIT
S-17    CALL process ( P )
S-18 END SUBROUTINE
```

Fortran

The following example demonstrates how to use the **task** construct to process elements of a linked list in parallel. The thread executing the **single** region generates all of the explicit tasks, which are then executed by the threads in the current team. The pointer *p* is **firstprivate** by default on the **task** construct so it is not necessary to specify it in a **firstprivate** clause.

C / C++

Example tasking.3.c

```
S-1  typedef struct node node;
S-2  struct node {
S-3      int data;
S-4      node * next;
S-5  };
S-6
S-7  void process(node * p)
S-8  {
S-9      /* do work here */
S-10 }
S-11 void increment_list_items(node * head)
S-12 {
S-13     #pragma omp parallel
S-14     {
S-15         #pragma omp single
S-16         {
S-17             node * p = head;
S-18             while (p) {
S-19                 #pragma omp task
S-20                 // p is firstprivate by default
S-21                 process(p);
S-22                 p = p->next;
S-23             }
S-24         }
S-25     }
S-26 }
```

C / C++

Example tasking.3.f90

```

S-1      MODULE LIST
S-2          TYPE NODE
S-3              INTEGER :: PAYLOAD
S-4              TYPE (NODE), POINTER :: NEXT
S-5          END TYPE NODE
S-6      CONTAINS
S-7          SUBROUTINE PROCESS(p)
S-8              TYPE (NODE), POINTER :: P
S-9              ! do work here
S-10         END SUBROUTINE
S-11         SUBROUTINE INCREMENT_LIST_ITEMS (HEAD)
S-12             TYPE (NODE), POINTER :: HEAD
S-13             TYPE (NODE), POINTER :: P
S-14             !$OMP PARALLEL PRIVATE(P)
S-15                 !$OMP SINGLE
S-16                     P => HEAD
S-17                     DO
S-18                         !$OMP TASK
S-19                             ! P is firstprivate by default
S-20                             CALL PROCESS(P)
S-21                         !$OMP END TASK
S-22                     P => P%NEXT
S-23                     IF ( .NOT. ASSOCIATED (P) ) EXIT
S-24                 END DO
S-25             !$OMP END SINGLE
S-26         !$OMP END PARALLEL
S-27     END SUBROUTINE
S-28 END MODULE

```

The **fib()** function should be called from within a **parallel** region for the different specified tasks to be executed in parallel. Also, only one thread of the **parallel** region should call **fib()** unless multiple concurrent Fibonacci computations are desired.

1

Example tasking.4.c

```

S-1      int fib(int n) {
S-2          int i, j;
S-3          if (n<2)
S-4              return n;
S-5          else {
S-6              #pragma omp task shared(i)
S-7                  i=fib(n-1);
S-8              #pragma omp task shared(j)
S-9                  j=fib(n-2);
S-10             #pragma omp taskwait
S-11                 return i+j;
S-12         }
S-13     }

```

2

Example tasking.4.f

```

S-1      RECURSIVE INTEGER FUNCTION fib(n) RESULT(res)
S-2          INTEGER n, i, j
S-3          IF ( n .LT. 2) THEN
S-4              res = n
S-5          ELSE
S-6              !$OMP TASK SHARED(i)
S-7                  i = fib( n-1 )
S-8              !$OMP END TASK
S-9              !$OMP TASK SHARED(j)
S-10                 j = fib( n-2 )
S-11             !$OMP END TASK
S-12             !$OMP TASKWAIT
S-13                 res = i+j
S-14             END IF
S-15         END FUNCTION

```

3

Note: There are more efficient algorithms for computing Fibonacci numbers. This classic recursion algorithm is for illustrative purposes.

4

5

The following example demonstrates a way to generate a large number of tasks with one thread and execute them with the threads in the team. While generating these tasks, the implementation may reach its limit on unassigned tasks. If it does, the implementation is allowed to cause the thread executing the task generating loop to suspend its task at the task scheduling point in the **task** directive, and start executing unassigned tasks. Once the number of unassigned tasks is sufficiently low, the thread may resume execution of the task generating loop.

6

7

8

9

10

1

Example tasking.5.c

```
S-1  #define LARGE_NUMBER 10000000
S-2  double item[LARGE_NUMBER];
S-3  extern void process(double);
S-4
S-5  int main() {
S-6  #pragma omp parallel
S-7  {
S-8      #pragma omp single
S-9      {
S-10         int i;
S-11         for (i=0; i<LARGE_NUMBER; i++)
S-12             #pragma omp task // i is firstprivate, item is shared
S-13                 process(item[i]);
S-14     }
S-15 }
S-16 }
```

Example tasking.5.f

```

S-1      real*8 item(10000000)
S-2      integer i
S-3
S-4      !$omp parallel
S-5      !$omp single ! loop iteration variable i is private
S-6      do i=1,10000000
S-7      !$omp task
S-8          ! i is firstprivate, item is shared
S-9          call process(item(i))
S-10     !$omp end task
S-11     end do
S-12     !$omp end single
S-13     !$omp end parallel
S-14     end

```

The following example is the same as the previous one, except that the tasks are generated in an untied task. While generating the tasks, the implementation may reach its limit on unassigned tasks. If it does, the implementation is allowed to cause the thread executing the task generating loop to suspend its task at the task scheduling point in the **task** directive, and start executing unassigned tasks. If that thread begins execution of a task that takes a long time to complete, the other threads may complete all the other tasks before it is finished.

In this case, since the loop is in an untied task, any other thread is eligible to resume the task generating loop. In the previous examples, the other threads would be forced to idle until the generating thread finishes its long task, since the task generating loop was in a tied task.

Example tasking.6.c

```

S-1      #define LARGE_NUMBER 10000000
S-2      double item[LARGE_NUMBER];
S-3      extern void process(double);
S-4      int main() {
S-5      #pragma omp parallel
S-6      {
S-7          #pragma omp single
S-8          {
S-9              int i;
S-10             #pragma omp task untied
S-11             // i is firstprivate, item is shared
S-12             {
S-13                 for (i=0; i<LARGE_NUMBER; i++)
S-14                     #pragma omp task

```

```

S-15         process(item[i]);
S-16     }
S-17 }
S-18 }
S-19 return 0;
S-20 }

```

1 *Example tasking.6.f*

```

S-1         real*8 item(10000000)
S-2 !$omp parallel
S-3 !$omp single
S-4 !$omp task untied
S-5         ! loop iteration variable i is private
S-6         do i=1,10000000
S-7 !$omp task ! i is firstprivate, item is shared
S-8         call process(item(i))
S-9 !$omp end task
S-10        end do
S-11 !$omp end task
S-12 !$omp end single
S-13 !$omp end parallel
S-14        end

```

2 The following two examples demonstrate how the scheduling rules illustrated in Section 2.11.3 of
3 the OpenMP 4.0 specification affect the usage of **threadprivate** variables in tasks. A
4 **threadprivate** variable can be modified by another task that is executed by the same thread.
5 Thus, the value of a **threadprivate** variable cannot be assumed to be unchanged across a task
6 scheduling point. In untied tasks, task scheduling points may be added in any place by the
7 implementation.

8 A task switch may occur at a task scheduling point. A single thread may execute both of the task
9 regions that modify **tp**. The parts of these task regions in which **tp** is modified may be executed in
10 any order so the resulting value of **var** can be either 1 or 2.

1

Example tasking.7.c

```

S-1
S-2  int tp;
S-3  #pragma omp threadprivate(tp)
S-4  int var;
S-5  void work()
S-6  {
S-7  #pragma omp task
S-8      {
S-9          /* do work here */
S-10 #pragma omp task
S-11     {
S-12         tp = 1;
S-13         /* do work here */
S-14 #pragma omp task
S-15     {
S-16         /* no modification of tp */
S-17     }
S-18         var = tp; //value of tp can be 1 or 2
S-19     }
S-20     tp = 2;
S-21 }
S-22 }

```

2

Example tasking.7.f

```

S-1      module example
S-2      integer tp
S-3  !$omp threadprivate(tp)
S-4      integer var
S-5      contains
S-6      subroutine work
S-7  !$omp task
S-8      ! do work here
S-9  !$omp task
S-10         tp = 1
S-11         ! do work here
S-12  !$omp task
S-13         ! no modification of tp
S-14  !$omp end task
S-15         var = tp      ! value of var can be 1 or 2
S-16  !$omp end task
S-17         tp = 2

```

```

S-18  !$omp end task
S-19      end subroutine
S-20  end module

```

Fortran

In this example, scheduling constraints prohibit a thread in the team from executing a new task that modifies **tp** while another such task region tied to the same thread is suspended. Therefore, the value written will persist across the task scheduling point.

C / C++

Example tasking.8.c

```

S-1
S-2  int tp;
S-3  #pragma omp threadprivate(tp)
S-4  int var;
S-5  void work()
S-6  {
S-7  #pragma omp parallel
S-8      {
S-9          /* do work here */
S-10 #pragma omp task
S-11     {
S-12         tp++;
S-13         /* do work here */
S-14 #pragma omp task
S-15     {
S-16         /* do work here but don't modify tp */
S-17     }
S-18     var = tp; //Value does not change after write above
S-19     }
S-20 }
S-21 }

```

C / C++

Fortran

Example tasking.8.f

```
S-1      module example
S-2      integer tp
S-3      !$omp threadprivate(tp)
S-4      integer var
S-5      contains
S-6      subroutine work
S-7      !$omp parallel
S-8          ! do work here
S-9      !$omp task
S-10         tp = tp + 1
S-11         ! do work here
S-12      !$omp task
S-13         ! do work here but don't modify tp
S-14      !$omp end task
S-15         var = tp      ! value does not change after write above
S-16      !$omp end task
S-17      !$omp end parallel
S-18      end subroutine
S-19      end module
```

Fortran

The following two examples demonstrate how the scheduling rules illustrated in Section 2.11.3 of the OpenMP 4.0 specification affect the usage of locks and critical sections in tasks. If a lock is held across a task scheduling point, no attempt should be made to acquire the same lock in any code that may be interleaved. Otherwise, a deadlock is possible.

In the example below, suppose the thread executing task 1 defers task 2. When it encounters the task scheduling point at task 3, it could suspend task 1 and begin task 2 which will result in a deadlock when it tries to enter critical region 1.

C / C++

Example tasking.9.c

```
S-1      void work()
S-2      {
S-3          #pragma omp task
S-4          { //Task 1
S-5              #pragma omp task
S-6              { //Task 2
S-7                  #pragma omp critical //Critical region 1
S-8                      { /*do work here */ }
S-9              }
S-10             #pragma omp critical //Critical Region 2
S-11             {
```

```

S-12          //Capture data for the following task
S-13          #pragma omp task
S-14          { /* do work here */ } //Task 3
S-15          }
S-16      }
S-17  }

```

1 *Example tasking.9.f*

```

S-1          module example
S-2          contains
S-3          subroutine work
S-4      !$omp task
S-5          ! Task 1
S-6      !$omp task
S-7          ! Task 2
S-8      !$omp critical
S-9          ! Critical region 1
S-10         ! do work here
S-11      !$omp end critical
S-12      !$omp end task
S-13      !$omp critical
S-14         ! Critical region 2
S-15         ! Capture data for the following task
S-16      !$omp task
S-17          !Task 3
S-18          ! do work here
S-19      !$omp end task
S-20      !$omp end critical
S-21      !$omp end task
S-22          end subroutine
S-23      end module

```

2 In the following example, **lock** is held across a task scheduling point. However, according to the
 3 scheduling restrictions, the executing thread can't begin executing one of the non-descendant tasks
 4 that also acquires **lock** before the task region is complete. Therefore, no deadlock is possible.

1

Example tasking.10.c

```

S-1  #include <omp.h>
S-2  void work() {
S-3      omp_lock_t lock;
S-4      omp_init_lock(&lock);
S-5      #pragma omp parallel
S-6      {
S-7          int i;
S-8      #pragma omp for
S-9          for (i = 0; i < 100; i++) {
S-10         #pragma omp task
S-11             {
S-12                 // lock is shared by default in the task
S-13                 omp_set_lock(&lock);
S-14                 // Capture data for the following task
S-15         #pragma omp task
S-16             // Task Scheduling Point 1
S-17                 { /* do work here */ }
S-18                 omp_unset_lock(&lock);
S-19             }
S-20         }
S-21     }
S-22     omp_destroy_lock(&lock);
S-23 }

```

2

Example tasking.10.f90

```

S-1      module example
S-2      include 'omp_lib.h'
S-3      integer (kind=omp_lock_kind) lock
S-4      integer i
S-5
S-6      contains
S-7
S-8      subroutine work
S-9      call omp_init_lock(lock)
S-10     !$omp parallel
S-11         !$omp do
S-12             do i=1,100
S-13                 !$omp task
S-14                     ! Outer task
S-15                     call omp_set_lock(lock)      ! lock is shared by
S-16                                                     ! default in the task

```

```

S-17             ! Capture data for the following task
S-18             !$omp task      ! Task Scheduling Point 1
S-19             ! do work here
S-20             !$omp end task
S-21             call omp_unset_lock(lock)
S-22             !$omp end task
S-23         end do
S-24     !$omp end parallel
S-25     call omp_destroy_lock(lock)
S-26     end subroutine
S-27
S-28     end module

```

Fortran

The following examples illustrate the use of the **mergeable** clause in the **task** construct. In this first example, the **task** construct has been annotated with the **mergeable** clause. The addition of this clause allows the implementation to reuse the data environment (including the ICVs) of the parent task for the task inside **foo** if the task is included or undeferred. Thus, the result of the execution may differ depending on whether the task is merged or not. Therefore the mergeable clause needs to be used with caution. In this example, the use of the mergeable clause is safe. As **x** is a shared variable the outcome does not depend on whether or not the task is merged (that is, the task will always increment the same variable and will always compute the same value for **x**).

C / C++

Example tasking.11.c

```

S-1  #include <stdio.h>
S-2  void foo ( )
S-3  {
S-4      int x = 2;
S-5      #pragma omp task shared(x) mergeable
S-6      {
S-7          x++;
S-8      }
S-9      #pragma omp taskwait
S-10     printf("%d\n",x); // prints 3
S-11 }

```

C / C++

Fortran

Example tasking.11.f90

```
S-1  subroutine foo()  
S-2      integer :: x  
S-3      x = 2  
S-4      !$omp task shared(x) mergeable  
S-5          x = x + 1  
S-6      !$omp end task  
S-7      !$omp taskwait  
S-8      print *, x      ! prints 3  
S-9  end subroutine
```

Fortran

This second example shows an incorrect use of the **mergeable** clause. In this example, the created task will access different instances of the variable **x** if the task is not merged, as **x** is **firstprivate**, but it will access the same variable **x** if the task is merged. As a result, the behavior of the program is unspecified and it can print two different values for **x** depending on the decisions taken by the implementation.

C / C++

Example tasking.12.c

```
S-1  #include <stdio.h>  
S-2  void foo ( )  
S-3  {  
S-4      int x = 2;  
S-5      #pragma omp task mergeable  
S-6      {  
S-7          x++;  
S-8      }  
S-9      #pragma omp taskwait  
S-10     printf("%d\n",x); // prints 2 or 3  
S-11 }
```

C / C++

Example *tasking.12.f90*

```

S-1  subroutine foo()
S-2      integer :: x
S-3      x = 2
S-4      !$omp task mergeable
S-5          x = x + 1
S-6      !$omp end task
S-7      !$omp taskwait
S-8      print *, x    ! prints 2 or 3
S-9  end subroutine

```

The following example shows the use of the **final** clause and the **omp_in_final** API call in a recursive binary search program. To reduce overhead, once a certain depth of recursion is reached the program uses the **final** clause to create only included tasks, which allow additional optimizations.

The use of the **omp_in_final** API call allows programmers to optimize their code by specifying which parts of the program are not necessary when a task can create only included tasks (that is, the code is inside a **final** task). In this example, the use of a different state variable is not necessary so once the program reaches the part of the computation that is finalized and copying from the parent state to the new state is eliminated. The allocation of **new_state** in the stack could also be avoided but it would make this example less clear. The **final** clause is most effective when used in conjunction with the **mergeable** clause since all tasks created in a **final** task region are included tasks that can be merged if the **mergeable** clause is present.

Example *tasking.13.c*

```

S-1  #include <string.h>
S-2  #include <omp.h>
S-3  #define LIMIT  3 /* arbitrary limit on recursion depth */
S-4  void check_solution(char *);
S-5  void bin_search (int pos, int n, char *state)
S-6  {
S-7      if ( pos == n ) {
S-8          check_solution(state);
S-9          return;
S-10     }
S-11     #pragma omp task final( pos > LIMIT ) mergeable
S-12     {
S-13         char new_state[n];
S-14         if (!omp_in_final() ) {
S-15             memcpy(new_state, state, pos );

```

```

S-16         state = new_state;
S-17     }
S-18     state[pos] = 0;
S-19     bin_search(pos+1, n, state );
S-20 }
S-21 #pragma omp task final( pos > LIMIT ) mergeable
S-22 {
S-23     char new_state[n];
S-24     if (! omp_in_final() ) {
S-25         memcpy(new_state, state, pos );
S-26         state = new_state;
S-27     }
S-28     state[pos] = 1;
S-29     bin_search(pos+1, n, state );
S-30 }
S-31 #pragma omp taskwait
S-32 }

```



1

Example tasking.13.f90

```

S-1 recursive subroutine bin_search(pos, n, state)
S-2     use omp_lib
S-3     integer :: pos, n
S-4     character, pointer :: state(:)
S-5     character, target, dimension(n) :: new_state1, new_state2
S-6     integer, parameter :: LIMIT = 3
S-7     if (pos .eq. n) then
S-8         call check_solution(state)
S-9         return
S-10    endif
S-11    !$omp task final(pos > LIMIT) mergeable
S-12    if (.not. omp_in_final()) then
S-13        new_state1(1:pos) = state(1:pos)
S-14        state => new_state1
S-15    endif
S-16    state(pos+1) = 'z'
S-17    call bin_search(pos+1, n, state)
S-18    !$omp end task
S-19    !$omp task final(pos > LIMIT) mergeable
S-20    if (.not. omp_in_final()) then
S-21        new_state2(1:pos) = state(1:pos)
S-22        state => new_state2
S-23    endif
S-24    state(pos+1) = 'y'
S-25    call bin_search(pos+1, n, state)

```

```

S-26  !$omp end task
S-27  !$omp taskwait
S-28  end subroutine

```

Fortran

1 The following example illustrates the difference between the **if** and the **final** clauses. The **if**
 2 clause has a local effect. In the first nest of tasks, the one that has the **if** clause will be undeferred
 3 but the task nested inside that task will not be affected by the **if** clause and will be created as usual.
 4 Alternatively, the **final** clause affects all **task** constructs in the **final** task region but not the
 5 **final** task itself. In the second nest of tasks, the nested tasks will be created as included tasks.
 6 Note also that the conditions for the **if** and **final** clauses are usually the opposite.

C / C++

7 *Example tasking.14.c*

```

S-1  void bar(void);
S-2
S-3  void foo ( )
S-4  {
S-5      int i;
S-6      #pragma omp task if(0)  // This task is undeferred
S-7      {
S-8          #pragma omp task    // This task is a regular task
S-9          for (i = 0; i < 3; i++) {
S-10             #pragma omp task    // This task is a regular task
S-11             bar();
S-12         }
S-13     }
S-14     #pragma omp task final(1) // This task is a regular task
S-15     {
S-16         #pragma omp task // This task is included
S-17         for (i = 0; i < 3; i++) {
S-18             #pragma omp task    // This task is also included
S-19             bar();
S-20         }
S-21     }
S-22 }

```

C / C++

1

Example tasking.14.f90

```

S-1  subroutine foo()
S-2  integer i
S-3  !$omp task if(.FALSE.) ! This task is undeferred
S-4  !$omp task             ! This task is a regular task
S-5      do i = 1, 3
S-6          !$omp task             ! This task is a regular task
S-7              call bar()
S-8          !$omp end task
S-9      enddo
S-10 !$omp end task
S-11 !$omp end task
S-12 !$omp task final(.TRUE.) ! This task is a regular task
S-13 !$omp task             ! This task is included
S-14      do i = 1, 3
S-15          !$omp task             ! This task is also included
S-16              call bar()
S-17          !$omp end task
S-18      enddo
S-19 !$omp end task
S-20 !$omp end task
S-21 end subroutine

```

1 3.2 Task Priority

2 In this example we compute arrays in a matrix through a *compute_array* routine. Each task has a
3 priority value equal to the value of the loop variable *i* at the moment of its creation. A higher
4 priority on a task means that a task is a candidate to run sooner.

5 The creation of tasks occurs in ascending order (according to the iteration space of the loop) but a
6 hint, by means of the **priority** clause, is provided to reverse the execution order.

▼ C / C++ ▼

7 *Example task_priority.I.c*

```
S-1 void compute_array (float *node, int M);  
S-2  
S-3 void compute_matrix (float *array, int N, int M)  
S-4 {  
S-5     int i;  
S-6     #pragma omp parallel private(i)  
S-7     #pragma omp single  
S-8     {  
S-9         for (i=0;i<N; i++) {  
S-10             #pragma omp task priority(i)  
S-11             compute_array(&array[i*M], M);  
S-12         }  
S-13     }  
S-14 }
```

▲ C / C++ ▲

▼ Fortran ▼

8 *Example task_priority.I.f90*

```
S-1 subroutine compute_matrix(matrix, M, N)  
S-2     implicit none  
S-3     integer :: M, N  
S-4     real :: matrix(M, N)  
S-5     integer :: i  
S-6     interface  
S-7         subroutine compute_array(node, M)  
S-8             implicit none  
S-9             integer :: M  
S-10            real :: node(M)  
S-11        end subroutine  
S-12    end interface  
S-13    !$omp parallel private(i)  
S-14    !$omp single  
S-15    do i=1,N  
S-16        !$omp task priority(i)
```



```
S-17      call compute_array(matrix(:, i), M)
S-18      !$omp end task
S-19  enddo
S-20      !$omp end single
S-21      !$omp end parallel
S-22  end subroutine compute_matrix
```

Fortran

1 3.3 Task Dependences

2 3.3.1 Flow Dependence

3 In this example we show a simple flow dependence expressed using the **depend** clause on the
4 **task** construct.

C / C++

5 *Example task_dep.1.c*

```
S-1 #include <stdio.h>
S-2 int main()
S-3 {
S-4     int x = 1;
S-5     #pragma omp parallel
S-6     #pragma omp single
S-7     {
S-8         #pragma omp task shared(x) depend(out: x)
S-9         x = 2;
S-10        #pragma omp task shared(x) depend(in: x)
S-11        printf("x = %d\n", x);
S-12    }
S-13    return 0;
S-14 }
```

C / C++

Fortran

6 *Example task_dep.1.f90*

```
S-1 program example
S-2     integer :: x
S-3     x = 1
S-4     !$omp parallel
S-5     !$omp single
S-6         !$omp task shared(x) depend(out: x)
S-7         x = 2
S-8     !$omp end task
S-9         !$omp task shared(x) depend(in: x)
S-10        print*, "x = ", x
S-11    !$omp end task
S-12    !$omp end single
S-13    !$omp end parallel
S-14 end program
```

The program will always print "x = 2", because the **depend** clauses enforce the ordering of the tasks. If the **depend** clauses had been omitted, then the tasks could execute in any order and the program and the program would have a race condition.

3.3.2 Anti-dependence

In this example we show an anti-dependence expressed using the **depend** clause on the **task** construct.

Example task_dep.2.c

```

S-1  #include <stdio.h>
S-2  int main()
S-3  {
S-4      int x = 1;
S-5      #pragma omp parallel
S-6      #pragma omp single
S-7      {
S-8          #pragma omp task shared(x) depend(in: x)
S-9          printf("x = %d\n", x);
S-10         #pragma omp task shared(x) depend(out: x)
S-11         x = 2;
S-12     }
S-13     return 0;
S-14 }
```

Example task_dep.2.f90

```

S-1  program example
S-2      integer :: x
S-3      x = 1
S-4      !$omp parallel
S-5          !$omp single
S-6              !$omp task shared(x) depend(in: x)
S-7                  print*, "x = ", x
S-8              !$omp end task
S-9              !$omp task shared(x) depend(out: x)
S-10                  x = 2
S-11              !$omp end task
S-12          !$omp end single
S-13      !$omp end parallel
S-14  end program

```

The program will always print "x = 1", because the **depend** clauses enforce the ordering of the tasks. If the **depend** clauses had been omitted, then the tasks could execute in any order and the program would have a race condition.

3.3.3 Output Dependence

In this example we show an output dependence expressed using the **depend** clause on the **task** construct.

Example task_dep.3.c

```

S-1  #include <stdio.h>
S-2  int main()
S-3  {
S-4      int x;
S-5      #pragma omp parallel
S-6          #pragma omp single
S-7          {
S-8              #pragma omp task shared(x) depend(out: x)
S-9                  x = 1;
S-10              #pragma omp task shared(x) depend(out: x)
S-11                  x = 2;
S-12              #pragma omp taskwait

```

```

S-13     printf("x = %d\n", x);
S-14     }
S-15     return 0;
S-16     }

```

C / C++

Fortran

1

Example task_dep.3.f90

```

S-1  program example
S-2      integer :: x
S-3      !$omp parallel
S-4      !$omp single
S-5          !$omp task shared(x) depend(out: x)
S-6              x = 1
S-7          !$omp end task
S-8          !$omp task shared(x) depend(out: x)
S-9              x = 2
S-10         !$omp end task
S-11         !$omp taskwait
S-12         print*, "x = ", x
S-13     !$omp end single
S-14     !$omp end parallel
S-15 end program

```

Fortran

2

The program will always print "x = 2", because the **depend** clauses enforce the ordering of the tasks. If the **depend** clauses had been omitted, then the tasks could execute in any order and the program would have a race condition.

3

4

5 3.3.4 Concurrent Execution with Dependences

6

In this example we show potentially concurrent execution of tasks using multiple flow dependences expressed using the **depend** clause on the **task** construct.

7

1 *Example task_dep.4.c*

```

S-1  #include <stdio.h>
S-2  int main()
S-3  {
S-4      int x = 1;
S-5      #pragma omp parallel
S-6      #pragma omp single
S-7      {
S-8          #pragma omp task shared(x) depend(out: x)
S-9              x = 2;
S-10         #pragma omp task shared(x) depend(in: x)
S-11             printf("x + 1 = %d. ", x+1);
S-12         #pragma omp task shared(x) depend(in: x)
S-13             printf("x + 2 = %d\n", x+2);
S-14     }
S-15     return 0;
S-16 }

```

2 *Example task_dep.4.f90*

```

S-1  program example
S-2      integer :: x
S-3      x = 1
S-4      !$omp parallel
S-5      !$omp single
S-6          !$omp task shared(x) depend(out: x)
S-7              x = 2
S-8          !$omp end task
S-9          !$omp task shared(x) depend(in: x)
S-10              print*, "x + 1 = ", x+1, "."
S-11          !$omp end task
S-12          !$omp task shared(x) depend(in: x)
S-13              print*, "x + 2 = ", x+2, "."
S-14          !$omp end task
S-15      !$omp end single
S-16      !$omp end parallel
S-17  end program

```

The last two tasks are dependent on the first task. However there is no dependence between the last two tasks, which may execute in any order (or concurrently if more than one thread is available). Thus, the possible outputs are "x + 1 = 3. x + 2 = 4. " and "x + 2 = 4. x + 1 = 3. ". If the **depend** clauses had been omitted, then all of the tasks could execute in any order and the program would have a race condition.

6 3.3.5 Matrix multiplication

This example shows a task-based blocked matrix multiplication. Matrices are of NxN elements, and the multiplication is implemented using blocks of BSxBS elements.

C / C++

Example task_dep.5.c

```

S-1 // Assume BS divides N perfectly
S-2 void matmul_depend(int N, int BS, float A[N][N], float B[N][N], float
S-3 C[N][N] )
S-4 {
S-5     int i, j, k, ii, jj, kk;
S-6     for (i = 0; i < N; i+=BS) {
S-7         for (j = 0; j < N; j+=BS) {
S-8             for (k = 0; k < N; k+=BS) {
S-9 // Note 1: i, j, k, A, B, C are firstprivate by default
S-10 // Note 2: A, B and C are just pointers
S-11 #pragma omp task private(ii, jj, kk) \
S-12     depend ( in: A[i:BS][k:BS], B[k:BS][j:BS] ) \
S-13     depend ( inout: C[i:BS][j:BS] )
S-14     for (ii = i; ii < i+BS; ii++ )
S-15         for (jj = j; jj < j+BS; jj++ )
S-16             for (kk = k; kk < k+BS; kk++ )
S-17                 C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
S-18         }
S-19     }
S-20 }
S-21 }
```

C / C++

1

Example task_dep.5.f90

```

S-1  ! Assume BS divides N perfectly
S-2  subroutine matmul_depend (N, BS, A, B, C)
S-3      implicit none
S-4      integer :: N, BS, BM
S-5      real, dimension(N, N) :: A, B, C
S-6      integer :: i, j, k, ii, jj, kk
S-7      BM = BS - 1
S-8      do i = 1, N, BS
S-9          do j = 1, N, BS
S-10             do k = 1, N, BS
S-11                 !$omp task shared(A,B,C) private(ii,jj,kk) & ! I,J,K are firstprivate by default
S-12                 !$omp depend ( in: A(i:i+BM, k:k+BM), B(k:k+BM, j:j+BM) ) &
S-13                 !$omp depend ( inout: C(i:i+BM, j:j+BM) )
S-14                     do ii = i, i+BM
S-15                         do jj = j, j+BM
S-16                             do kk = k, k+BM
S-17                                 C(jj,ii) = C(jj,ii) + A(kk,ii) * B(jj,kk)
S-18                             end do
S-19                         end do
S-20                     end do
S-21                 !$omp end task
S-22             end do
S-23         end do
S-24     end do
S-25 end subroutine

```


1 3.4 The taskgroup Construct

2 In this example, tasks are grouped and synchronized using the **taskgroup** construct.

3 Initially, one task (the task executing the **start_background_work()** call) is created in the
4 **parallel** region, and later a parallel tree traversal is started (the task executing the root of the
5 recursive **compute_tree()** calls). While synchronizing tasks at the end of each tree traversal,
6 using the **taskgroup** construct ensures that the formerly started background task does not
7 participate in the synchronization, and is left free to execute in parallel. This is opposed to the
8 behaviour of the **taskwait** construct, which would include the background tasks in the
9 synchronization.

C / C++

10 *Example taskgroup.1.c*

```
S-1 extern void start_background_work(void);
S-2 extern void check_step(void);
S-3 extern void print_results(void);
S-4 struct tree_node
S-5 {
S-6     struct tree_node *left;
S-7     struct tree_node *right;
S-8 };
S-9 typedef struct tree_node* tree_type;
S-10 extern void init_tree(tree_type);
S-11 #define max_steps 100
S-12 void compute_something(tree_type tree)
S-13 {
S-14     // some computation
S-15 }
S-16 void compute_tree(tree_type tree)
S-17 {
S-18     if (tree->left)
S-19     {
S-20         #pragma omp task
S-21         compute_tree(tree->left);
S-22     }
S-23     if (tree->right)
S-24     {
S-25         #pragma omp task
S-26         compute_tree(tree->right);
S-27     }
S-28     #pragma omp task
S-29     compute_something(tree);
S-30 }
S-31 int main()
S-32 {
```

```

S-33     int i;
S-34     tree_type tree;
S-35     init_tree(tree);
S-36     #pragma omp parallel
S-37     #pragma omp single
S-38     {
S-39         #pragma omp task
S-40         start_background_work();
S-41         for (i = 0; i < max_steps; i++)
S-42         {
S-43             #pragma omp taskgroup
S-44             {
S-45                 #pragma omp task
S-46                 compute_tree(tree);
S-47             } // wait on tree traversal in this step
S-48             check_step();
S-49         }
S-50     } // only now is background work required to be complete
S-51     print_results();
S-52     return 0;
S-53 }

```

▲ C / C++ ▲

▼ Fortran ▼

Example taskgroup.1.f90

```

S-1  module tree_type_mod
S-2      integer, parameter :: max_steps=100
S-3      type tree_type
S-4          type(tree_type), pointer :: left, right
S-5      end type
S-6      contains
S-7          subroutine compute_something(tree)
S-8              type(tree_type), pointer :: tree
S-9              ! some computation
S-10         end subroutine
S-11         recursive subroutine compute_tree(tree)
S-12             type(tree_type), pointer :: tree
S-13             if (associated(tree%left)) then
S-14             !$omp task
S-15                 call compute_tree(tree%left)
S-16             !$omp end task
S-17             endif
S-18             if (associated(tree%right)) then
S-19             !$omp task
S-20                 call compute_tree(tree%right)
S-21             !$omp end task

```

```

S-22         endif
S-23     !$omp task
S-24         call compute_something(tree)
S-25     !$omp end task
S-26     end subroutine
S-27 end module
S-28 program main
S-29     use tree_type_mod
S-30     type(tree_type), pointer :: tree
S-31     call init_tree(tree);
S-32     !$omp parallel
S-33     !$omp single
S-34     !$omp task
S-35         call start_background_work()
S-36     !$omp end task
S-37         do i=1, max_steps
S-38     !$omp taskgroup
S-39     !$omp task
S-40         call compute_tree(tree)
S-41     !$omp end task
S-42     !$omp end taskgroup ! wait on tree traversal in this step
S-43         call check_step()
S-44     enddo
S-45     !$omp end single
S-46     !$omp end parallel    ! only now is background work required to be complete
S-47         call print_results()
S-48 end program

```


Fortran

1 3.5 The taskyield Construct

2 The following example illustrates the use of the **taskyield** directive. The tasks in the example
3 compute something useful and then do some computation that must be done in a critical region. By
4 using **taskyield** when a task cannot get access to the **critical** region the implementation
5 can suspend the current task and schedule some other task that can do something useful.

▼ C / C++ ▼

6 *Example taskyield.1.c*

```
S-1 #include <omp.h>
S-2
S-3 void something_useful ( void );
S-4 void something_critical ( void );
S-5 void foo ( omp_lock_t * lock, int n )
S-6 {
S-7     int i;
S-8
S-9     for ( i = 0; i < n; i++ )
S-10         #pragma omp task
S-11         {
S-12             something_useful();
S-13             while ( !omp_test_lock(lock) ) {
S-14                 #pragma omp taskyield
S-15             }
S-16             something_critical();
S-17             omp_unset_lock(lock);
S-18         }
S-19 }
```

▲ C / C++ ▲

▼ Fortran ▼

7 *Example taskyield.1.f90*

```
S-1 subroutine foo ( lock, n )
S-2     use omp_lib
S-3     integer (kind=omp_lock_kind) :: lock
S-4     integer n
S-5     integer i
S-6
S-7     do i = 1, n
S-8         !$omp task
S-9         call something_useful()
S-10        do while ( .not. omp_test_lock(lock) )
S-11            !$omp taskyield
S-12        end do
```

```
S-13      call something_critical()
S-14      call omp_unset_lock(lock)
S-15      !$omp end task
S-16  end do
S-17
S-18  end subroutine
```

Fortran

1 3.6 The taskloop Construct

2 The following example illustrates how to execute a long running task concurrently with tasks
3 created with a **taskloop** directive for a loop having unbalanced amounts of work for its iterations.

4 The **grainsize** clause specifies that each task is to execute at least 500 iterations of the loop.

5 The **nogroup** clause removes the implicit taskgroup of the **taskloop** construct; the explicit
6 **taskgroup** construct in the example ensures that the function is not exited before the
7 long-running task and the loops have finished execution.

▼ C / C++ ▼

8 *Example taskloop.1.c*

```
S-1 void long_running_task(void);  
S-2 void loop_body(int i, int j);  
S-3  
S-4 void parallel_work(void) {  
S-5     int i, j;  
S-6     #pragma omp taskgroup  
S-7     {  
S-8         #pragma omp task  
S-9             long_running_task(); // can execute concurrently  
S-10  
S-11         #pragma omp taskloop private(j) grainsize(500) nogroup  
S-12             for (i = 0; i < 10000; i++) { // can execute concurrently  
S-13                 for (j = 0; j < i; j++) {  
S-14                     loop_body(i, j);  
S-15                 }  
S-16             }  
S-17     }  
S-18 }
```

▲ C / C++ ▲

1

Example taskloop.1.f90

```
S-1  subroutine parallel_work
S-2      integer i
S-3      integer j
S-4      !$omp taskgroup
S-5
S-6      !$omp task
S-7          call long_running_task()
S-8      !$omp end task
S-9
S-10     !$omp taskloop private(j) grainsize(500) nogroup
S-11         do i=1,10000
S-12             do j=1,i
S-13                 call loop_body(i, j)
S-14             end do
S-15         end do
S-16     !$omp end taskloop
S-17
S-18     !$omp end taskgroup
S-19 end subroutine
```

CHAPTER 4

Devices

The **target** construct consists of a **target** directive and an execution region. The **target** region is executed on the default device or the device specified in the **device** clause.

In OpenMP version 4.0, by default, all variables within the lexical scope of the construct are copied *to* and *from* the device, unless the device is the host, or the data exists on the device from a previously executed data-type construct that has created space on the device and possibly copied host data to the device storage.

The constructs that explicitly create storage, transfer data, and free storage on the device are categorized as structured and unstructured. The **target data** construct is structured. It creates a data region around **target** constructs, and is convenient for providing persistent data throughout multiple **target** regions. The **target enter data** and **target exit data** constructs are unstructured, because they can occur anywhere and do not support a "structure" (a region) for enclosing **target** constructs, as does the **target data** construct.

The **map** clause is used on **target** constructs and the data-type constructs to map host data. It specifies the device storage and data movement **to** and **from** the device, and controls on the storage duration.

There is an important change in the OpenMP 4.5 specification that alters the data model for scalar variables and C/C++ pointer variables. The default behavior for scalar variables and C/C++ pointer variables in an 4.5 compliant code is **firstprivate**. Example codes that have been updated to reflect this new behavior are annotated with a description that describes changes required for correct execution. Often it is a simple matter of mapping the variable as **tofrom** to obtain the intended 4.0 behavior.

In OpenMP version 4.5 the mechanism for target execution is specified as occurring through a *target task*. When the **target** construct is encountered a new *target task* is generated. The *target task* completes after the **target** region has executed and all data transfers have finished.

This new specification does not affect the execution of pre-4.5 code; it is a necessary element for asynchronous execution of the **target** region when using the new **nowait** clause introduced in OpenMP 4.5.

1 4.1 target Construct

2 4.1.1 target Construct on parallel Construct

3 This following example shows how the **target** construct offloads a code region to a target device.
4 The variables p , $v1$, $v2$, and N are implicitly mapped to the target device.

▼ C / C++ ▼

5 *Example target.1.c*

```
S-1 extern void init(float*, float*, int);  
S-2 extern void output(float*, int);  
S-3 void vec_mult(int N)  
S-4 {  
S-5     int i;  
S-6     float p[N], v1[N], v2[N];  
S-7     init(v1, v2, N);  
S-8     #pragma omp target  
S-9     #pragma omp parallel for private(i)  
S-10    for (i=0; i<N; i++)  
S-11        p[i] = v1[i] * v2[i];  
S-12    output(p, N);  
S-13 }
```

▲ C / C++ ▲
▼ Fortran ▼

6 *Example target.1.f90*

```
S-1 subroutine vec_mult(N)  
S-2     integer :: i,N  
S-3     real    :: p(N), v1(N), v2(N)  
S-4     call init(v1, v2, N)  
S-5     !$omp target  
S-6     !$omp parallel do  
S-7     do i=1,N  
S-8         p(i) = v1(i) * v2(i)  
S-9     end do  
S-10    !$omp end target  
S-11    call output(p, N)  
S-12 end subroutine
```

▲ Fortran ▲

1 4.1.2 target Construct with map Clause

2 This following example shows how the **target** construct offloads a code region to a target device.
3 The variables *p*, *v1* and *v2* are explicitly mapped to the target device using the **map** clause. The
4 variable *N* is implicitly mapped to the target device.

▼ C / C++ ▼

5 *Example target.2.c*

```
S-1 extern void init(float*, float*, int);  
S-2 extern void output(float*, int);  
S-3 void vec_mult(int N)  
S-4 {  
S-5     int i;  
S-6     float p[N], v1[N], v2[N];  
S-7     init(v1, v2, N);  
S-8     #pragma omp target map(v1, v2, p)  
S-9     #pragma omp parallel for  
S-10    for (i=0; i<N; i++)  
S-11        p[i] = v1[i] * v2[i];  
S-12    output(p, N);  
S-13 }
```

▲ C / C++ ▲

▼ Fortran ▼

6 *Example target.2.f90*

```
S-1 subroutine vec_mult(N)  
S-2     integer :: i,N  
S-3     real    :: p(N), v1(N), v2(N)  
S-4     call init(v1, v2, N)  
S-5     !$omp target map(v1,v2,p)  
S-6     !$omp parallel do  
S-7     do i=1,N  
S-8         p(i) = v1(i) * v2(i)  
S-9     end do  
S-10    !$omp end target  
S-11    call output(p, N)  
S-12 end subroutine
```

▲ Fortran ▲

1 4.1.3 map Clause with **to/from** map-types

2 The following example shows how the **target** construct offloads a code region to a target device.
3 In the **map** clause, the **to** and **from** map-types define the mapping between the original (host) data
4 and the target (device) data. The **to** map-type specifies that the data will only be read on the
5 device, and the **from** map-type specifies that the data will only be written to on the device. By
6 specifying a guaranteed access on the device, data transfers can be reduced for the **target** region.

7 The **to** map-type indicates that at the start of the **target** region the variables *v1* and *v2* are
8 initialized with the values of the corresponding variables on the host device, and at the end of the
9 **target** region the variables *v1* and *v2* are not assigned to their corresponding variables on the
10 host device.

11 The **from** map-type indicates that at the start of the **target** region the variable *p* is not initialized
12 with the value of the corresponding variable on the host device, and at the end of the **target**
13 region the variable *p* is assigned to the corresponding variable on the host device.

▼ C / C++ ▼

14 *Example target.3.c*

```
S-1 extern void init(float*, float*, int);  
S-2 extern void output(float*, int);  
S-3 void vec_mult(int N)  
S-4 {  
S-5     int i;  
S-6     float p[N], v1[N], v2[N];  
S-7     init(v1, v2, N);  
S-8     #pragma omp target map(to: v1, v2) map(from: p)  
S-9     #pragma omp parallel for  
S-10    for (i=0; i<N; i++)  
S-11        p[i] = v1[i] * v2[i];  
S-12    output(p, N);  
S-13 }
```

▲ C / C++ ▲

15 The **to** and **from** map-types allow programmers to optimize data motion. Since data for the *v*
16 arrays are not returned, and data for the *p* array are not transferred to the device, only one-half of
17 the data is moved, compared to the default behavior of an implicit mapping.

Example target.3.f90

```

S-1  subroutine vec_mult(N)
S-2      integer :: i,N
S-3      real    :: p(N), v1(N), v2(N)
S-4      call init(v1, v2, N)
S-5      !$omp target map(to: v1,v2) map(from: p)
S-6      !$omp parallel do
S-7          do i=1,N
S-8              p(i) = v1(i) * v2(i)
S-9          end do
S-10     !$omp end target
S-11     call output(p, N)
S-12 end subroutine

```

4.1.4 map Clause with Array Sections

The following example shows how the **target** construct offloads a code region to a target device. In the **map** clause, map-types are used to optimize the mapping of variables to the target device. Because variables *p*, *v1* and *v2* are pointers, array section notation must be used to map the arrays. The notation **:N** is equivalent to **0:N**.

Example target.4.c

```

S-1  extern void init(float*, float*, int);
S-2  extern void output(float*, int);
S-3  void vec_mult(float *p, float *v1, float *v2, int N)
S-4  {
S-5      int i;
S-6      init(v1, v2, N);
S-7      #pragma omp target map(to: v1[0:N], v2[:N]) map(from: p[0:N])
S-8      #pragma omp parallel for
S-9      for (i=0; i<N; i++)
S-10         p[i] = v1[i] * v2[i];
S-11         output(p, N);
S-12 }

```

C / C++

In C, the length of the pointed-to array must be specified. In Fortran the extent of the array is known and the length need not be specified. A section of the array can be specified with the usual Fortran syntax, as shown in the following example. The value 1 is assumed for the lower bound for array section `v2(:N)`.

Fortran

Example target.4.f90

```
S-1 module mults
S-2 contains
S-3 subroutine vec_mult(p,v1,v2,N)
S-4   real,pointer,dimension(:) :: p, v1, v2
S-5   integer :: N,i
S-6   call init(v1, v2, N)
S-7   !$omp target map(to: v1(1:N), v2(:N)) map(from: p(1:N))
S-8   !$omp parallel do
S-9   do i=1,N
S-10    p(i) = v1(i) * v2(i)
S-11  end do
S-12  !$omp end target
S-13  call output(p, N)
S-14 end subroutine
S-15 end module
```

Fortran

A more realistic situation in which an assumed-size array is passed to `vec_mult` requires that the length of the arrays be specified, because the compiler does not know the size of the storage. A section of the array must be specified with the usual Fortran syntax, as shown in the following example. The value 1 is assumed for the lower bound for array section `v2(:N)`.

Example *target.4b.f90*

```

S-1  module mults
S-2  contains
S-3  subroutine vec_mult(p,v1,v2,N)
S-4      real,dimension(*) :: p, v1, v2
S-5      integer          :: N,i
S-6      call init(v1, v2, N)
S-7      !$omp target map(to: v1(1:N), v2(:N)) map(from: p(1:N))
S-8      !$omp parallel do
S-9      do i=1,N
S-10         p(i) = v1(i) * v2(i)
S-11      end do
S-12      !$omp end target
S-13      call output(p, N)
S-14  end subroutine
S-15  end module

```

2 4.1.5 target Construct with if Clause

The following example shows how the **target** construct offloads a code region to a target device.

The **if** clause on the **target** construct indicates that if the variable *N* is smaller than a given threshold, then the **target** region will be executed by the host device.

The **if** clause on the **parallel** construct indicates that if the variable *N* is smaller than a second threshold then the **parallel** region is inactive.

Example *target.5.c*

```

S-1  #define THRESHOLD1 1000000
S-2  #define THRESHOLD2 1000
S-3  extern void init(float*, float*, int);
S-4  extern void output(float*, int);
S-5  void vec_mult(float *p, float *v1, float *v2, int N)
S-6  {
S-7      int i;
S-8      init(v1, v2, N);
S-9      #pragma omp target if(N>THRESHOLD1) map(to: v1[0:N], v2[:N])\
S-10         map(from: p[0:N])
S-11      #pragma omp parallel for if(N>THRESHOLD2)

```

```

S-12     for (i=0; i<N; i++)
S-13         p[i] = v1[i] * v2[i];
S-14     output(p, N);
S-15 }

```

C / C++

Fortran

1

Example target.5.f90

```

S-1  module params
S-2  integer,parameter :: THRESHOLD1=1000000, THRESHHOLD2=1000
S-3  end module
S-4  subroutine vec_mult(p, v1, v2, N)
S-5      use params
S-6      real      :: p(N), v1(N), v2(N)
S-7      integer :: i
S-8      call init(v1, v2, N)
S-9      !$omp target if(N>THRESHHOLD1) map(to: v1, v2 ) map(from: p)
S-10     !$omp parallel do if(N>THRESHOLD2)
S-11         do i=1,N
S-12             p(i) = v1(i) * v2(i)
S-13         end do
S-14     !$omp end target
S-15     call output(p, N)
S-16 end subroutine

```

Fortran

2

3

4

The following example is a modification of the above *target.5* code to show the combined **target** and parallel loop directives. It uses the *directive-name* modifier in multiple **if** clauses to specify the component directive to which it applies.

5

6

7

The **if** clause with the **target** modifier applies to the **target** component of the combined directive, and the **if** clause with the **parallel** modifier applies to the **parallel** component of the combined directive.

1 *Example target.6.c*

```

S-1  #define THRESHOLD1 1000000
S-2  #define THRESHOLD2 1000
S-3  extern void init(float*, float*, int);
S-4  extern void output(float*, int);
S-5  void vec_mult(float *p, float *v1, float *v2, int N)
S-6  {
S-7      int i;
S-8      init(v1, v2, N);
S-9      #pragma omp target parallel for \
S-10         if(target: N>THRESHOLD1) if(parallel: N>THRESHOLD2) \
S-11         map(to: v1[0:N], v2[:N]) map(from: p[0:N])
S-12         for (i=0; i<N; i++)
S-13             p[i] = v1[i] * v2[i];
S-14         output(p, N);
S-15 }

```

2 *Example target.6.f90*

```

S-1  module params
S-2  integer,parameter :: THRESHOLD1=1000000, THRESHHOLD2=1000
S-3  end module
S-4  subroutine vec_mult(p, v1, v2, N)
S-5      use params
S-6      real :: p(N), v1(N), v2(N)
S-7      integer :: i
S-8      call init(v1, v2, N)
S-9      !$omp target parallel do &
S-10         !$omp& if(target: N>THRESHHOLD1) if(parallel: N>THRESHOLD2) &
S-11         !$omp& map(to: v1, v2 ) map(from: p)
S-12         do i=1,N
S-13             p(i) = v1(i) * v2(i)
S-14         end do
S-15         !$omp end target parallel do
S-16         call output(p, N)
S-17 end subroutine

```


1 4.2 target data Construct

2 4.2.1 Simple target data Construct

3 This example shows how the **target data** construct maps variables to a device data
4 environment. The **target data** construct creates a new device data environment and maps the
5 variables $v1$, $v2$, and p to the new device data environment. The **target** construct enclosed in the
6 **target data** region creates a new device data environment, which inherits the variables $v1$, $v2$,
7 and p from the enclosing device data environment. The variable N is mapped into the new device
8 data environment from the encountering task's data environment.

▼ C / C++ ▼

9 *Example target_data.1.c*

```
S-1 extern void init(float*, float*, int);  
S-2 extern void output(float*, int);  
S-3 void vec_mult(float *p, float *v1, float *v2, int N)  
S-4 {  
S-5     int i;  
S-6     init(v1, v2, N);  
S-7     #pragma omp target data map(to: v1[0:N], v2[:N]) map(from: p[0:N])  
S-8     {  
S-9         #pragma omp target  
S-10        #pragma omp parallel for  
S-11        for (i=0; i<N; i++)  
S-12            p[i] = v1[i] * v2[i];  
S-13    }  
S-14    output(p, N);  
S-15 }
```

▲ C / C++ ▲

10 The Fortran code passes a reference and specifies the extent of the arrays in the declaration. No
11 length information is necessary in the map clause, as is required with C/C++ pointers.

Example *target_data.1.f90*

```

S-1  subroutine vec_mult(p, v1, v2, N)
S-2      real    :: p(N), v1(N), v2(N)
S-3      integer :: i
S-4      call init(v1, v2, N)
S-5      !$omp target data map(to: v1, v2) map(from: p)
S-6      !$omp target
S-7      !$omp parallel do
S-8          do i=1,N
S-9              p(i) = v1(i) * v2(i)
S-10         end do
S-11     !$omp end target
S-12     !$omp end target data
S-13     call output(p, N)
S-14 end subroutine

```

4.2.2 target data Region Enclosing Multiple target Regions

The following examples show how the **target data** construct maps variables to a device data environment of a **target** region. The **target data** construct creates a device data environment and encloses **target** regions, which have their own device data environments. The device data environment of the **target data** region is inherited by the device data environment of an enclosed **target** region. The **target data** construct is used to create variables that will persist throughout the **target data** region.

In the following example the variables *v1* and *v2* are mapped at each **target** construct. Instead of mapping the variable *p* twice, once at each **target** construct, *p* is mapped once by the **target data** construct.

C / C++

Example target_data.2.c

```

S-1  extern void init(float*, float*, int);
S-2  extern void init_again(float*, float*, int);
S-3  extern void output(float*, int);
S-4  void vec_mult(float *p, float *v1, float *v2, int N)
S-5  {
S-6      int i;
S-7      init(v1, v2, N);
S-8      #pragma omp target data map(from: p[0:N])
S-9      {
S-10         #pragma omp target map(to: v1[:N], v2[:N])
S-11         #pragma omp parallel for
S-12         for (i=0; i<N; i++)
S-13             p[i] = v1[i] * v2[i];
S-14         init_again(v1, v2, N);
S-15         #pragma omp target map(to: v1[:N], v2[:N])
S-16         #pragma omp parallel for
S-17         for (i=0; i<N; i++)
S-18             p[i] = p[i] + (v1[i] * v2[i]);
S-19     }
S-20     output(p, N);
S-21 }

```

C / C++

The Fortran code uses reference and specifies the extent of the *p*, *v1* and *v2* arrays. No length information is necessary in the **map** clause, as is required with C/C++ pointers. The arrays *v1* and *v2* are mapped at each **target** construct. Instead of mapping the array *p* twice, once at each target construct, *p* is mapped once by the **target data** construct.

Fortran

Example target_data.2.f90

```

S-1  subroutine vec_mult(p, v1, v2, N)
S-2      real    :: p(N), v1(N), v2(N)
S-3      integer :: i
S-4      call init(v1, v2, N)
S-5      !$omp target data map(from: p)
S-6          !$omp target map(to: v1, v2 )
S-7          !$omp parallel do
S-8              do i=1,N
S-9                  p(i) = v1(i) * v2(i)
S-10             end do
S-11         !$omp end target
S-12         call init_again(v1, v2, N)

```

```

S-13      !$omp target map(to: v1, v2 )
S-14      !$omp parallel do
S-15      do i=1,N
S-16      p(i) = p(i) + v1(i) * v2(i)
S-17      end do
S-18      !$omp end target
S-19      !$omp end target data
S-20      call output(p, N)
S-21      end subroutine

```

Fortran

In the following example, the variable `tmp` defaults to **tofrom** map-type and is mapped at each **target** construct. The array `Q` is mapped once at the enclosing **target data** region instead of at each **target** construct.

C / C++

Example target_data.3.c

```

S-1  #include <math.h>
S-2  #define COLS 100
S-3  void gramSchmidt(float Q[][COLS], const int rows)
S-4  {
S-5      int cols = COLS;
S-6      #pragma omp target data map(Q[0:rows][0:cols])
S-7      for(int k=0; k < cols; k++)
S-8      {
S-9          double tmp = 0.0;
S-10         #pragma omp target map(tofrom: tmp)
S-11         #pragma omp parallel for reduction(+:tmp)
S-12         for(int i=0; i < rows; i++)
S-13             tmp += (Q[i][k] * Q[i][k]);
S-14
S-15         tmp = 1/sqrt(tmp);
S-16
S-17         #pragma omp target
S-18         #pragma omp parallel for
S-19         for(int i=0; i < rows; i++)
S-20             Q[i][k] *= tmp;
S-21     }
S-22 }
S-23
S-24 /* Note: The variable tmp is now mapped with tofrom, for correct
S-25    execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
S-26    */

```

C / C++

In the following example the arrays *v1* and *v2* are mapped at each **target** construct. Instead of mapping the array *Q* twice at each **target** construct, *Q* is mapped once by the **target data** construct. Note, the *tmp* variable is implicitly remapped for each **target** region, mapping the value from the device to the host at the end of the first **target** region, and from the host to the device for the second **target** region.

Fortran

Example target_data.3.f90

```

S-1  subroutine gramSchmidt(Q,rows,cols)
S-2  integer                ::  rows,cols,  i,k
S-3  double precision      ::  Q(rows,cols), tmp
S-4      !$omp target data map(Q)
S-5      do k=1,cols
S-6          tmp = 0.0d0
S-7          !$omp target map(tofrom: tmp)
S-8              !$omp parallel do reduction(+:tmp)
S-9                  do i=1,rows
S-10                     tmp = tmp + (Q(i,k) * Q(i,k))
S-11                 end do
S-12             !$omp end target
S-13
S-14             tmp = 1.0d0/sqrt(tmp)
S-15
S-16             !$omp target
S-17                 !$omp parallel do
S-18                     do i=1,rows
S-19                         Q(i,k) = Q(i,k)*tmp
S-20                     enddo
S-21             !$omp end target
S-22         end do
S-23         !$omp end target data
S-24     end subroutine
S-25
S-26     ! Note: The variable tmp is now mapped with tofrom, for correct
S-27     ! execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.

```

Fortran

1 4.2.3 target data Construct with Orphaned Call

2 The following two examples show how the **target data** construct maps variables to a device
3 data environment. The **target data** construct's device data environment encloses the **target**
4 construct's device data environment in the function **vec_mult()**.

5 When the type of the variable appearing in an array section is pointer, the pointer variable and the
6 storage location of the corresponding array section are mapped to the device data environment. The
7 pointer variable is treated as if it had appeared in a **map** clause with a map-type of **alloc**. The
8 array section's storage location is mapped according to the map-type in the **map** clause (the default
9 map-type is **tofrom**).

10 The **target** construct's device data environment inherits the storage locations of the array
11 sections $v1[0:N]$, $v2[:n]$, and $p0[0:N]$ from the enclosing target data construct's device data
12 environment. Neither initialization nor assignment is performed for the array sections in the new
13 device data environment.

14 The pointer variables $p1$, $v3$, and $v4$ are mapped into the target construct's device data environment
15 with an implicit map-type of **alloc** and they are assigned the address of the storage location
16 associated with their corresponding array sections. Note that the following pairs of array section
17 storage locations are equivalent ($p0[:N]$, $p1[:N]$), ($v1[:N]$, $v3[:N]$), and ($v2[:N]$, $v4[:N]$).

▼ C / C++ ▼

18 *Example target_data.4.c*

```
S-1 void vec_mult(float*, float*, float*, int);
S-2 extern void init(float*, float*, int);
S-3 extern void output(float*, int);
S-4 void foo(float *p0, float *v1, float *v2, int N)
S-5 {
S-6     init(v1, v2, N);
S-7     #pragma omp target data map(to: v1[0:N], v2[:N]) map(from: p0[0:N])
S-8     {
S-9         vec_mult(p0, v1, v2, N);
S-10    }
S-11    output(p0, N);
S-12 }
S-13 void vec_mult(float *p1, float *v3, float *v4, int N)
S-14 {
S-15     int i;
S-16     #pragma omp target map(to: v3[0:N], v4[:N]) map(from: p1[0:N])
S-17     #pragma omp parallel for
S-18     for (i=0; i<N; i++)
S-19         p1[i] = v3[i] * v4[i];
S-20 }
```

The Fortran code maps the pointers and storage in an identical manner (same extent, but uses indices from 1 to N).

The **target** construct's device data environment inherits the storage locations of the arrays $v1$, $v2$ and $p0$ from the enclosing **target data** constructs's device data environment. However, in Fortran the associated data of the pointer is known, and the shape is not required.

The pointer variables $p1$, $v3$, and $v4$ are mapped into the **target** construct's device data environment with an implicit map-type of **alloc** and they are assigned the address of the storage location associated with their corresponding array sections. Note that the following pair of array storage locations are equivalent ($p0,p1$), ($v1,v3$), and ($v2,v4$).

Example target_data.4.f90

```

S-1  module mults
S-2  contains
S-3  subroutine foo(p0,v1,v2,N)
S-4  real,pointer,dimension(:) :: p0, v1, v2
S-5  integer :: N,i
S-6      call init(v1, v2, N)
S-7      !$omp target data map(to: v1, v2) map(from: p0)
S-8      call vec_mult(p0,v1,v2,N)
S-9      !$omp end target data
S-10     call output(p0, N)
S-11 end subroutine
S-12 subroutine vec_mult(p1,v3,v4,N)
S-13 real,pointer,dimension(:) :: p1, v3, v4
S-14 integer :: N,i
S-15     !$omp target map(to: v3, v4) map(from: p1)
S-16     !$omp parallel do
S-17     do i=1,N
S-18         p1(i) = v3(i) * v4(i)
S-19     end do
S-20     !$omp end target
S-21 end subroutine
S-22 end module

```

In the following example, the variables $p1$, $v3$, and $v4$ are references to the pointer variables $p0$, $v1$ and $v2$ respectively. The **target** construct's device data environment inherits the pointer variables $p0$, $v1$, and $v2$ from the enclosing **target data** construct's device data environment. Thus, $p1$, $v3$, and $v4$ are already present in the device data environment.

C++

Example *target_data.5.cpp*

```

S-1 void vec_mult(float* &, float* &, float* &, int &);
S-2 extern void init(float*, float*, int);
S-3 extern void output(float*, int);
S-4 void foo(float *p0, float *v1, float *v2, int N)
S-5 {
S-6     init(v1, v2, N);
S-7     #pragma omp target data map(to: v1[0:N], v2[:N]) map(from: p0[0:N])
S-8     {
S-9         vec_mult(p0, v1, v2, N);
S-10    }
S-11    output(p0, N);
S-12 }
S-13 void vec_mult(float* &p1, float* &v3, float* &v4, int &N)
S-14 {
S-15     int i;
S-16     #pragma omp target map(to: v3[0:N], v4[:N]) map(from: p1[0:N])
S-17     #pragma omp parallel for
S-18     for (i=0; i<N; i++)
S-19         p1[i] = v3[i] * v4[i];
S-20 }

```

C++

In the following example, the usual Fortran approach is used for dynamic memory. The *p0*, *v1*, and *v2* arrays are allocated in the main program and passed as references from one routine to another. In **vec_mult**, *p1*, *v3* and *v4* are references to the *p0*, *v1*, and *v2* arrays, respectively. The **target** construct's device data environment inherits the arrays *p0*, *v1*, and *v2* from the enclosing target data construct's device data environment. Thus, *p1*, *v3*, and *v4* are already present in the device data environment.

Fortran

Example *target_data.5.f90*

```

S-1 module my_mult
S-2 contains
S-3 subroutine foo(p0,v1,v2,N)
S-4 real,dimension(:) :: p0, v1, v2
S-5 integer           :: N,i
S-6     call init(v1, v2, N)
S-7     !$omp target data map(to: v1, v2) map(from: p0)
S-8     call vec_mult(p0,v1,v2,N)
S-9     !$omp end target data
S-10    call output(p0, N)
S-11 end subroutine

```



```

S-12  subroutine vec_mult (p1,v3,v4,N)
S-13  real,dimension(:) :: p1, v3, v4
S-14  integer           :: N,i
S-15      !$omp target map(to: v3, v4) map(from: p1)
S-16      !$omp parallel do
S-17          do i=1,N
S-18              p1(i) = v3(i) * v4(i)
S-19          end do
S-20      !$omp end target
S-21  end subroutine
S-22  end module
S-23  program main
S-24  use my_mult
S-25  integer, parameter :: N=1024
S-26  real,allocatable, dimension(:) :: p, v1, v2
S-27      allocate( p(N), v1(N), v2(N) )
S-28      call foo(p,v1,v2,N)
S-29  end program

```

Fortran

1 4.2.4 target data Construct with if Clause

2 The following two examples show how the **target data** construct maps variables to a device
3 data environment.

4 In the following example, the if clause on the **target data** construct indicates that if the variable
5 *N* is smaller than a given threshold, then the **target data** construct will not create a device data
6 environment.

7 The **target** constructs enclosed in the **target data** region must also use an **if** clause on the
8 same condition, otherwise the pointer variable *p* is implicitly mapped with a map-type of **tofrom**,
9 but the storage location for the array section *p[0:N]* will not be mapped in the device data
10 environments of the **target** constructs.

Example *target_data.6.c*

```

S-1  #define THRESHOLD 1000000
S-2  extern void init(float*, float*, int);
S-3  extern void init_again(float*, float*, int);
S-4  extern void output(float*, int);
S-5  void vec_mult(float *p, float *v1, float *v2, int N)
S-6  {
S-7      int i;
S-8      init(v1, v2, N);
S-9      #pragma omp target data if (N>THRESHOLD) map(from: p[0:N])
S-10     {
S-11         #pragma omp target if (N>THRESHOLD) map(to: v1[:N], v2[:N])
S-12         #pragma omp parallel for
S-13         for (i=0; i<N; i++)
S-14             p[i] = v1[i] * v2[i];
S-15         init_again(v1, v2, N);
S-16         #pragma omp target if (N>THRESHOLD) map(to: v1[:N], v2[:N])
S-17         #pragma omp parallel for
S-18         for (i=0; i<N; i++)
S-19             p[i] = p[i] + (v1[i] * v2[i]);
S-20     }
S-21     output(p, N);
S-22 }

```

The **if** clauses work the same way for the following Fortran code. The **target** constructs enclosed in the **target data** region should also use an **if** clause with the same condition, so that the **target data** region and the **target** region are either both created for the device, or are both ignored.

Example *target_data.6.f90*

```

S-1  module params
S-2  integer,parameter :: THRESHOLD=1000000
S-3  end module
S-4  subroutine vec_mult(p, v1, v2, N)
S-5      use params
S-6      real :: p(N), v1(N), v2(N)
S-7      integer :: i
S-8      call init(v1, v2, N)
S-9      !$omp target data if (N>THRESHOLD) map(from: p)
S-10         !$omp target if (N>THRESHOLD) map(to: v1, v2)
S-11         !$omp parallel do

```

```

S-12         do i=1,N
S-13             p(i) = v1(i) * v2(i)
S-14         end do
S-15         !$omp end target
S-16         call init_again(v1, v2, N)
S-17         !$omp target if(N>THRESHOLD) map(to: v1, v2)
S-18             !$omp parallel do
S-19                 do i=1,N
S-20                     p(i) = p(i) + v1(i) * v2(i)
S-21                 end do
S-22             !$omp end target
S-23         !$omp end target data
S-24         call output(p, N)
S-25     end subroutine

```

Fortran

1 In the following example, when the **if** clause conditional expression on the **target** construct
2 evaluates to *false*, the target region will execute on the host device. However, the **target data**
3 construct created an enclosing device data environment that mapped $p[0:N]$ to a device data
4 environment on the default device. At the end of the **target data** region the array section
5 $p[0:N]$ will be assigned from the device data environment to the corresponding variable in the data
6 environment of the task that encountered the **target data** construct, resulting in undefined
7 values in $p[0:N]$.

C / C++

8 *Example target_data.7.c*

```

S-1     #define THRESHOLD 1000000
S-2     extern void init(float*, float*, int);
S-3     extern void output(float*, int);
S-4     void vec_mult(float *p, float *v1, float *v2, int N)
S-5     {
S-6         int i;
S-7         init(v1, v2, N);
S-8         #pragma omp target data map(from: p[0:N])
S-9         {
S-10            #pragma omp target if (N>THRESHOLD) map(to: v1[:N], v2[:N])
S-11            #pragma omp parallel for
S-12            for (i=0; i<N; i++)
S-13                p[i] = v1[i] * v2[i];
S-14        } /* UNDEFINED behavior if N<=THRESHOLD */
S-15        output(p, N);
S-16    }

```

The **if** clauses work the same way for the following Fortran code. When the **if** clause conditional expression on the **target** construct evaluates to *false*, the **target** region will execute on the host device. However, the **target data** construct created an enclosing device data environment that mapped the *p* array (and *v1* and *v2*) to a device data environment on the default target device. At the end of the **target data** region the *p* array will be assigned from the device data environment to the corresponding variable in the data environment of the task that encountered the **target data** construct, resulting in undefined values in *p*.

Example target_data.f90

```

S-1  module params
S-2  integer, parameter :: THRESHOLD=1000000
S-3  end module
S-4  subroutine vec_mult(p, v1, v2, N)
S-5      use params
S-6      real      :: p(N), v1(N), v2(N)
S-7      integer :: i
S-8      call init(v1, v2, N)
S-9      !$omp target data map(from: p)
S-10         !$omp target if(N>THRESHOLD) map(to: v1, v2)
S-11         !$omp parallel do
S-12             do i=1,N
S-13                 p(i) = v1(i) * v2(i)
S-14             end do
S-15         !$omp end target
S-16     !$omp end target data
S-17     call output(p, N)  !*** UNDEFINED behavior if N<=THRESHOLD
S-18 end subroutine

```

1 4.3 target enter data and target exit data 2 Constructs

3 The structured data construct (**target data**) provides persistent data on a device for subsequent
4 **target** constructs as shown in the **target data** examples above. This is accomplished by
5 creating a single **target data** region containing **target** constructs.

6 The unstructured data constructs allow the creation and deletion of data on the device at any
7 appropriate point within the host code, as shown below with the **target enter data** and
8 **target exit data** constructs.

9 The following C++ code creates/deletes a vector in a constructor/destructor of a class. The
10 constructor creates a vector with **target enter data** and uses an **alloc** modifier in the **map**
11 clause to avoid copying values to the device. The destructor deletes the data
12 (**target exit data**) and uses the **delete** modifier in the **map** clause to avoid copying data
13 back to the host. Note, the stand-alone **target enter data** occurs after the host vector is
14 created, and the **target exit data** construct occurs before the host data is deleted.

C++

15 *Example target_unstructured_data.1.cpp*

```
S-1 class Matrix
S-2 {
S-3
S-4     Matrix(int n) {
S-5         len = n;
S-6         v = new double[len];
S-7         #pragma omp target enter data map(alloc:v[0:len])
S-8     }
S-9
S-10    ~Matrix() {
S-11        // NOTE: delete map type should be used, since the corresponding
S-12        // host data will cease to exist after the deconstructor is called.
S-13
S-14        #pragma omp target exit data map(delete:v[0:len])
S-15        delete[] v;
S-16    }
S-17
S-18    private:
S-19        double* v;
S-20        int len;
S-21
S-22 };
```

C++

16 The following C code allocates and frees the data member of a Matrix structure. The

init_matrix function allocates the memory used in the structure and uses the **target enter data** directive to map it to the target device. The **free_matrix** function removes the mapped array from the target device and then frees the memory on the host. Note, the stand-alone **target enter data** occurs after the host memory is allocated, and the **target exit data** construct occurs before the host data is freed.

C / C++

Example target_unstructured_data.1.c

```
S-1  #include <stdlib.h>
S-2  typedef struct {
S-3      double *A;
S-4      int N;
S-5  } Matrix;
S-6
S-7  void init_matrix(Matrix *mat, int n)
S-8  {
S-9      mat->A = (double *)malloc(n*sizeof(double));
S-10     mat->N = n;
S-11     #pragma omp target enter data map(alloc:mat->A[:n])
S-12 }
S-13
S-14 void free_matrix(Matrix *mat)
S-15 {
S-16     #pragma omp target exit data map(delete:mat->A[:mat->N])
S-17     mat->N = 0;
S-18     free(mat->A);
S-19     mat->A = NULL;
S-20 }
```

C / C++

The following Fortran code allocates and deallocates a module array. The **initialize** subroutine allocates the module array and uses the **target enter data** directive to map it to the target device. The **finalize** subroutine removes the mapped array from the target device and then deallocates the array on the host. Note, the stand-alone **target enter data** occurs after the host memory is allocated, and the **target exit data** construct occurs before the host data is deallocated.

1 *Example target_unstructured_data.1.f90*

```
S-1 module example
S-2   real(8), allocatable :: A(:)
S-3
S-4   contains
S-5     subroutine initialize(N)
S-6       integer :: N
S-7
S-8       allocate(A(N))
S-9       !$omp target enter data map(alloc:A)
S-10
S-11     end subroutine initialize
S-12
S-13     subroutine finalize()
S-14
S-15       !$omp target exit data map(delete:A)
S-16       deallocate(A)
S-17
S-18     end subroutine finalize
S-19 end module example
```

1 4.4 target update Construct

2 4.4.1 Simple target data and target update Constructs

3 The following example shows how the **target update** construct updates variables in a device
4 data environment.

5 The **target data** construct maps array sections $v1[:N]$ and $v2[:N]$ (arrays $v1$ and $v2$ in the
6 Fortran code) into a device data environment.

7 The task executing on the host device encounters the first **target** region and waits for the
8 completion of the region.

9 After the execution of the first **target** region, the task executing on the host device then assigns
10 new values to $v1[:N]$ and $v2[:N]$ ($v1$ and $v2$ arrays in Fortran code) in the task's data environment
11 by calling the function `init_again()`.

12 The **target update** construct assigns the new values of $v1$ and $v2$ from the task's data
13 environment to the corresponding mapped array sections in the device data environment of the
14 **target data** construct.

15 The task executing on the host device then encounters the second **target** region and waits for the
16 completion of the region.

17 The second **target** region uses the updated values of $v1[:N]$ and $v2[:N]$.

▼ C / C++ ▼

18 *Example target_update.1.c*

```
S-1 extern void init(float *, float *, int);  
S-2 extern void init_again(float *, float *, int);  
S-3 extern void output(float *, int);  
S-4 void vec_mult(float *p, float *v1, float *v2, int N)  
S-5 {  
S-6     int i;  
S-7     init(v1, v2, N);  
S-8     #pragma omp target data map(to: v1[:N], v2[:N]) map(from: p[0:N])  
S-9     {  
S-10        #pragma omp target  
S-11        #pragma omp parallel for  
S-12        for (i=0; i<N; i++)  
S-13            p[i] = v1[i] * v2[i];  
S-14        init_again(v1, v2, N);  
S-15        #pragma omp target update to(v1[:N], v2[:N])  
S-16        #pragma omp target  
S-17        #pragma omp parallel for  
S-18        for (i=0; i<N; i++)
```



```

S-19         p[i] = p[i] + (v1[i] * v2[i]);
S-20     }
S-21     output(p, N);
S-22 }

```

C / C++

Fortran

1

Example target_update.1.f90

```

S-1  subroutine vec_mult(p, v1, v2, N)
S-2      real    :: p(N), v1(N), v2(N)
S-3      integer :: i
S-4      call init(v1, v2, N)
S-5      !$omp target data map(to: v1, v2) map(from: p)
S-6          !$omp target
S-7          !$omp parallel do
S-8              do i=1,N
S-9                  p(i) = v1(i) * v2(i)
S-10             end do
S-11         !$omp end target
S-12         call init_again(v1, v2, N)
S-13         !$omp target update to(v1, v2)
S-14         !$omp target
S-15         !$omp parallel do
S-16             do i=1,N
S-17                 p(i) = p(i) + v1(i) * v2(i)
S-18             end do
S-19         !$omp end target
S-20     !$omp end target data
S-21     call output(p, N)
S-22 end subroutine

```

Fortran

1 4.4.2 target update Construct with if Clause

2 The following example shows how the **target update** construct updates variables in a device
3 data environment.

4 The **target data** construct maps array sections $v1[:N]$ and $v2[:N]$ (arrays $v1$ and $v2$ in the
5 Fortran code) into a device data environment. In between the two **target** regions, the task
6 executing on the host device conditionally assigns new values to $v1$ and $v2$ in the task's data
7 environment. The function **maybe_init_again()** returns *true* if new data is written.

8 When the conditional expression (the return value of **maybe_init_again()**) in the **if** clause
9 is *true*, the **target update** construct assigns the new values of $v1$ and $v2$ from the task's data
10 environment to the corresponding mapped array sections in the **target data** construct's device
11 data environment.

▼ C / C++ ▼

12 *Example target_update.2.c*

```
S-1 extern void init(float *, float *, int);
S-2 extern int maybe_init_again(float *, int);
S-3 extern void output(float *, int);
S-4 void vec_mult(float *p, float *v1, float *v2, int N)
S-5 {
S-6     int i;
S-7     init(v1, v2, N);
S-8     #pragma omp target data map(to: v1[:N], v2[:N]) map(from: p[0:N])
S-9     {
S-10        int changed;
S-11        #pragma omp target
S-12        #pragma omp parallel for
S-13        for (i=0; i<N; i++)
S-14            p[i] = v1[i] * v2[i];
S-15        changed = maybe_init_again(v1, N);
S-16        #pragma omp target update if (changed) to(v1[:N])
S-17        changed = maybe_init_again(v2, N);
S-18        #pragma omp target update if (changed) to(v2[:N])
S-19        #pragma omp target
S-20        #pragma omp parallel for
S-21        for (i=0; i<N; i++)
S-22            p[i] = p[i] + (v1[i] * v2[i]);
S-23    }
S-24    output(p, N);
S-25 }
```

▲ C / C++ ▲

1

Example target_update.2.f90

```

S-1  subroutine vec_mult(p, v1, v2, N)
S-2      interface
S-3          logical function maybe_init_again (v1, N)
S-4              real :: v1(N)
S-5              integer :: N
S-6          end function
S-7      end interface
S-8      real    :: p(N), v1(N), v2(N)
S-9      integer :: i
S-10     logical :: changed
S-11     call init(v1, v2, N)
S-12     !$omp target data map(to: v1, v2) map(from: p)
S-13         !$omp target
S-14             !$omp parallel do
S-15                 do i=1, N
S-16                     p(i) = v1(i) * v2(i)
S-17                 end do
S-18             !$omp end target
S-19             changed = maybe_init_again(v1, N)
S-20             !$omp target update if(changed) to(v1(:N))
S-21             changed = maybe_init_again(v2, N)
S-22             !$omp target update if(changed) to(v2(:N))
S-23         !$omp target
S-24             !$omp parallel do
S-25                 do i=1, N
S-26                     p(i) = p(i) + v1(i) * v2(i)
S-27                 end do
S-28             !$omp end target
S-29         !$omp end target data
S-30         call output(p, N)
S-31     end subroutine

```

1 4.5 declare target Construct

2 4.5.1 declare target and end declare target 3 for a Function

4 The following example shows how the **declare target** directive is used to indicate that the
5 corresponding call inside a **target** region is to a **fib** function that can execute on the default
6 target device.

7 A version of the function is also available on the host device. When the **if** clause conditional
8 expression on the **target** construct evaluates to *false*, the **target** region (thus **fib**) will execute
9 on the host device.

10 For C/C++ codes the declaration of the function **fib** appears between the **declare target** and
11 **end declare target** directives.

▼ C / C++ ▼

12 *Example declare_target.f.c*

```
S-1 #pragma omp declare target
S-2 extern void fib(int N);
S-3 #pragma omp end declare target
S-4 #define THRESHOLD 1000000
S-5 void fib_wrapper(int n)
S-6 {
S-7     #pragma omp target if(n > THRESHOLD)
S-8     {
S-9         fib(n);
S-10    }
S-11 }
```

▲ C / C++ ▲

13 The Fortran **fib** subroutine contains a **declare target** declaration to indicate to the compiler
14 to create an device executable version of the procedure. The subroutine name has not been included
15 on the **declare target** directive and is, therefore, implicitly assumed.

16 The program uses the **module_fib** module, which presents an explicit interface to the compiler
17 with the **declare target** declarations for processing the **fib** call.

1 *Example declare_target.1.f90*

```

S-1 module module_fib
S-2 contains
S-3     subroutine fib(N)
S-4         integer :: N
S-5         !$omp declare target
S-6         !...
S-7     end subroutine
S-8 end module
S-9 module params
S-10 integer :: THRESHOLD=1000000
S-11 end module
S-12 program my_fib
S-13 use params
S-14 use module_fib
S-15     !$omp target if( N > THRESHOLD )
S-16         call fib(N)
S-17     !$omp end target
S-18 end program

```

2 The next Fortran example shows the use of an external subroutine. Without an explicit interface
 3 (through module use or an interface block) the **declare target** declarations within a external
 4 subroutine are unknown to the main program unit; therefore, a **declare target** must be
 5 provided within the program scope for the compiler to determine that a target binary should be
 6 available.

7 *Example declare_target.2.f90*

```

S-1 program my_fib
S-2 integer :: N = 8
S-3 !$omp declare target(fib)
S-4     !$omp target
S-5         call fib(N)
S-6     !$omp end target
S-7 end program
S-8 subroutine fib(N)
S-9 integer :: N
S-10 !$omp declare target
S-11     print*, "hello from fib"
S-12     !...
S-13 end subroutine

```

1 4.5.2 declare target Construct for Class Type

C++

2 The following example shows how the **declare target** and **end declare target** directives
3 are used to enclose the declaration of a variable *varY* with a class type **typeY**. The member
4 function **typeY::foo()** cannot be accessed on a target device because its declaration did not
5 appear between **declare target** and **end declare target** directives.

6 *Example declare_target.2.cpp*

```
S-1 struct typeX
S-2 {
S-3     int a;
S-4 };
S-5 class typeY
S-6 {
S-7     int a;
S-8     public:
S-9         int foo() { return a^0x01;}
S-10 };
S-11 #pragma omp declare target
S-12 struct typeX varX; // ok
S-13 class typeY varY; // ok if varY.foo() not called on target device
S-14 #pragma omp end declare target
S-15 void foo()
S-16 {
S-17     #pragma omp target
S-18     {
S-19         varX.a = 100; // ok
S-20         varY.foo(); // error foo() is not available on a target device
S-21     }
S-22 }
```

C++

7 4.5.3 declare target and end declare target 8 for Variables

9 The following examples show how the **declare target** and **end declare target** directives
10 are used to indicate that global variables are mapped to the implicit device data environment of
11 each target device.

12 In the following example, the declarations of the variables *p*, *v1*, and *v2* appear between **declare**
13 **target** and **end declare target** directives indicating that the variables are mapped to the

implicit device data environment of each target device. The **target update** directive is then used to manage the consistency of the variables *p*, *v1*, and *v2* between the data environment of the encountering host device task and the implicit device data environment of the default target device.

C / C++

Example declare_target.3.c

```
S-1  #define N 1000
S-2  #pragma omp declare target
S-3  float p[N], v1[N], v2[N];
S-4  #pragma omp end declare target
S-5  extern void init(float *, float *, int);
S-6  extern void output(float *, int);
S-7  void vec_mult()
S-8  {
S-9      int i;
S-10     init(v1, v2, N);
S-11     #pragma omp target update to(v1, v2)
S-12     #pragma omp target
S-13     #pragma omp parallel for
S-14     for (i=0; i<N; i++)
S-15         p[i] = v1[i] * v2[i];
S-16     #pragma omp target update from(p)
S-17     output(p, N);
S-18 }
```

C / C++

The Fortran version of the above C code uses a different syntax. Fortran modules use a list syntax on the **declare target** directive to declare mapped variables.

Fortran

Example declare_target.3.f90

```
S-1  module my_arrays
S-2  !$omp declare target (N, p, v1, v2)
S-3  integer, parameter :: N=1000
S-4  real                :: p(N), v1(N), v2(N)
S-5  end module
S-6  subroutine vec_mult()
S-7  use my_arrays
S-8      integer :: i
S-9      call init(v1, v2, N);
S-10     !$omp target update to(v1, v2)
S-11     !$omp target
S-12     !$omp parallel do
S-13     do i = 1,N
S-14         p(i) = v1(i) * v2(i)
```

```

S-15     end do
S-16     !$omp end target
S-17     !$omp target update from (p)
S-18     call output(p, N)
S-19 end subroutine

```

Fortran

The following example also indicates that the function **Pfun()** is available on the target device, as well as the variable Q , which is mapped to the implicit device data environment of each target device. The **target update** directive is then used to manage the consistency of the variable Q between the data environment of the encountering host device task and the implicit device data environment of the default target device.

In the following example, the function and variable declarations appear between the **declare target** and **end declare target** directives.

C / C++

Example declare_target.4.c

```

S-1  #define N 10000
S-2  #pragma omp declare target
S-3  float Q[N][N];
S-4  float Pfun(const int i, const int k)
S-5  { return Q[i][k] * Q[k][i]; }
S-6  #pragma omp end declare target
S-7  float accum(int k)
S-8  {
S-9      float tmp = 0.0;
S-10     #pragma omp target update to(Q)
S-11     #pragma omp target map(tofrom: tmp)
S-12     #pragma omp parallel for reduction(+:tmp)
S-13     for(int i=0; i < N; i++)
S-14         tmp += Pfun(i,k);
S-15     return tmp;
S-16 }
S-17
S-18 /* Note: The variable tmp is now mapped with tofrom, for correct
S-19 execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
S-20 */

```

C / C++

The Fortran version of the above C code uses a different syntax. In Fortran modules a list syntax on the **declare target** directive is used to declare mapped variables and procedures. The N and Q variables are declared as a comma separated list. When the **declare target** directive is used to declare just the procedure, the procedure name need not be listed – it is implicitly assumed, as illustrated in the **Pfun()** function.

Example declare_target.4.f90

```

S-1  module my_global_array
S-2  !$omp declare target (N,Q)
S-3  integer, parameter :: N=10
S-4  real                :: Q(N,N)
S-5  contains
S-6  function Pfun(i,k)
S-7  !$omp declare target
S-8  real                :: Pfun
S-9  integer,intent(in) :: i,k
S-10     Pfun=(Q(i,k) * Q(k,i))
S-11  end function
S-12  end module

S-13
S-14  function accum(k) result(tmp)
S-15  use my_global_array
S-16  real    :: tmp
S-17  integer :: i, k
S-18     tmp = 0.0e0
S-19     !$omp target map(tofrom: tmp)
S-20     !$omp parallel do reduction(+:tmp)
S-21     do i=1,N
S-22         tmp = tmp + Pfun(k,i)
S-23     end do
S-24     !$omp end target
S-25  end function
S-26
S-27  ! Note: The variable tmp is now mapped with tofrom, for correct
S-28  ! execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.

```

2 4.5.4 declare target and end declare target 3 with declare simd

4 The following example shows how the **declare target** and **end declare target** directives
5 are used to indicate that a function is available on a target device. The **declare simd** directive
6 indicates that there is a SIMD version of the function **P ()** that is available on the target device as
7 well as one that is available on the host device.

Example declare_target.5.c

```

S-1  #define N 10000
S-2  #define M 1024
S-3  #pragma omp declare target
S-4  float Q[N][N];
S-5  #pragma omp declare simd uniform(i) linear(k) notinbranch
S-6  float P(const int i, const int k)
S-7  {
S-8      return Q[i][k] * Q[k][i];
S-9  }
S-10 #pragma omp end declare target
S-11
S-12 float accum(void)
S-13 {
S-14     float tmp = 0.0;
S-15     int i, k;
S-16     #pragma omp target map(tofrom: tmp)
S-17     #pragma omp parallel for reduction(+:tmp)
S-18     for (i=0; i < N; i++) {
S-19         float tmp1 = 0.0;
S-20         #pragma omp simd reduction(+:tmp1)
S-21         for (k=0; k < M; k++) {
S-22             tmp1 += P(i,k);
S-23         }
S-24         tmp += tmp1;
S-25     }
S-26     return tmp;
S-27 }
S-28
S-29 /* Note: The variable tmp is now mapped with tofrom, for correct
S-30     execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
S-31 */

```

The Fortran version of the above C code uses a different syntax. Fortran modules use a list syntax of the **declare target** declaration for the mapping. Here the *N* and *Q* variables are declared in the list form as a comma separated list. The function declaration does not use a list and implicitly assumes the function name. In this Fortran example row and column indices are reversed relative to the C/C++ example, as is usual for codes optimized for memory access.

1

Example declare_target.f90

```

S-1  module my_global_array
S-2  !$omp declare target (N,Q)
S-3  integer, parameter :: N=10000, M=1024
S-4  real                :: Q(N,N)
S-5  contains
S-6  function P(k,i)
S-7  !$omp declare simd uniform(i) linear(k) notinbranch
S-8  !$omp declare target
S-9  real                :: P
S-10 integer,intent(in) :: k,i
S-11    P=(Q(k,i) * Q(i,k))
S-12 end function
S-13 end module
S-14
S-15 function accum() result(tmp)
S-16 use my_global_array
S-17 real    :: tmp, tmp1
S-18 integer :: i
S-19    tmp = 0.0e0
S-20    !$omp target map(tofrom: tmp)
S-21    !$omp parallel do private(tmp1) reduction(+:tmp)
S-22    do i=1,N
S-23        tmp1 = 0.0e0
S-24        !$omp simd reduction(+:tmp1)
S-25        do k = 1,M
S-26            tmp1 = tmp1 + P(k,i)
S-27        end do
S-28        tmp = tmp + tmp1
S-29    end do
S-30    !$omp end target
S-31 end function
S-32
S-33 ! Note: The variable tmp is now mapped with tofrom, for correct
S-34 ! execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.

```

1 4.5.5 declare target Directive with link Clause

2 In the OpenMP 4.5 standard the **declare target** directive was extended to allow static data to
3 be mapped, *when needed*, through a **link** clause.

4 Data storage for items listed in the **link** clause becomes available on the device when it is mapped
5 implicitly or explicitly in a **map** clause, and it persists for the scope of the mapping (as specified by
6 a **target** construct, a **target data** construct, or **target enter/exit data** constructs).

7 Tip: When all the global data items will not fit on a device and are not needed simultaneously, use
8 the **link** clause and map the data only when it is needed.

9 The following C and Fortran examples show two sets of data (single precision and double
10 precision) that are global on the host for the entire execution on the host; but are only used globally
11 on the device for part of the program execution. The single precision data are allocated and persist
12 only for the first **target** region. Similarly, the double precision data are in scope on the device
13 only for the second **target** region.

▼ C / C++ ▼

14 *Example declare_target.6.c*

```
S-1 #define N 100000000
S-2
S-3 #pragma omp declare target link(sp,sv1,sv2) \
S-4                               link(dp,dv1,dv2)
S-5 float sp[N], sv1[N], sv2[N];
S-6 double dp[N], dv1[N], dv2[N];
S-7
S-8 void s_init(float *, float *, int);
S-9 void d_init(double *, double *, int);
S-10 void s_output(float *, int);
S-11 void d_output(double *, int);
S-12
S-13 #pragma omp declare target
S-14 void s_vec_mult_accum()
S-15 {
S-16     int i;
S-17
S-18     #pragma omp parallel for
S-19     for (i=0; i<N; i++)
S-20         sp[i] = sv1[i] * sv2[i];
S-21 }
S-22
S-23 void d_vec_mult_accum()
S-24 {
S-25     int i;
```

```

S-27     #pragma omp parallel for
S-28     for (i=0; i<N; i++)
S-29         dp[i] = dv1[i] * dv2[i];
S-30     }
S-31     #pragma omp end declare target
S-32
S-33     int main()
S-34     {
S-35         s_init(sv1, sv2, N);
S-36         #pragma omp target map(to:sv1,sv2) map(from:sp)
S-37             s_vec_mult_accum();
S-38         s_output(sp, N);
S-39
S-40         d_init(dv1, dv2, N);
S-41         #pragma omp target map(to:dv1,dv2) map(from:dp)
S-42             d_vec_mult_accum();
S-43         d_output(dp, N);
S-44
S-45         return 0;
S-46     }

```



1

Example declare_target.6.f90

```

S-1     module m_dat
S-2         integer, parameter :: N=100000000
S-3         !$omp declare target link(sp,sv1,sv2)
S-4         real :: sp(N), sv1(N), sv2(N)
S-5
S-6         !$omp declare target link(dp,dv1,dv2)
S-7         double precision :: dp(N), dv1(N), dv2(N)
S-8
S-9     contains
S-10         subroutine s_vec_mult_accum()
S-11             !$omp declare target
S-12             integer :: i
S-13
S-14             !$omp parallel do
S-15                 do i = 1,N
S-16                     sp(i) = sv1(i) * sv2(i)
S-17                 end do
S-18
S-19         end subroutine s_vec_mult_accum
S-20
S-21         subroutine d_vec_mult_accum()
S-22             !$omp declare target

```

```

S-23         integer :: i
S-24
S-25         !$omp parallel do
S-26         do i = 1,N
S-27             dp(i) = dv1(i) * dv2(i)
S-28         end do
S-29
S-30     end subroutine
S-31 end module m_dat
S-32
S-33 program prec_vec_mult
S-34     use m_dat
S-35
S-36     call s_init(sv1, sv2, N)
S-37     !$omp target map(to:sv1,sv2) map(from:sp)
S-38         call s_vec_mult_accum()
S-39     !$omp end target
S-40     call s_output(sp, N)
S-41
S-42     call d_init(dv1, dv2, N)
S-43     !$omp target map(to:dv1,dv2) map(from:dp)
S-44         call d_vec_mult_accum()
S-45     !$omp end target
S-46     call d_output(dp, N)
S-47
S-48 end program

```

Fortran

1 4.6 teams Constructs

2 4.6.1 target and teams Constructs with omp_get_num_teams 3 and omp_get_team_num Routines

4 The following example shows how the **target** and **teams** constructs are used to create a league
5 of thread teams that execute a region. The **teams** construct creates a league of at most two teams
6 where the master thread of each team executes the **teams** region.

7 The **omp_get_num_teams** routine returns the number of teams executing in a **teams** region.
8 The **omp_get_team_num** routine returns the team number, which is an integer between 0 and
9 one less than the value returned by **omp_get_num_teams**. The following example manually
10 distributes a loop across two teams.

▼ C / C++ ▼

11 *Example teams.1.c*

```
S-1 #include <stdlib.h>
S-2 #include <omp.h>
S-3 float dotprod(float B[], float C[], int N)
S-4 {
S-5     float sum0 = 0.0;
S-6     float sum1 = 0.0;
S-7     #pragma omp target map(to: B[:N], C[:N]) map(tofrom: sum0, sum1)
S-8     #pragma omp teams num_teams(2)
S-9     {
S-10         int i;
S-11         if (omp_get_num_teams() != 2)
S-12             abort();
S-13         if (omp_get_team_num() == 0)
S-14         {
S-15             #pragma omp parallel for reduction(+:sum0)
S-16             for (i=0; i<N/2; i++)
S-17                 sum0 += B[i] * C[i];
S-18         }
S-19         else if (omp_get_team_num() == 1)
S-20         {
S-21             #pragma omp parallel for reduction(+:sum1)
S-22             for (i=N/2; i<N; i++)
S-23                 sum1 += B[i] * C[i];
S-24         }
S-25     }
S-26     return sum0 + sum1;
S-27 }
S-28
S-29 /* Note: The variables sum0,sum1 are now mapped with tofrom, for correct
```

S-30 execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
S-31 */

Example teams.1.f90

```

S-1  function dotprod(B,C,N) result(sum)
S-2  use omp_lib, ONLY : omp_get_num_teams, omp_get_team_num
S-3      real      :: B(N), C(N), sum,sum0, sum1
S-4      integer :: N, i
S-5      sum0 = 0.0e0
S-6      sum1 = 0.0e0
S-7      !$omp target map(to: B, C) map(tofrom: sum0, sum1)
S-8      !$omp teams num_teams(2)
S-9          if (omp_get_num_teams() /= 2) stop "2 teams required"
S-10         if (omp_get_team_num() == 0) then
S-11             !$omp parallel do reduction(+:sum0)
S-12             do i=1,N/2
S-13                 sum0 = sum0 + B(i) * C(i)
S-14             end do
S-15         else if (omp_get_team_num() == 1) then
S-16             !$omp parallel do reduction(+:sum1)
S-17             do i=N/2+1,N
S-18                 sum1 = sum1 + B(i) * C(i)
S-19             end do
S-20         end if
S-21     !$omp end teams
S-22     !$omp end target
S-23     sum = sum0 + sum1
S-24 end function
S-25
S-26 ! Note: The variables sum0,sum1 are now mapped with tofrom, for correct
S-27 ! execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.

```

Fortran

1 4.6.2 target, teams, and distribute Constructs

2 The following example shows how the **target**, **teams**, and **distribute** constructs are used to
3 execute a loop nest in a **target** region. The **teams** construct creates a league and the master
4 thread of each team executes the **teams** region. The **distribute** construct schedules the
5 subsequent loop iterations across the master threads of each team.

6 The number of teams in the league is less than or equal to the variable *num_blocks*. Each team in
7 the league has a number of threads less than or equal to the variable *block_threads*. The iterations
8 in the outer loop are distributed among the master threads of each team.

9 When a team's master thread encounters the parallel loop construct before the inner loop, the other
10 threads in its team are activated. The team executes the **parallel** region and then workshares the
11 execution of the loop.

12 Each master thread executing the **teams** region has a private copy of the variable *sum* that is
13 created by the **reduction** clause on the **teams** construct. The master thread and all threads in
14 its team have a private copy of the variable *sum* that is created by the **reduction** clause on the
15 parallel loop construct. The second private *sum* is reduced into the master thread's private copy of
16 *sum* created by the **teams** construct. At the end of the **teams** region, each master thread's private
17 copy of *sum* is reduced into the final *sum* that is implicitly mapped into the **target** region.

▼ C / C++ ▼

18 *Example teams.2.c*

```
S-1 #define min(x, y) (((x) < (y)) ? (x) : (y))
S-2
S-3 float dotprod(float B[], float C[], int N, int block_size,
S-4 int num_teams, int block_threads)
S-5 {
S-6     float sum = 0.0;
S-7     int i, i0;
S-8     #pragma omp target map(to: B[0:N], C[0:N]) map(tofrom: sum)
S-9     #pragma omp teams num_teams(num_teams) thread_limit(block_threads) \
S-10         reduction(+:sum)
S-11     #pragma omp distribute
S-12     for (i0=0; i0<N; i0 += block_size)
S-13         #pragma omp parallel for reduction(+:sum)
S-14         for (i=i0; i< min(i0+block_size,N); i++)
S-15             sum += B[i] * C[i];
S-16     return sum;
S-17 }
S-18
S-19 /* Note: The variable sum is now mapped with tofrom, for correct
S-20 execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
S-21 */
```

▲ C / C++ ▲

Example teams.2.f90

```

S-1  function dotprod(B,C,N, block_size, num_teams, block_threads) result(sum)
S-2  implicit none
S-3      real    :: B(N), C(N), sum
S-4      integer :: N, block_size, num_teams, block_threads, i, i0
S-5      sum = 0.0e0
S-6      !$omp target map(to: B, C) map(tofrom: sum)
S-7      !$omp teams num_teams(num_teams) thread_limit(block_threads) &
S-8      !$omp& reduction(+:sum)
S-9      !$omp distribute
S-10     do i0=1,N, block_size
S-11         !$omp parallel do reduction(+:sum)
S-12         do i = i0, min(i0+block_size,N)
S-13             sum = sum + B(i) * C(i)
S-14         end do
S-15     end do
S-16     !$omp end teams
S-17     !$omp end target
S-18 end function
S-19
S-20 ! Note: The variable sum is now mapped with tofrom, for correct
S-21 ! execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.

```

4.6.3 target teams, and Distribute Parallel Loop Constructs

The following example shows how the **target teams** and distribute parallel loop constructs are used to execute a **target** region. The **target teams** construct creates a league of teams where the master thread of each team executes the **teams** region.

The distribute parallel loop construct schedules the loop iterations across the master threads of each team and then across the threads of each team.

1

Example teams.3.c

```

S-1 float dotprod(float B[], float C[], int N)
S-2 {
S-3     float sum = 0;
S-4     int i;
S-5     #pragma omp target teams map(to: B[0:N], C[0:N]) \
S-6                               defaultmap(tofrom:scalar) reduction(+:sum)
S-7     #pragma omp distribute parallel for reduction(+:sum)
S-8     for (i=0; i<N; i++)
S-9         sum += B[i] * C[i];
S-10    return sum;
S-11 }
S-12
S-13 /* Note: The variable sum is now mapped with tofrom from the defaultmap
S-14    clause on the combined target teams construct, for correct
S-15    execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
S-16 */

```

2

Example teams.3.f90

```

S-1 function dotprod(B,C,N) result(sum)
S-2     real    :: B(N), C(N), sum
S-3     integer :: N, i
S-4     sum = 0.0e0
S-5     !$omp target teams map(to: B, C) &
S-6     !$omp&                defaultmap(tofrom:scalar) reduction(+:sum)
S-7     !$omp distribute parallel do reduction(+:sum)
S-8         do i = 1,N
S-9             sum = sum + B(i) * C(i)
S-10        end do
S-11    !$omp end target teams
S-12 end function
S-13
S-14 ! Note: The variable sum is now mapped with tofrom from the defaultmap
S-15 ! clause on the combined target teams construct, for correct
S-16 ! execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.

```

1 4.6.4 target teams and Distribute Parallel Loop Constructs 2 with Scheduling Clauses

3 The following example shows how the **target teams** and distribute parallel loop constructs are
4 used to execute a **target** region. The **teams** construct creates a league of at most eight teams
5 where the master thread of each team executes the **teams** region. The number of threads in each
6 team is less than or equal to 16.

7 The **distribute** parallel loop construct schedules the subsequent loop iterations across the
8 master threads of each team and then across the threads of each team.

9 The **dist_schedule** clause on the distribute parallel loop construct indicates that loop iterations
10 are distributed to the master thread of each team in chunks of 1024 iterations.

11 The **schedule** clause indicates that the 1024 iterations distributed to a master thread are then
12 assigned to the threads in its associated team in chunks of 64 iterations.

▼ C / C++ ▼

13 *Example teams.4.c*

```
S-1 #define N 1024*1024
S-2 float dotprod(float B[], float C[])
S-3 {
S-4     float sum = 0.0;
S-5     int i;
S-6     #pragma omp target map(to: B[0:N], C[0:N]) map(tofrom: sum)
S-7     #pragma omp teams num_teams(8) thread_limit(16) reduction(+:sum)
S-8     #pragma omp distribute parallel for reduction(+:sum) \
S-9         dist_schedule(static, 1024) schedule(static, 64)
S-10    for (i=0; i<N; i++)
S-11        sum += B[i] * C[i];
S-12    return sum;
S-13 }
S-14
S-15 /* Note: The variable sum is now mapped with tofrom, for correct
S-16    execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
S-17    */
```

▲ C / C++ ▲

1

Example teams.4.f90

```

S-1  module arrays
S-2  integer,parameter :: N=1024*1024
S-3  real :: B(N), C(N)
S-4  end module
S-5  function dotprod() result(sum)
S-6  use arrays
S-7      real :: sum
S-8      integer :: i
S-9      sum = 0.0e0
S-10     !$omp target map(to: B, C) map(tofrom: sum)
S-11     !$omp teams num_teams(8) thread_limit(16) reduction(+:sum)
S-12     !$omp distribute parallel do reduction(+:sum) &
S-13     !$omp& dist_schedule(static, 1024) schedule(static, 64)
S-14         do i = 1,N
S-15             sum = sum + B(i) * C(i)
S-16         end do
S-17     !$omp end teams
S-18     !$omp end target
S-19 end function
S-20
S-21 ! Note: The variable sum is now mapped with tofrom, for correct
S-22 ! execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.

```

2 4.6.5 target teams and distribute simd Constructs

3 The following example shows how the **target teams** and **distribute simd** constructs are
 4 used to execute a loop in a **target** region. The **target teams** construct creates a league of
 5 teams where the master thread of each team executes the **teams** region.

6 The **distribute simd** construct schedules the loop iterations across the master thread of each
 7 team and then uses SIMD parallelism to execute the iterations.

1 *Example teams.5.c*

```

S-1 extern void init(float *, float *, int);
S-2 extern void output(float *, int);
S-3 void vec_mult(float *p, float *v1, float *v2, int N)
S-4 {
S-5     int i;
S-6     init(v1, v2, N);
S-7     #pragma omp target teams map(to: v1[0:N], v2[:N]) map(from: p[0:N])
S-8     #pragma omp distribute simd
S-9     for (i=0; i<N; i++)
S-10        p[i] = v1[i] * v2[i];
S-11     output(p, N);
S-12 }

```

2 *Example teams.5.f90*

```

S-1 subroutine vec_mult(p, v1, v2, N)
S-2     real    :: p(N), v1(N), v2(N)
S-3     integer :: i
S-4     call init(v1, v2, N)
S-5     !$omp target teams map(to: v1, v2) map(from: p)
S-6     !$omp distribute simd
S-7     do i=1,N
S-8         p(i) = v1(i) * v2(i)
S-9     end do
S-10    !$omp end target teams
S-11    call output(p, N)
S-12 end subroutine

```

1 4.6.6 target teams and Distribute Parallel Loop SIMD 2 Constructs

3 The following example shows how the **target teams** and the distribute parallel loop SIMD
4 constructs are used to execute a loop in a **target teams** region. The **target teams** construct
5 creates a league of teams where the master thread of each team executes the **teams** region.

6 The distribute parallel loop SIMD construct schedules the loop iterations across the master thread
7 of each team and then across the threads of each team where each thread uses SIMD parallelism.

▼ C / C++ ▼

8 *Example teams.6.c*

```
S-1 extern void init(float *, float *, int);  
S-2 extern void output(float *, int);  
S-3 void vec_mult(float *p, float *v1, float *v2, int N)  
S-4 {  
S-5     int i;  
S-6     init(v1, v2, N);  
S-7     #pragma omp target teams map(to: v1[0:N], v2[:N]) map(from: p[0:N])  
S-8     #pragma omp distribute parallel for simd  
S-9     for (i=0; i<N; i++)  
S-10         p[i] = v1[i] * v2[i];  
S-11     output(p, N);  
S-12 }
```

▲ C / C++ ▲
▼ Fortran ▼

9 *Example teams.6.f90*

```
S-1 subroutine vec_mult(p, v1, v2, N)  
S-2     real :: p(N), v1(N), v2(N)  
S-3     integer :: i  
S-4     call init(v1, v2, N)  
S-5     !$omp target teams map(to: v1, v2) map(from: p)  
S-6     !$omp distribute parallel do simd  
S-7         do i=1,N  
S-8             p(i) = v1(i) * v2(i)  
S-9         end do  
S-10     !$omp end target teams  
S-11     call output(p, N)  
S-12 end subroutine
```

▲ Fortran ▲

1 4.7 Asynchronous **target** Execution and Dependences

2 Asynchronous execution of a **target** region can be accomplished by creating an explicit task
3 around the **target** region. Examples with explicit tasks are shown at the beginning of this section.

4 As of OpenMP 4.5 and beyond the **nowait** clause can be used on the **target** directive for
5 asynchronous execution. Examples with **nowait** clauses follow the explicit **task** examples.

6 This section also shows the use of **depend** clauses to order executions through dependences.

7 4.7.1 Asynchronous **target** with Tasks

8 The following example shows how the **task** and **target** constructs are used to execute multiple
9 **target** regions asynchronously. The task that encounters the **task** construct generates an
10 explicit task that contains a **target** region. The thread executing the explicit task encounters a
11 task scheduling point while waiting for the execution of the **target** region to complete, allowing
12 the thread to switch back to the execution of the encountering task or one of the previously
13 generated explicit tasks.

▼ C / C++ ▼

14 *Example `async_target.1.c`*

```
S-1  #pragma omp declare target
S-2  float F(float);
S-3  #pragma omp end declare target
S-4  #define N 1000000000
S-5  #define CHUNKSZ 1000000
S-6  void init(float *, int);
S-7  float Z[N];
S-8  void pipedF()
S-9  {
S-10     int C, i;
S-11     init(Z, N);
S-12     for (C=0; C<N; C+=CHUNKSZ)
S-13     {
S-14         #pragma omp task shared(Z)
S-15         #pragma omp target map(Z[C:CHUNKSZ])
S-16         #pragma omp parallel for
S-17         for (i=0; i<CHUNKSZ; i++)
S-18             Z[i] = F(Z[i]);
S-19     }
S-20     #pragma omp taskwait
S-21 }
```


1 The Fortran version has an interface block that contains the **declare target**. An identical
 2 statement exists in the function declaration (not shown here).

3 *Example async_target.1.f90*

```

S-1 module parameters
S-2 integer, parameter :: N=1000000000, CHUNKSZ=1000000
S-3 end module
S-4 subroutine pipedF()
S-5 use parameters, ONLY: N, CHUNKSZ
S-6 integer          :: C, i
S-7 real            :: z(N)
S-8
S-9 interface
S-10   function F(z)
S-11   !$omp declare target
S-12   real, intent(IN) :: z
S-13   real            :: F
S-14   end function F
S-15 end interface
S-16
S-17   call init(z,N)
S-18
S-19   do C=1,N,CHUNKSZ
S-20
S-21     !$omp task shared(z)
S-22     !$omp target map(z(C:C+CHUNKSZ-1))
S-23     !$omp parallel do
S-24       do i=C,C+CHUNKSZ-1
S-25         z(i) = F(z(i))
S-26       end do
S-27     !$omp end target
S-28     !$omp end task
S-29
S-30   end do
S-31   !$omp taskwait
S-32   print*, z
S-33
S-34 end subroutine pipedF

```

4 The following example shows how the **task** and **target** constructs are used to execute multiple
 5 **target** regions asynchronously. The task dependence ensures that the storage is allocated and
 6 initialized on the device before it is accessed.

Example async_target.2.c

```

S-1  #include <stdlib.h>
S-2  #include <omp.h>
S-3  #pragma omp declare target
S-4  extern void init(float *, float *, int);
S-5  #pragma omp end declare target
S-6  extern void foo();
S-7  extern void output(float *, int);
S-8  void vec_mult(float *p, int N, int dev)
S-9  {
S-10     float *v1, *v2;
S-11     int i;
S-12     #pragma omp task shared(v1, v2) depend(out: v1, v2)
S-13     #pragma omp target device(dev) map(v1, v2)
S-14     {
S-15         // check whether on device dev
S-16         if (omp_is_initial_device())
S-17             abort();
S-18         v1 = malloc(N*sizeof(float));
S-19         v2 = malloc(N*sizeof(float));
S-20         init(v1, v2, N);
S-21     }
S-22     foo(); // execute other work asynchronously
S-23     #pragma omp task shared(v1, v2, p) depend(in: v1, v2)
S-24     #pragma omp target device(dev) map(to: v1, v2) map(from: p[0:N])
S-25     {
S-26         // check whether on device dev
S-27         if (omp_is_initial_device())
S-28             abort();
S-29         #pragma omp parallel for
S-30         for (i=0; i<N; i++)
S-31             p[i] = v1[i] * v2[i];
S-32         free(v1);
S-33         free(v2);
S-34     }
S-35     #pragma omp taskwait
S-36     output(p, N);
S-37 }

```

The Fortran example below is similar to the C version above. Instead of pointers, though, it uses the convenience of Fortran allocatable arrays on the device. In order to preserve the arrays allocated on the device across multiple **target** regions, a **target data** region is used in this case.

If there is no shape specified for an allocatable array in a **map** clause, only the array descriptor (also called a dope vector) is mapped. That is, device space is created for the descriptor, and it is initially populated with host values. In this case, the *v1* and *v2* arrays will be in a non-associated state on the device. When space for *v1* and *v2* is allocated on the device in the first **target** region the addresses to the space will be included in their descriptors.

At the end of the first **target** region, the arrays *v1* and *v2* are preserved on the device for access in the second **target** region. At the end of the second **target** region, the data in array *p* is copied back, the arrays *v1* and *v2* are not.

A **depend** clause is used in the **task** directive to provide a wait at the beginning of the second **target** region, to insure that there is no race condition with *v1* and *v2* in the two tasks. It would be noncompliant to use *v1* and/or *v2* in lieu of *N* in the **depend** clauses, because the use of non-allocated allocatable arrays as list items in a **depend** clause would lead to unspecified behavior.

Note – This example is not strictly compliant with the OpenMP 4.5 specification since the allocation status of allocatable arrays *v1* and *v2* is changed inside the **target** region, which is not allowed. (See the restrictions for the **map** clause in the *Data-mapping Attribute Rules and Clauses* section of the specification.) However, the intention is to relax the restrictions on mapping of allocatable variables in the next release of the specification so that the example will be compliant.

Fortran

Example async_target.2.f90

```

S-1  subroutine mult(p,  N, idev)
S-2      use omp_lib, ONLY: omp_is_initial_device
S-3      real                :: p(N)
S-4      real,allocatable :: v1(:), v2(:)
S-5      integer :: i, idev
S-6      !$omp declare target (init)
S-7
S-8      !$omp target data map(v1,v2)
S-9
S-10     !$omp task shared(v1,v2) depend(out: N)
S-11         !$omp target device(idev)
S-12             if( omp_is_initial_device() ) &
S-13                 stop "not executing on target device"
S-14             allocate(v1(N), v2(N))
S-15             call init(v1,v2,N)
S-16         !$omp end target
S-17     !$omp end task
S-18
S-19     call foo()  ! execute other work asynchronously
S-20
S-21     !$omp task shared(v1,v2,p) depend(in: N)
S-22         !$omp target device(idev) map(from: p)

```

```

S-23         if( omp_is_initial_device() ) &
S-24             stop "not executing on target device"
S-25         !$omp parallel do
S-26             do i = 1,N
S-27                 p(i) = v1(i) * v2(i)
S-28             end do
S-29         deallocate(v1,v2)
S-30
S-31         !$omp end target
S-32     !$omp end task
S-33
S-34     !$omp taskwait
S-35
S-36     !$omp end target data
S-37
S-38     call output(p, N)
S-39
S-40 end subroutine

```

Fortran

1 4.7.2 **nowait** Clause on target Construct

2 The following example shows how to execute code asynchronously on a device without an explicit
3 task. The **nowait** clause on a **target** construct allows the thread of the *target task* to perform
4 other work while waiting for the **target** region execution to complete. Hence, the the **target**
5 region can execute asynchronously on the device (without requiring a host thread to idle while
6 waiting for the *target task* execution to complete).

7 In this example the product of two vectors (arrays), *v1* and *v2*, is formed. One half of the operations
8 is performed on the device, and the last half on the host, concurrently.

9 After a team of threads is formed the master thread generates the *target task* while the other threads
10 can continue on, without a barrier, to the execution of the host portion of the vector product. The
11 completion of the *target task* (asynchronous target execution) is guaranteed by the synchronization
12 in the implicit barrier at the end of the host vector-product worksharing loop region. See the
13 **barrier** glossary entry in the OpenMP specification for details.

14 The host loop scheduling is **dynamic**, to balance the host thread executions, since one thread is
15 being used for offload generation. In the situation where little time is spent by the *target task* in
16 setting up and tearing down the the target execution, **static** scheduling may be desired.

1

Example async_target.3.c

```

S-1
S-2 #include <stdio.h>
S-3
S-4 #define N 1000000 //N must be even
S-5 void init(int n, float *v1, float *v2);
S-6
S-7 int main(){
S-8     int i, n=N;
S-9     int chunk=1000;
S-10    float v1[N],v2[N],vxv[N];
S-11
S-12    init(n, v1,v2);
S-13
S-14    #pragma omp parallel
S-15    {
S-16
S-17        #pragma omp master
S-18        #pragma omp target teams distribute parallel for nowait \
S-19                                map(to: v1[0:n/2]) \
S-20                                map(to: v2[0:n/2]) \
S-21                                map(from: vxv[0:n/2])
S-22        for(i=0; i<n/2; i++){ vxv[i] = v1[i]*v2[i]; }
S-23
S-24        #pragma omp for schedule(dynamic,chunk)
S-25        for(i=n/2; i<n; i++){ vxv[i] = v1[i]*v2[i]; }
S-26
S-27    }
S-28    printf(" vxv[0] vxv[n-1] %f %f\n", vxv[0], vxv[n-1]);
S-29    return 0;
S-30 }

```

2

Example async_target.3.f90

```

S-1
S-2 program concurrent_async
S-3     use omp_lib
S-4     integer,parameter :: n=1000000 !!n must be even
S-5     integer           :: i, chunk=1000
S-6     real              :: v1(n),v2(n),vxv(n)
S-7
S-8     call init(n, v1,v2)
S-9

```

```

S-10      !$omp parallel
S-11
S-12          !$omp master
S-13          !$omp target teams distribute parallel do nowait &
S-14          !$omp&                                map(to: v1(1:n/2))    &
S-15          !$omp&                                map(to: v2(1:n/2))    &
S-16          !$omp&                                map(from: vxv(1:n/2))
S-17          do i = 1,n/2;    vxv(i) = v1(i)*v2(i); end do
S-18          !$omp end master
S-19
S-20          !$omp do schedule(dynamic,chunk)
S-21          do i = n/2+1,n;    vxv(i) = v1(i)*v2(i); end do
S-22
S-23      !$omp end parallel
S-24
S-25      print*, " vxv(1) vxv(n) :", vxv(1), vxv(n)
S-26
S-27  end program

```

Fortran

1 4.7.3 Asynchronous target with nowait and depend 2 Clauses

3 More details on dependences can be found in Section 3.3 on page 73, Task Dependences. In this
4 example, there are three flow dependences. In the first two dependences the target task does not
5 execute until the preceding explicit tasks have finished. These dependences are produced by arrays
6 *v1* and *v2* with the **out** dependence type in the first two tasks, and the **in** dependence type in the
7 target task.

8 The last dependence is produced by array *p* with the **out** dependence type in the target task, and
9 the **in** dependence type in the last task. The last task does not execute until the target task finishes.

10 The **nowait** clause on the **target** construct creates a deferrable *target task*, allowing the
11 encountering task to continue execution without waiting for the completion of the *target task*.

1

Example async_target.4.c

```

S-1
S-2  extern void init( float*, int);
S-3  extern void output(float*, int);
S-4
S-5  void vec_mult(int N)
S-6  {
S-7      int i;
S-8      float p[N], v1[N], v2[N];
S-9
S-10     #pragma omp parallel num_threads(2)
S-11     {
S-12         #pragma omp single
S-13         {
S-14             #pragma omp task depend(out:v1)
S-15             init(v1, N);
S-16
S-17             #pragma omp task depend(out:v2)
S-18             init(v2, N);
S-19
S-20             #pragma omp target nowait depend(in:v1,v2) depend(out:p) \
S-21                 map(to:v1,v2) map( from: p)
S-22             #pragma omp parallel for private(i)
S-23             for (i=0; i<N; i++)
S-24                 p[i] = v1[i] * v2[i];
S-25
S-26             #pragma omp task depend(in:p)
S-27             output(p, N);
S-28         }
S-29     }
S-30 }

```

2

Example async_target.4.f90

```

S-1
S-2  subroutine vec_mult(N)
S-3      implicit none
S-4      integer          :: i, N
S-5      real, allocatable :: p(:), v1(:), v2(:)
S-6      allocate( p(N), v1(N), v2(N) )
S-7
S-8      !$omp parallel num_threads(2)
S-9

```

```

S-10      !$omp single
S-11
S-12      !$omp task depend(out:v1)
S-13      call init(v1, N)
S-14      !$omp end task
S-15
S-16      !$omp task depend(out:v2)
S-17      call init(v2, N)
S-18      !$omp end task
S-19
S-20      !$omp target nowait depend(in:v1,v2) depend(out:p) &
S-21      !$omp&                                map(to:v1,v2)  map(from: p)
S-22      !$omp parallel do
S-23      do i=1,N
S-24          p(i) = v1(i) * v2(i)
S-25      end do
S-26      !$omp end target
S-27
S-28
S-29      !$omp task depend(in:p)
S-30      call output(p, N)
S-31      !$omp end task
S-32
S-33      !$omp end single
S-34      !$omp end parallel
S-35
S-36      deallocate( p, v1, v2 )
S-37
S-38      end subroutine

```

Fortran

1 4.8 Array Sections in Device Constructs

2 The following examples show the usage of array sections in **map** clauses on **target** and **target**
3 **data** constructs.

4 This example shows the invalid usage of two separate sections of the same array inside of a
5 **target** construct.

▼ C / C++ ▼

6 *Example array_sections.1.c*

```
S-1 void foo ()  
S-2 {  
S-3     int A[30];  
S-4     #pragma omp target data map( A[0:4] )  
S-5     {  
S-6         /* Cannot map distinct parts of the same array */  
S-7         #pragma omp target map( A[7:20] )  
S-8         {  
S-9             A[2] = 0;  
S-10        }  
S-11    }  
S-12 }
```

▲ C / C++ ▲
▼ Fortran ▼

7 *Example array_sections.1.f90*

```
S-1 subroutine foo()  
S-2 integer :: A(30)  
S-3     A = 1  
S-4     !$omp target data map( A(1:4) )  
S-5         ! Cannot map distinct parts of the same array  
S-6         !$omp target map( A(8:27) )  
S-7         A(3) = 0  
S-8         !$omp end target  
S-9         !$omp end target data  
S-10 end subroutine
```

▲ Fortran ▲

8 This example shows the invalid usage of two separate sections of the same array inside of a
9 **target** construct.

1 *Example array_sections.2.c*

```

S-1 void foo ()
S-2 {
S-3     int A[30], *p;
S-4     #pragma omp target data map( A[0:4] )
S-5     {
S-6         p = &A[0];
S-7         /* invalid because p[3] and A[3] are the same
S-8          * location on the host but the array section
S-9          * specified via p[...] is not a subset of A[0:4] */
S-10        #pragma omp target map( p[3:20] )
S-11        {
S-12            A[2] = 0;
S-13            p[8] = 0;
S-14        }
S-15    }
S-16 }

```

2 *Example array_sections.2.f90*

```

S-1 subroutine foo()
S-2 integer,target :: A(30)
S-3 integer,pointer :: p(:)
S-4     A=1
S-5     !$omp target data map( A(1:4) )
S-6     p=>A
S-7     ! invalid because p(4) and A(4) are the same
S-8     ! location on the host but the array section
S-9     ! specified via p(...) is not a subset of A(1:4)
S-10    !$omp target map( p(4:23) )
S-11        A(3) = 0
S-12        p(9) = 0
S-13    !$omp end target
S-14    !$omp end target data
S-15 end subroutine

```

3 This example shows the valid usage of two separate sections of the same array inside of a **target**
 4 construct.

C / C++

1 *Example array_sections.3.c*

```

S-1  void foo ()
S-2  {
S-3      int A[30], *p;
S-4      #pragma omp target data map( A[0:4] )
S-5      {
S-6          p = &A[0];
S-7          #pragma omp target map( p[7:20] )
S-8          {
S-9              A[2] = 0;
S-10             p[8] = 0;
S-11         }
S-12     }
S-13 }
```

C / C++

Fortran

2 *Example array_sections.3.f90*

```

S-1  subroutine foo()
S-2  integer,target  :: A(30)
S-3  integer,pointer :: p(:)
S-4      !$omp target data map( A(1:4) )
S-5      p=>A
S-6      !$omp target map( p(8:27) )
S-7          A(3) = 0
S-8          p(9) = 0
S-9      !$omp end target
S-10     !$omp end target data
S-11 end subroutine
```

Fortran

3 This example shows the valid usage of a wholly contained array section of an already mapped array
 4 section inside of a **target** construct.

1 *Example array_sections.4.c*

```

S-1 void foo ()
S-2 {
S-3     int A[30], *p;
S-4     #pragma omp target data map( A[0:10] )
S-5     {
S-6         p = &A[0];
S-7         #pragma omp target map( p[3:7] )
S-8         {
S-9             A[2] = 0;
S-10            p[8] = 0;
S-11            A[8] = 1;
S-12        }
S-13    }
S-14 }

```

2 *Example array_sections.4.f90*

```

S-1 subroutine foo()
S-2 integer,target :: A(30)
S-3 integer,pointer :: p(:)
S-4 !$omp target data map( A(1:10) )
S-5     p=>A
S-6 !$omp target map( p(4:10) )
S-7     A(3) = 0
S-8     p(9) = 0
S-9     A(9) = 1
S-10 !$omp end target
S-11 !$omp end target data
S-12 end subroutine

```

1 4.9 Device Routines

2 4.9.1 omp_is_initial_device Routine

3 The following example shows how the **omp_is_initial_device** runtime library routine can
4 be used to query if a code is executing on the initial host device or on a target device. The example
5 then sets the number of threads in the **parallel** region based on where the code is executing.

▶ C / C++ ◀

6 *Example device.1.c*

```
S-1  #include <stdio.h>
S-2  #include <omp.h>
S-3  #pragma omp declare target
S-4  void vec_mult(float *p, float *v1, float *v2, int N);
S-5  extern float *p, *v1, *v2;
S-6  extern int N;
S-7  #pragma omp end declare target
S-8  extern void init_vars(float *, float *, int);
S-9  extern void output(float *, int);
S-10 void foo()
S-11 {
S-12     init_vars(v1, v2, N);
S-13     #pragma omp target device(42) map(p[:N], v1[:N], v2[:N])
S-14     {
S-15         vec_mult(p, v1, v2, N);
S-16     }
S-17     output(p, N);
S-18 }
S-19 void vec_mult(float *p, float *v1, float *v2, int N)
S-20 {
S-21     int i;
S-22     int nthreads;
S-23     if (!omp_is_initial_device())
S-24     {
S-25         printf("1024 threads on target device\n");
S-26         nthreads = 1024;
S-27     }
S-28     else
S-29     {
S-30         printf("8 threads on initial device\n");
S-31         nthreads = 8;
S-32     }
S-33     #pragma omp parallel for private(i) num_threads(nthreads)
S-34     for (i=0; i<N; i++)
```

```
S-35     p[i] = v1[i] * v2[i];
S-36 }
```

C / C++

Fortran

1

Example device.f90

```
S-1  module params
S-2      integer,parameter :: N=1024
S-3  end module params
S-4  module vmult
S-5      contains
S-6      subroutine vec_mult(p, v1, v2, N)
S-7          use omp_lib, ONLY : omp_is_initial_device
S-8          !$omp declare target
S-9          real    :: p(N), v1(N), v2(N)
S-10         integer :: i, nthreads, N
S-11         if (.not. omp_is_initial_device()) then
S-12             print*, "1024 threads on target device"
S-13             nthreads = 1024
S-14         else
S-15             print*, "8 threads on initial device"
S-16             nthreads = 8
S-17         endif
S-18         !$omp parallel do private(i) num_threads(nthreads)
S-19         do i = 1,N
S-20             p(i) = v1(i) * v2(i)
S-21         end do
S-22     end subroutine vec_mult
S-23 end module vmult
S-24 program prog_vec_mult
S-25     use params
S-26     use vmult
S-27     real :: p(N), v1(N), v2(N)
S-28     call init(v1,v2,N)
S-29     !$omp target device(42) map(p, v1, v2)
S-30     call vec_mult(p, v1, v2, N)
S-31     !$omp end target
S-32     call output(p, N)
S-33 end program
```

Fortran

1 4.9.2 omp_get_num_devices Routine

2 The following example shows how the `omp_get_num_devices` runtime library routine can be
3 used to determine the number of devices.

▼ C / C++ ▼

4 *Example device.2.c*

```
S-1 #include <omp.h>
S-2 extern void init(float *, float *, int);
S-3 extern void output(float *, int);
S-4 void vec_mult(float *p, float *v1, float *v2, int N)
S-5 {
S-6     int i;
S-7     init(v1, v2, N);
S-8     int ndev = omp_get_num_devices();
S-9     int do_offload = (ndev>0 && N>1000000);
S-10    #pragma omp target if(do_offload) map(to: v1[0:N], v2[:N]) map(from: p[0:N])
S-11    #pragma omp parallel for if(N>1000) private(i)
S-12    for (i=0; i<N; i++)
S-13        p[i] = v1[i] * v2[i];
S-14    output(p, N);
S-15 }
```

▲ C / C++ ▲
▼ Fortran ▼

5 *Example device.2.f90*

```
S-1 subroutine vec_mult(p, v1, v2, N)
S-2 use omp_lib, ONLY : omp_get_num_devices
S-3 real :: p(N), v1(N), v2(N)
S-4 integer :: N, i, ndev
S-5 logical :: do_offload
S-6 call init(v1, v2, N)
S-7 ndev = omp_get_num_devices()
S-8 do_offload = (ndev>0) .and. (N>1000000)
S-9 !$omp target if(do_offload) map(to: v1, v2) map(from: p)
S-10 !$omp parallel do if(N>1000)
S-11 do i=1,N
S-12 p(i) = v1(i) * v2(i)
S-13 end do
S-14 !$omp end target
S-15 call output(p, N)
S-16 end subroutine
```

▲ Fortran ▲

1 4.9.3 omp_set_default_device and 2 omp_get_default_device Routines

3 The following example shows how the **omp_set_default_device** and
4 **omp_get_default_device** runtime library routines can be used to set the default device and
5 determine the default device respectively.

C / C++

6 *Example device.3.c*

```
S-1 #include <omp.h>
S-2 #include <stdio.h>
S-3 void foo(void)
S-4 {
S-5     int default_device = omp_get_default_device();
S-6     printf("Default device = %d\n", default_device);
S-7     omp_set_default_device(default_device+1);
S-8     if (omp_get_default_device() != default_device+1)
S-9         printf("Default device is still = %d\n", default_device);
S-10 }
```

C / C++

Fortran

7 *Example device.3.f90*

```
S-1 program foo
S-2 use omp_lib, ONLY : omp_get_default_device, omp_set_default_device
S-3 integer :: old_default_device, new_default_device
S-4     old_default_device = omp_get_default_device()
S-5     print*, "Default device = ", old_default_device
S-6     new_default_device = old_default_device + 1
S-7     call omp_set_default_device(new_default_device)
S-8     if (omp_get_default_device() == old_default_device) &
S-9         print*, "Default device is STILL = ", old_default_device
S-10 end program
```

Fortran

1 4.9.4 Target Memory and Device Pointers Routines

2 The following example shows how to create space on a device, transfer data to and from that space,
3 and free the space, using API calls. The API calls directly execute allocation, copy and free
4 operations on the device, without invoking any mapping through a **target** directive. The
5 **omp_target_alloc** routine allocates space and returns a device pointer for referencing the
6 space in the **omp_target_memcpy** API routine on the host. The **omp_target_free** routine
7 frees the space on the device.

8 The example also illustrates how to access that space in a **target** region by exposing the device
9 pointer in an **is_device_ptr** clause.

10 The example creates an array of cosine values on the default device, to be used on the host device.
11 The function fails if a default device is not available.

▼ C / C++ ▼

12 *Example device.4.c*

```
S-1 #include <stdio.h>
S-2 #include <math.h>
S-3 #include <stdlib.h>
S-4 #include <omp.h>
S-5
S-6 void get_dev_cos(double *mem, size_t s)
S-7 {
S-8     int h, t, i;
S-9     double * mem_dev_cpy;
S-10    h = omp_get_initial_device();
S-11    t = omp_get_default_device();
S-12
S-13    if (omp_get_num_devices() < 1 || t < 0){
S-14        printf(" ERROR: No device found.\n");
S-15        exit(1);
S-16    }
S-17
S-18    mem_dev_cpy = omp_target_alloc( sizeof(double) * s, t);
S-19    if(mem_dev_cpy == NULL){
S-20        printf(" ERROR: No space left on device.\n");
S-21        exit(1);
S-22    }
S-23
S-24                                /* dst  src */
S-25    omp_target_memcpy(mem_dev_cpy, mem, sizeof(double)*s,
S-26                      0, 0,
S-27                      t, h);
S-28
S-29    #pragma omp target is_device_ptr(mem_dev_cpy) device(t)
```

```

S-30 #pragma omp teams distribute parallel for
S-31     for(i=0;i<s;i++){ mem_dev_cpy[i] = cos((double)i); } /* init data */
S-32
S-33         /* dst  src */
S-34     omp_target_memcpy(mem, mem_dev_cpy, sizeof(double)*s,
S-35                       0,          0,
S-36                       h,          t);
S-37
S-38     omp_target_free(mem_dev_cpy, t);
S-39 }

```

▲ ————— C / C++ ————— ▲

SIMD

3 Single instruction, multiple data (SIMD) is a form of parallel execution in which the same operation
4 is performed on multiple data elements independently in hardware vector processing units (VPU),
5 also called SIMD units. The addition of two vectors to form a third vector is a SIMD operation.
6 Many processors have SIMD (vector) units that can perform simultaneously 2, 4, 8 or more
7 executions of the same operation (by a single SIMD unit).

8 Loops without loop-carried backward dependency (or with dependency preserved using ordered
9 `simd`) are candidates for vectorization by the compiler for execution with SIMD units. In addition,
10 with state-of-the-art vectorization technology and **`declare simd`** construct extensions for
11 function vectorization in the OpenMP 4.5 specification, loops with function calls can be vectorized
12 as well. The basic idea is that a scalar function call in a loop can be replaced by a vector version of
13 the function, and the loop can be vectorized simultaneously by combining a loop vectorization
14 (**`simd`** directive on the loop) and a function vectorization (**`declare simd`** directive on the
15 function).

16 A **`simd`** construct states that SIMD operations be performed on the data within the loop. A number
17 of clauses are available to provide data-sharing attributes (**`private`**, **`linear`**, **`reduction`** and
18 **`lastprivate`**). Other clauses provide vector length preference/restrictions (**`simdlen`** /
19 **`safelen`**), loop fusion (**`collapse`**), and data alignment (**`aligned`**).

20 The **`declare simd`** directive designates that a vector version of the function should also be
21 constructed for execution within loops that contain the function and have a **`simd`** directive. Clauses
22 provide argument specifications (**`linear`**, **`uniform`**, and **`aligned`**), a requested vector length
23 (**`simdlen`**), and designate whether the function is always/never called conditionally in a loop
24 (**`branch/inbranch`**). The latter is for optimizing performance.

25 Also, the **`simd`** construct has been combined with the worksharing loop constructs (**`for simd`**
26 and **`do simd`**) to enable simultaneous thread execution in different SIMD units.

1 5.1 simd and declare simd Constructs

2 The following example illustrates the basic use of the **simd** construct to assure the compiler that
3 the loop can be vectorized.

▼ C / C++ ▼

4 *Example SIMD.1.c*

```
S-1 void star( double *a, double *b, double *c, int n, int *ioff )  
S-2 {  
S-3     int i;  
S-4     #pragma omp simd  
S-5     for ( i = 0; i < n; i++ )  
S-6         a[i] *= b[i] * c[i+ *ioff];  
S-7 }
```

▲ C / C++ ▲

▼ Fortran ▼

5 *Example SIMD.1.f90*

```
S-1 subroutine star(a,b,c,n,ioff_ptr)  
S-2     implicit none  
S-3     double precision :: a(*),b(*),c(*)  
S-4     integer          :: n, i  
S-5     integer, pointer :: ioff_ptr  
S-6  
S-7     !$omp simd  
S-8     do i = 1,n  
S-9         a(i) = a(i) * b(i) * c(i+ioff_ptr)  
S-10    end do  
S-11  
S-12 end subroutine
```

When a function can be inlined within a loop the compiler has an opportunity to vectorize the loop. By guaranteeing SIMD behavior of a function's operations, characterizing the arguments of the function and privatizing temporary variables of the loop, the compiler can often create faster, vector code for the loop. In the examples below the **declare simd** construct is used on the *add1* and *add2* functions to enable creation of their corresponding SIMD function versions for execution within the associated SIMD loop. The functions characterize two different approaches of accessing data within the function: by a single variable and as an element in a data array, respectively. The *add3* C function uses dereferencing.

The **declare simd** constructs also illustrate the use of **uniform** and **linear** clauses. The **uniform(fact)** clause indicates that the variable *fact* is invariant across the SIMD lanes. In the *add2* function *a* and *b* are included in the **uniform** list because the C pointer and the Fortran array references are constant. The *i* index used in the *add2* function is included in a **linear** clause with a constant-linear-step of 1, to guarantee a unity increment of the associated loop. In the **declare simd** construct for the *add3* C function the **linear(a,b:1)** clause instructs the compiler to generate unit-stride loads across the SIMD lanes; otherwise, costly *gather* instructions would be generated for the unknown sequence of access of the pointer dereferences.

In the **simd** constructs for the loops the **private(tmp)** clause is necessary to assure that the each vector operation has its own *tmp* variable.

Example SIMD.2.c

```
S-1  #include <stdio.h>
S-2
S-3  #pragma omp declare simd uniform(fact)
S-4  double add1(double a, double b, double fact)
S-5  {
S-6      double c;
S-7      c = a + b + fact;
S-8      return c;
S-9  }
S-10
S-11 #pragma omp declare simd uniform(a,b,fact) linear(i:1)
S-12 double add2(double *a, double *b, int i, double fact)
S-13 {
S-14     double c;
S-15     c = a[i] + b[i] + fact;
S-16     return c;
S-17 }
S-18
S-19 #pragma omp declare simd uniform(fact) linear(a,b:1)
S-20 double add3(double *a, double *b, double fact)
S-21 {
```

```

S-22     double c;
S-23     c = *a + *b + fact;
S-24     return c;
S-25 }
S-26
S-27 void work( double *a, double *b, int n )
S-28 {
S-29     int i;
S-30     double tmp;
S-31     #pragma omp simd private(tmp)
S-32     for ( i = 0; i < n; i++ ) {
S-33         tmp = add1( a[i], b[i], 1.0);
S-34         a[i] = add2( a,      b, i, 1.0) + tmp;
S-35         a[i] = add3(&a[i], &b[i], 1.0);
S-36     }
S-37 }
S-38
S-39 int main(){
S-40     int i;
S-41     const int N=32;
S-42     double a[N], b[N];
S-43
S-44     for ( i=0; i<N; i++ ) {
S-45         a[i] = i; b[i] = N-i;
S-46     }
S-47
S-48     work(a, b, N );
S-49
S-50     for ( i=0; i<N; i++ ) {
S-51         printf("%d %f\n", i, a[i]);
S-52     }
S-53
S-54     return 0;
S-55 }

```

▲ C / C++ ▲

▼ Fortran ▼

1

Example SIMD.2.f90

```

S-1 program main
S-2     implicit none
S-3     integer, parameter :: N=32
S-4     integer :: i
S-5     double precision    :: a(N), b(N)
S-6     do i = 1,N
S-7         a(i) = i-1
S-8         b(i) = N-(i-1)

```

```

S-9      end do
S-10     call work(a, b, N )
S-11     do i = 1,N
S-12         print*, i,a(i)
S-13     end do
S-14 end program

S-15
S-16 function add1(a,b,fact) result(c)
S-17 !$omp declare simd(add1) uniform(fact)
S-18     implicit none
S-19     double precision :: a,b,fact, c
S-20     c = a + b + fact
S-21 end function

S-22
S-23 function add2(a,b,i, fact) result(c)
S-24 !$omp declare simd(add2) uniform(a,b,fact) linear(i:1)
S-25     implicit none
S-26     integer          :: i
S-27     double precision :: a(*),b(*),fact, c
S-28     c = a(i) + b(i) + fact
S-29 end function

S-30
S-31 subroutine work(a, b, n )
S-32     implicit none
S-33     double precision          :: a(n),b(n), tmp
S-34     integer                  :: n, i
S-35     double precision, external :: add1, add2
S-36
S-37     !$omp simd private(tmp)
S-38     do i = 1,n
S-39         tmp = add1(a(i), b(i), 1.0d0)
S-40         a(i) = add2(a,    b, i, 1.0d0) + tmp
S-41         a(i) = a(i) + b(i) + 1.0d0
S-42     end do
S-43 end subroutine

```

Fortran

- 1 A thread that encounters a SIMD construct executes a vectorized code of the iterations. Similar to
- 2 the concerns of a worksharing loop a loop vectorized with a SIMD construct must assure that
- 3 temporary and reduction variables are privatized and declared as reductions with clauses. The
- 4 example below illustrates the use of **private** and **reduction** clauses in a SIMD construct.

Example SIMD.3.c

```

S-1 double work( double *a, double *b, int n )
S-2 {
S-3     int i;
S-4     double tmp, sum;
S-5     sum = 0.0;
S-6     #pragma omp simd private(tmp) reduction(+:sum)
S-7     for (i = 0; i < n; i++) {
S-8         tmp = a[i] + b[i];
S-9         sum += tmp;
S-10    }
S-11    return sum;
S-12 }

```

Example SIMD.3.f90

```

S-1 subroutine work( a, b, n, sum )
S-2     implicit none
S-3     integer :: i, n
S-4     double precision :: a(n), b(n), sum, tmp
S-5
S-6     sum = 0.0d0
S-7     !$omp simd private(tmp) reduction(+:sum)
S-8     do i = 1,n
S-9         tmp = a(i) + b(i)
S-10        sum = sum + tmp
S-11    end do
S-12
S-13 end subroutine work

```

A **safelen(N)** clause in a **simd** construct assures the compiler that there are no loop-carried dependencies for vectors of size N or below. If the **safelen** clause is not specified, then the default safelen value is the number of loop iterations.

The **safelen(16)** clause in the example below guarantees that the vector code is safe for vectors up to and including size 16. In the loop, m can be 16 or greater, for correct code execution. If the value of m is less than 16, the behavior is undefined.

1

Example SIMD.4.c

```

S-1 void work( float *b, int n, int m )
S-2 {
S-3     int i;
S-4     #pragma omp simd safelen(16)
S-5     for (i = m; i < n; i++)
S-6         b[i] = b[i-m] - 1.0f;
S-7 }

```

2

Example SIMD.4.f90

```

S-1 subroutine work( b, n, m )
S-2     implicit none
S-3     real      :: b(n)
S-4     integer   :: i,n,m
S-5
S-6     !$omp simd safelen(16)
S-7     do i = m+1, n
S-8         b(i) = b(i-m) - 1.0
S-9     end do
S-10 end subroutine work

```

3

The following SIMD construct instructs the compiler to collapse the *i* and *j* loops into a single SIMD loop in which SIMD chunks are executed by threads of the team. Within the workshared loop chunks of a thread, the SIMD chunks are executed in the lanes of the vector units.

6

Example SIMD.5.c

```

S-1 void work( double **a, double **b, double **c, int n )
S-2 {
S-3     int i, j;
S-4     double tmp;
S-5     #pragma omp for simd collapse(2) private(tmp)
S-6     for (i = 0; i < n; i++) {
S-7         for (j = 0; j < n; j++) {
S-8             tmp = a[i][j] + b[i][j];
S-9             c[i][j] = tmp;
S-10        }
S-11    }
S-12 }

```

C / C++

Fortran

Example SIMD.5.f90

```

S-1  subroutine work( a, b, c,  n )
S-2      implicit none
S-3      integer :: i,j,n
S-4      double precision :: a(n,n), b(n,n), c(n,n), tmp
S-5
S-6      !$omp do simd collapse(2) private(tmp)
S-7      do j = 1,n
S-8          do i = 1,n
S-9              tmp = a(i,j) + b(i,j)
S-10             c(i,j) = tmp
S-11         end do
S-12     end do
S-13
S-14 end subroutine work

```

Fortran

5.2 inbranch and notinbranch Clauses

The following examples illustrate the use of the **declare simd** construct with the **inbranch** and **notinbranch** clauses. The **notinbranch** clause informs the compiler that the function *foo* is never called conditionally in the SIMD loop of the function *myaddint*. On the other hand, the **inbranch** clause for the function *goo* indicates that the function is always called conditionally in the SIMD loop inside the function *myaddfloat*.

1

Example SIMD.6.c

```

S-1  #pragma omp declare simd linear(p:1) notinbranch
S-2  int foo(int *p){
S-3      *p = *p + 10;
S-4      return *p;
S-5  }
S-6
S-7  int myaddint(int *a, int *b, int n)
S-8  {
S-9      #pragma omp simd
S-10     for (int i=0; i<n; i++){
S-11         a[i] = foo(&b[i]); /* foo is not called under a condition */
S-12     }
S-13     return a[n-1];
S-14 }
S-15
S-16 #pragma omp declare simd linear(p:1) inbranch
S-17 float goo(float *p){
S-18     *p = *p + 18.5f;
S-19     return *p;
S-20 }
S-21
S-22 int myaddfloat(float *x, float *y, int n)
S-23 {
S-24     #pragma omp simd
S-25     for (int i=0; i<n; i++){
S-26         x[i] = (x[i] > y[i]) ? goo(&y[i]) : y[i];
S-27         /* goo is called under the condition (or within a branch) */
S-28     }
S-29     return x[n-1];
S-30 }

```

2

Example SIMD.6.f90

```

S-1  function foo(p) result(r)
S-2  !$omp declare simd(foo) notinbranch
S-3      implicit none
S-4      integer :: p, r
S-5      p = p + 10
S-6      r = p
S-7  end function foo
S-8
S-9  function myaddint(a, b, n) result(r)

```

```

S-10      implicit none
S-11      integer :: a(*), b(*), n, r
S-12      integer :: i
S-13      integer, external :: foo
S-14
S-15      !$omp simd
S-16      do i=1, n
S-17          a(i) = foo(b(i))  ! foo is not called under a condition
S-18      end do
S-19      r = a(n)
S-20
S-21  end function myaddint
S-22
S-23  function goo(p) result(r)
S-24      !$omp declare simd(goo) inbranch
S-25      implicit none
S-26      real :: p, r
S-27      p = p + 18.5
S-28      r = p
S-29  end function goo
S-30
S-31  function myaddfloat(x, y, n) result(r)
S-32      implicit none
S-33      real :: x(*), y(*), r
S-34      integer :: n
S-35      integer :: i
S-36      real, external :: goo
S-37
S-38      !$omp simd
S-39      do i=1, n
S-40          if (x(i) > y(i)) then
S-41              x(i) = goo(y(i))
S-42              ! goo is called under the condition (or within a branch)
S-43          else
S-44              x(i) = y(i)
S-45          endif
S-46      end do
S-47
S-48      r = x(n)
S-49  end function myaddfloat

```

Fortran

- 1 In the code below, the function *fib()* is called in the main program and also recursively called in the
- 2 function *fib()* within an **if** condition. The compiler creates a masked vector version and a
- 3 non-masked vector version for the function *fib()* while retaining the original scalar version of the
- 4 *fib()* function.

1

Example SIMD.7.c

```

S-1  #include <stdio.h>
S-2  #include <stdlib.h>
S-3
S-4  #define N 45
S-5  int a[N], b[N], c[N];
S-6
S-7  #pragma omp declare simd inbranch
S-8  int fib( int n )
S-9  {
S-10     if (n <= 1)
S-11         return n;
S-12     else {
S-13         return fib(n-1) + fib(n-2);
S-14     }
S-15 }
S-16
S-17 int main(void)
S-18 {
S-19     int i;
S-20
S-21     #pragma omp simd
S-22     for (i=0; i < N; i++) b[i] = i;
S-23
S-24     #pragma omp simd
S-25     for (i=0; i < N; i++) {
S-26         a[i] = fib(b[i]);
S-27     }
S-28     printf("Done a[%d] = %d\n", N-1, a[N-1]);
S-29     return 0;
S-30 }

```

2

Example SIMD.7.f90

```

S-1  program fibonacci
S-2      implicit none
S-3      integer,parameter :: N=45
S-4      integer           :: a(0:N-1), b(0:N-1)
S-5      integer           :: i
S-6      integer, external :: fib
S-7
S-8      !$omp simd
S-9      do i = 0,N-1

```

```

S-10      b(i) = i
S-11      end do
S-12
S-13      !$omp simd
S-14      do i=0,N-1
S-15          a(i) = fib(b(i))
S-16      end do
S-17
S-18      write(*,*) "Done a(", N-1, ") = ", a(N-1)
S-19                      ! 44  701408733
S-20      end program
S-21
S-22      recursive function fib(n) result(r)
S-23      !$omp declare simd(fib) inbranch
S-24          implicit none
S-25          integer :: n, r
S-26
S-27          if (n <= 1) then
S-28              r = n
S-29          else
S-30              r = fib(n-1) + fib(n-2)
S-31          endif
S-32
S-33      end function fib

```

Fortran

1 5.3 Loop-Carried Lexical Forward Dependence

2 The following example tests the restriction on an SIMD loop with the loop-carried lexical
3 forward-dependence. This dependence must be preserved for the correct execution of SIMD loops.

4 A loop can be vectorized even though the iterations are not completely independent when it has
5 loop-carried dependences that are forward lexical dependences, indicated in the code below by the
6 read of $A[j+1]$ and the write to $A[j]$ in C/C++ code (or $A(j+1)$ and $A(j)$ in Fortran). That is, the
7 read of $A[j+1]$ (or $A(j+1)$ in Fortran) before the write to $A[j]$ (or $A(j)$ in Fortran) ordering must be
8 preserved for each iteration in j for valid SIMD code generation.

9 This test assures that the compiler preserves the loop carried lexical forward-dependence for
10 generating a correct SIMD code.

1

Example SIMD.8.c

```

S-1  #include <stdio.h>
S-2  #include <math.h>
S-3
S-4  int    P[1000];
S-5  float  A[1000];
S-6
S-7  float do_work(float *arr)
S-8  {
S-9      float pri;
S-10     int i;
S-11     #pragma omp simd lastprivate(pri)
S-12     for (i = 0; i < 999; ++i) {
S-13         int j = P[i];
S-14
S-15         pri = 0.5f;
S-16         if (j % 2 == 0) {
S-17             pri = A[j+1] + arr[i];
S-18         }
S-19         A[j] = pri * 1.5f;
S-20         pri = pri + A[j];
S-21     }
S-22     return pri;
S-23 }
S-24
S-25 int main(void)
S-26 {
S-27     float pri, arr[1000];
S-28     int i;
S-29
S-30     for (i = 0; i < 1000; ++i) {
S-31         P[i]    = i;
S-32         A[i]    = i * 1.5f;
S-33         arr[i]  = i * 1.8f;
S-34     }
S-35     pri = do_work(&arr[0]);
S-36     if (pri == 8237.25) {
S-37         printf("passed: result pri = %7.2f (8237.25) \n", pri);
S-38     }
S-39     else {
S-40         printf("failed: result pri = %7.2f (8237.25) \n", pri);
S-41     }
S-42     return 0;
S-43 }

```

1

Example SIMD.8.f90

```

S-1  module work
S-2
S-3  integer :: P(1000)
S-4  real    :: A(1000)
S-5
S-6  contains
S-7  function do_work(arr) result(pri)
S-8      implicit none
S-9      real, dimension(*) :: arr
S-10
S-11      real :: pri
S-12      integer :: i, j
S-13
S-14      !$omp simd private(j) lastprivate(pri)
S-15      do i = 1, 999
S-16          j = P(i)
S-17
S-18          pri = 0.5
S-19          if (mod(j-1, 2) == 0) then
S-20              pri = A(j+1) + arr(i)
S-21          endif
S-22          A(j) = pri * 1.5
S-23          pri = pri + A(j)
S-24      end do
S-25
S-26  end function do_work
S-27
S-28  end module work
S-29
S-30  program simd_8f
S-31      use work
S-32      implicit none
S-33      real :: pri, arr(1000)
S-34      integer :: i
S-35
S-36      do i = 1, 1000
S-37          P(i) = i
S-38          A(i) = (i-1) * 1.5
S-39          arr(i) = (i-1) * 1.8
S-40      end do
S-41      pri = do_work(arr)
S-42      if (pri == 8237.25) then

```



```
S-43      print 2, "passed", pri
S-44      else
S-45          print 2, "failed", pri
S-46      endif
S-47      2 format(a, ": result pri = ", f7.2, " (8237.25)")
S-48
S-49      end program
```

Fortran

Synchronization

The **barrier** construct is a stand-alone directive that requires all threads of a team (within a contention group) to execute the barrier and complete execution of all tasks within the region, before continuing past the barrier.

The **critical** construct is a directive that contains a structured block. The construct allows only a single thread at a time to execute the structured block (region). Multiple critical regions may exist in a parallel region, and may act cooperatively (only one thread at a time in all **critical** regions), or separately (only one thread at a time in each **critical** regions when a unique name is supplied on each **critical** construct). An optional (lock) **hint** clause may be specified on a named **critical** construct to provide the OpenMP runtime guidance in selection a locking mechanism.

On a finer scale the **atomic** construct allows only a single thread at a time to have atomic access to a storage location involving a single read, write, update or capture statement, and a limited number of combinations when specifying the **capture atomic-clause** clause. The *atomic-clause* clause is required for some expression statements, but are not required for **update** statements. Please see the details in the *atomic Construct* subsection of the *Directives* chapter in the OpenMP Specifications document.

The **ordered** construct either specifies a structured block in a loop, simd, or loop SIMD region that will be executed in the order of the loop iterations. The ordered construct sequentializes and orders the execution of ordered regions while allowing code outside the region to run in parallel.

Since OpenMP 4.5 the **ordered** construct can also be a stand-alone directive that specifies cross-iteration dependences in a doacross loop nest. The **depend** clause uses a **sink dependence-type**, along with a iteration vector argument (*vec*) to indicate the iteration that satisfies the dependence. The **depend** clause with a **source dependence-type** specifies dependence satisfaction.

The **flush** directive is a stand-alone construct that forces a thread's temporal local storage (view) of a variable to memory where a consistent view of the variable storage can be accesses. When the construct is used without a variable list, all the locally thread-visible data as defined by the base language are flushed. A construct with a list applies the flush operation only to the items in the list.

1 The **flush** construct also effectively insures that no memory (load or store) operation for the
2 variable set (list items, or default set) may be reordered across the **flush** directive.

3 General-purpose routines provide mutual exclusion semantics through locks, represented by lock
4 variables. The semantics allows a task to *set*, and hence *own* a lock, until it is *unset* by the task that
5 set it. A *nestable* lock can be set multiple times by a task, and is used when in code requires nested
6 control of locks. A *simple lock* can only be set once by the owning task. There are specific calls for
7 the two types of locks, and the variable of a specific lock type cannot be used by the other lock type.

8 Any explicit task will observe the synchronization prescribed in a **barrier** construct and an
9 implied barrier. Also, additional synchronizations are available for tasks. All children of a task will
10 wait at a **taskwait** (for their siblings to complete). A **taskgroup** construct creates a region in
11 which the current task is suspended at the end of the region until all sibling tasks, and their
12 descendants, have completed. Scheduling constraints on task execution can be prescribed by the
13 **depend** clause to enforce dependence on previously generated tasks. More details on controlling
14 task executions can be found in the *Tasking* Chapter in the OpenMP Specifications document.

1 6.1 The critical Construct

2 The following example includes several **critical** constructs. The example illustrates a queuing
3 model in which a task is dequeued and worked on. To guard against multiple threads dequeuing the
4 same task, the dequeuing operation must be in a **critical** region. Because the two queues in this
5 example are independent, they are protected by **critical** constructs with different names, *xaxis*
6 and *yaxis*.

▼ C / C++ ▼

7 *Example critical.1.c*

```
S-1 int dequeue(float *a);  
S-2 void work(int i, float *a);  
S-3  
S-4 void critical_example(float *x, float *y)  
S-5 {  
S-6     int ix_next, iy_next;  
S-7  
S-8     #pragma omp parallel shared(x, y) private(ix_next, iy_next)  
S-9     {  
S-10         #pragma omp critical (xaxis)  
S-11         ix_next = dequeue(x);  
S-12         work(ix_next, x);  
S-13  
S-14         #pragma omp critical (yaxis)  
S-15         iy_next = dequeue(y);  
S-16         work(iy_next, y);  
S-17     }  
S-18 }  
S-19 }
```

▲ C / C++ ▲

▼ Fortran ▼

8 *Example critical.1.f*

```
S-1 SUBROUTINE CRITICAL_EXAMPLE(X, Y)  
S-2  
S-3 REAL X(*), Y(*)  
S-4 INTEGER IX_NEXT, IY_NEXT  
S-5  
S-6 !$OMP PARALLEL SHARED(X, Y) PRIVATE(IX_NEXT, IY_NEXT)  
S-7  
S-8 !$OMP CRITICAL(XAXIS)  
S-9 CALL DEQUEUE(IX_NEXT, X)  
S-10 !$OMP END CRITICAL(XAXIS)  
S-11 CALL WORK(IX_NEXT, X)
```

```

S-12
S-13  !$OMP CRITICAL(YAXIS)
S-14      CALL DEQUEUE(IY_NEXT, Y)
S-15  !$OMP END CRITICAL(YAXIS)
S-16      CALL WORK(IY_NEXT, Y)
S-17
S-18  !$OMP END PARALLEL
S-19
S-20      END SUBROUTINE CRITICAL_EXAMPLE

```

Fortran

1 The following example extends the previous example by adding the **hint** clause to the **critical**
 2 constructs.

C / C++

3 *Example critical.2.c*

```

S-1  #include <omp.h>
S-2
S-3  int dequeue(float *a);
S-4  void work(int i, float *a);
S-5
S-6  void critical_example(float *x, float *y)
S-7  {
S-8      int ix_next, iy_next;
S-9
S-10     #pragma omp parallel shared(x, y) private(ix_next, iy_next)
S-11     {
S-12         #pragma omp critical (xaxis) hint(omp_lock_hint_contended)
S-13         ix_next = dequeue(x);
S-14         work(ix_next, x);
S-15
S-16         #pragma omp critical (yaxis) hint(omp_lock_hint_contended)
S-17         iy_next = dequeue(y);
S-18         work(iy_next, y);
S-19     }
S-20
S-21 }

```

C / C++

1

Example critical.2.f

```

S-1      SUBROUTINE CRITICAL_EXAMPLE(X, Y)
S-2          USE OMP_LIB          ! or INCLUDE "omp_lib.h"
S-3
S-4          REAL X(*), Y(*)
S-5          INTEGER IX_NEXT, IY_NEXT
S-6
S-7      !$OMP PARALLEL SHARED(X, Y) PRIVATE(IX_NEXT, IY_NEXT)
S-8
S-9      !$OMP CRITICAL(XAXIS) HINT(OMP_LOCK_HINT_CONTENTENDED)
S-10         CALL DEQUEUE(IX_NEXT, X)
S-11      !$OMP END CRITICAL(XAXIS)
S-12         CALL WORK(IX_NEXT, X)
S-13
S-14      !$OMP CRITICAL(YAXIS) HINT(OMP_LOCK_HINT_CONTENTENDED)
S-15         CALL DEQUEUE(IY_NEXT, Y)
S-16      !$OMP END CRITICAL(YAXIS)
S-17         CALL WORK(IY_NEXT, Y)
S-18
S-19      !$OMP END PARALLEL
S-20
S-21      END SUBROUTINE CRITICAL_EXAMPLE

```

1 6.2 Worksharing Constructs Inside a **critical** 2 Construct

3 The following example demonstrates using a worksharing construct inside a **critical** construct.
4 This example is conforming because the worksharing **single** region is not closely nested inside
5 the **critical** region. A single thread executes the one and only section in the **sections**
6 region, and executes the **critical** region. The same thread encounters the nested **parallel**
7 region, creates a new team of threads, and becomes the master of the new team. One of the threads
8 in the new team enters the **single** region and increments **i** by 1. At the end of this example **i** is
9 equal to 2.

▼ C / C++ ▼

10 *Example worksharing_critical.1.c*

```
S-1 void critical_work()  
S-2 {  
S-3     int i = 1;  
S-4     #pragma omp parallel sections  
S-5     {  
S-6         #pragma omp section  
S-7         {  
S-8             #pragma omp critical (name)  
S-9             {  
S-10                #pragma omp parallel  
S-11                {  
S-12                    #pragma omp single  
S-13                    {  
S-14                        i++;  
S-15                    }  
S-16                }  
S-17            }  
S-18        }  
S-19    }  
S-20 }
```

▲ C / C++ ▲

1

Example worksharing_critical.1.f

```
S-1      SUBROUTINE CRITICAL_WORK()  
S-2  
S-3          INTEGER I  
S-4          I = 1  
S-5  
S-6      !$OMP  PARALLEL SECTIONS  
S-7      !$OMP  SECTION  
S-8      !$OMP  CRITICAL (NAME)  
S-9      !$OMP  PARALLEL  
S-10     !$OMP  SINGLE  
S-11         I = I + 1  
S-12     !$OMP  END SINGLE  
S-13     !$OMP  END PARALLEL  
S-14     !$OMP  END CRITICAL (NAME)  
S-15     !$OMP  END PARALLEL SECTIONS  
S-16     END SUBROUTINE CRITICAL_WORK
```


1 6.3 Binding of barrier Regions

2 The binding rules call for a **barrier** region to bind to the closest enclosing **parallel** region.

3 In the following example, the call from the main program to *sub2* is conforming because the
4 **barrier** region (in *sub3*) binds to the **parallel** region in *sub2*. The call from the main
5 program to *sub1* is conforming because the **barrier** region binds to the **parallel** region in
6 subroutine *sub2*.

7 The call from the main program to *sub3* is conforming because the **barrier** region binds to the
8 implicit inactive **parallel** region enclosing the sequential part. Also note that the **barrier**
9 region in *sub3* when called from *sub2* only synchronizes the team of threads in the enclosing
10 **parallel** region and not all the threads created in *sub1*.

▼ C / C++ ▼

11 *Example barrier_regions.1.c*

```
S-1 void work(int n) {}
S-2
S-3 void sub3(int n)
S-4 {
S-5     work(n);
S-6     #pragma omp barrier
S-7     work(n);
S-8 }
S-9
S-10 void sub2(int k)
S-11 {
S-12     #pragma omp parallel shared(k)
S-13         sub3(k);
S-14 }
S-15
S-16 void sub1(int n)
S-17 {
S-18     int i;
S-19     #pragma omp parallel private(i) shared(n)
S-20     {
S-21         #pragma omp for
S-22         for (i=0; i<n; i++)
S-23             sub2(i);
S-24     }
S-25 }
S-26
S-27 int main()
S-28 {
S-29     sub1(2);
S-30     sub2(2);
```

```

S-31     sub3(2);
S-32     return 0;
S-33 }

```

C / C++

Fortran

1

Example barrier_regions.1.f

```

S-1      SUBROUTINE WORK(N)
S-2      INTEGER N
S-3      END SUBROUTINE WORK
S-4
S-5      SUBROUTINE SUB3(N)
S-6      INTEGER N
S-7      CALL WORK(N)
S-8      !$OMP BARRIER
S-9      CALL WORK(N)
S-10     END SUBROUTINE SUB3
S-11
S-12     SUBROUTINE SUB2(K)
S-13     INTEGER K
S-14     !$OMP PARALLEL SHARED(K)
S-15         CALL SUB3(K)
S-16     !$OMP END PARALLEL
S-17     END SUBROUTINE SUB2
S-18
S-19
S-20     SUBROUTINE SUB1(N)
S-21     INTEGER N
S-22     INTEGER I
S-23     !$OMP PARALLEL PRIVATE(I) SHARED(N)
S-24     !$OMP DO
S-25         DO I = 1, N
S-26             CALL SUB2(I)
S-27         END DO
S-28     !$OMP END PARALLEL
S-29     END SUBROUTINE SUB1
S-30
S-31     PROGRAM EXAMPLE
S-32         CALL SUB1(2)
S-33         CALL SUB2(2)
S-34         CALL SUB3(2)
S-35     END PROGRAM EXAMPLE

```

Fortran

1 6.4 The `atomic` Construct

2 The following example avoids race conditions (simultaneous updates of an element of x by multiple
3 threads) by using the **`atomic`** construct .

4 The advantage of using the **`atomic`** construct in this example is that it allows updates of two
5 different elements of x to occur in parallel. If a **`critical`** construct were used instead, then all
6 updates to elements of x would be executed serially (though not in any guaranteed order).

7 Note that the **`atomic`** directive applies only to the statement immediately following it. As a result,
8 elements of y are not updated atomically in this example.

▼ C / C++ ▼

9 *Example atomic.1.c*

```
S-1 float work1(int i)
S-2 {
S-3     return 1.0 * i;
S-4 }
S-5
S-6 float work2(int i)
S-7 {
S-8     return 2.0 * i;
S-9 }
S-10
S-11 void atomic_example(float *x, float *y, int *index, int n)
S-12 {
S-13     int i;
S-14
S-15     #pragma omp parallel for shared(x, y, index, n)
S-16     for (i=0; i<n; i++) {
S-17         #pragma omp atomic update
S-18         x[index[i]] += work1(i);
S-19         y[i] += work2(i);
S-20     }
S-21 }
S-22
S-23 int main()
S-24 {
S-25     float x[1000];
S-26     float y[10000];
S-27     int index[10000];
S-28     int i;
S-29
S-30     for (i = 0; i < 10000; i++) {
S-31         index[i] = i % 1000;
S-32         y[i]=0.0;
```

```

S-33     }
S-34     for (i = 0; i < 1000; i++)
S-35         x[i] = 0.0;
S-36     atomic_example(x, y, index, 10000);
S-37     return 0;
S-38 }

```

▲ C / C++ ▲

▼ Fortran ▼

1

Example atomic.f

```

S-1      REAL FUNCTION WORK1(I)
S-2      INTEGER I
S-3      WORK1 = 1.0 * I
S-4      RETURN
S-5      END FUNCTION WORK1
S-6
S-7      REAL FUNCTION WORK2(I)
S-8      INTEGER I
S-9      WORK2 = 2.0 * I
S-10     RETURN
S-11     END FUNCTION WORK2
S-12
S-13     SUBROUTINE SUB(X, Y, INDEX, N)
S-14     REAL X(*), Y(*)
S-15     INTEGER INDEX(*), N
S-16
S-17     INTEGER I
S-18
S-19     !$OMP PARALLEL DO SHARED(X, Y, INDEX, N)
S-20     DO I=1,N
S-21     !$OMP ATOMIC UPDATE
S-22         X(INDEX(I)) = X(INDEX(I)) + WORK1(I)
S-23         Y(I) = Y(I) + WORK2(I)
S-24     ENDDO
S-25
S-26     END SUBROUTINE SUB
S-27
S-28     PROGRAM ATOMIC_EXAMPLE
S-29     REAL X(1000), Y(10000)
S-30     INTEGER INDEX(10000)
S-31     INTEGER I
S-32
S-33     DO I=1,10000
S-34         INDEX(I) = MOD(I, 1000) + 1
S-35         Y(I) = 0.0
S-36     ENDDO

```

```

S-37
S-38      DO I = 1,1000
S-39          X(I) = 0.0
S-40      ENDDO
S-41
S-42      CALL SUB(X, Y, INDEX, 10000)
S-43
S-44      END PROGRAM ATOMIC_EXAMPLE

```

Fortran

1 The following example illustrates the **read** and **write** clauses for the **atomic** directive. These
2 clauses ensure that the given variable is read or written, respectively, as a whole. Otherwise, some
3 other thread might read or write part of the variable while the current thread was reading or writing
4 another part of the variable. Note that most hardware provides atomic reads and writes for some set
5 of properly aligned variables of specific sizes, but not necessarily for all the variable types
6 supported by the OpenMP API.

C / C++

7 *Example atomic.2.c*

```

S-1  int atomic_read(const int *p)
S-2  {
S-3      int value;
S-4      /* Guarantee that the entire value of *p is read atomically. No part of
S-5       * *p can change during the read operation.
S-6       */
S-7      #pragma omp atomic read
S-8          value = *p;
S-9      return value;
S-10 }
S-11
S-12 void atomic_write(int *p, int value)
S-13 {
S-14     /* Guarantee that value is stored atomically into *p. No part of *p can
S-15     change
S-16     * until after the entire write operation is completed.
S-17     */
S-18     #pragma omp atomic write
S-19         *p = value;
S-20 }

```

C / C++

Example atomic.2.f

```

S-1      function atomic_read(p)
S-2      integer :: atomic_read
S-3      integer, intent(in) :: p
S-4      ! Guarantee that the entire value of p is read atomically. No part of
S-5      ! p can change during the read operation.
S-6
S-7      !$omp atomic read
S-8      atomic_read = p
S-9      return
S-10     end function atomic_read
S-11
S-12     subroutine atomic_write(p, value)
S-13     integer, intent(out) :: p
S-14     integer, intent(in) :: value
S-15     ! Guarantee that value is stored atomically into p. No part of p can change
S-16     ! until after the entire write operation is completed.
S-17     !$omp atomic write
S-18     p = value
S-19     end subroutine atomic_write

```

The following example illustrates the **capture** clause for the **atomic** directive. In this case the value of a variable is captured, and then the variable is incremented. These operations occur atomically. This particular example could be implemented using the fetch-and-add instruction available on many kinds of hardware. The example also shows a way to implement a spin lock using the **capture** and **read** clauses.

Example atomic.3.c

```

S-1      int fetch_and_add(int *p)
S-2      {
S-3      /* Atomically read the value of *p and then increment it. The previous value
S-4      is
S-5      * returned. This can be used to implement a simple lock as shown below.
S-6      */
S-7      int old;
S-8      #pragma omp atomic capture
S-9      { old = *p; (*p)++; }
S-10     return old;
S-11     }
S-12
S-13     /*

```

```

S-14      * Use fetch_and_add to implement a lock
S-15      */
S-16      struct locktype {
S-17          int ticketnumber;
S-18          int turn;
S-19      };
S-20      void do_locked_work(struct locktype *lock)
S-21      {
S-22          int atomic_read(const int *p);
S-23          void work();
S-24
S-25          // Obtain the lock
S-26          int myturn = fetch_and_add(&lock->ticketnumber);
S-27          while (atomic_read(&lock->turn) != myturn)
S-28              ;
S-29          // Do some work. The flush is needed to ensure visibility of
S-30          // variables not involved in atomic directives
S-31
S-32          #pragma omp flush
S-33          work();
S-34          #pragma omp flush
S-35          // Release the lock
S-36          fetch_and_add(&lock->turn);
S-37      }

```

C / C++

Fortran

1

Example atomic.3.f

```

S-1          function fetch_and_add(p)
S-2          integer :: fetch_and_add
S-3          integer, intent(inout) :: p
S-4
S-5          ! Atomically read the value of p and then increment it. The previous value is
S-6          ! returned. This can be used to implement a simple lock as shown below.
S-7          !$omp atomic capture
S-8              fetch_and_add = p
S-9              p = p + 1
S-10         !$omp end atomic
S-11         end function fetch_and_add
S-12         module m
S-13         interface
S-14             function fetch_and_add(p)
S-15                 integer :: fetch_and_add
S-16                 integer, intent(inout) :: p
S-17             end function
S-18             function atomic_read(p)

```

```

S-19         integer :: atomic_read
S-20         integer, intent(in) :: p
S-21     end function
S-22 end interface
S-23 type locktype
S-24     integer ticketnumber
S-25     integer turn
S-26 end type
S-27 contains
S-28     subroutine do_locked_work(lock)
S-29     type(locktype), intent(inout) :: lock
S-30     integer myturn
S-31     integer junk
S-32 ! obtain the lock
S-33     myturn = fetch_and_add(lock%ticketnumber)
S-34     do while (atomic_read(lock%turn) .ne. myturn)
S-35         continue
S-36     enddo
S-37 ! Do some work. The flush is needed to ensure visibility of variables
S-38 ! not involved in atomic directives
S-39 !$omp flush
S-40     call work
S-41 !$omp flush
S-42 ! Release the lock
S-43     junk = fetch_and_add(lock%turn)
S-44 end subroutine
S-45 end module

```

▲ ————— Fortran ————— ▲

1 6.5 Restrictions on the `atomic` Construct

2 The following non-conforming examples illustrate the restrictions on the `atomic` construct.

▼ C / C++ ▼

3 *Example `atomic_restrict.1.c`*

```
S-1 void atomic_wrong ()
S-2 {
S-3     union {int n; float x;} u;
S-4
S-5     #pragma omp parallel
S-6     {
S-7         #pragma omp atomic update
S-8             u.n++;
S-9
S-10        #pragma omp atomic update
S-11            u.x += 1.0;
S-12
S-13        /* Incorrect because the atomic constructs reference the same location
S-14           through incompatible types */
S-15        }
S-16    }
```

▲ C / C++ ▲

▼ Fortran ▼

4 *Example `atomic_restrict.1.f`*

```
S-1      SUBROUTINE ATOMIC_WRONG()
S-2          INTEGER:: I
S-3          REAL:: R
S-4          EQUIVALENCE (I,R)
S-5
S-6      !$OMP    PARALLEL
S-7      !$OMP    ATOMIC UPDATE
S-8          I = I + 1
S-9      !$OMP    ATOMIC UPDATE
S-10         R = R + 1.0
S-11      ! incorrect because I and R reference the same location
S-12      ! but have different types
S-13      !$OMP    END PARALLEL
S-14      END SUBROUTINE ATOMIC_WRONG
```

▲ Fortran ▲

Example atomic_restrict.2.c

```

S-1 void atomic_wrong2 ()
S-2 {
S-3     int x;
S-4     int *i;
S-5     float *r;
S-6
S-7     i = &x;
S-8     r = (float *)&x;
S-9
S-10    #pragma omp parallel
S-11    {
S-12    #pragma omp atomic update
S-13        *i += 1;
S-14
S-15    #pragma omp atomic update
S-16        *r += 1.0;
S-17
S-18    /* Incorrect because the atomic constructs reference the same location
S-19       through incompatible types */
S-20
S-21    }
S-22 }

```

The following example is non-conforming because **I** and **R** reference the same location but have different types.

Example atomic_restrict.2.f

```

S-1     SUBROUTINE SUB()
S-2     COMMON /BLK/ R
S-3     REAL R
S-4
S-5     !$OMP ATOMIC UPDATE
S-6         R = R + 1.0
S-7     END SUBROUTINE SUB
S-8
S-9     SUBROUTINE ATOMIC_WRONG2()
S-10    COMMON /BLK/ I
S-11    INTEGER I
S-12
S-13    !$OMP PARALLEL
S-14

```


1 6.6 The flush Construct without a List

2 The following example distinguishes the shared variables affected by a **flush** construct with no
3 list from the shared objects that are not affected:

C / C++

4 *Example flush_nolist.1.c*

```
S-1 int x, *p = &x;
S-2
S-3 void f1(int *q)
S-4 {
S-5     *q = 1;
S-6     #pragma omp flush
S-7     /* x, p, and *q are flushed */
S-8     /* because they are shared and accessible */
S-9     /* q is not flushed because it is not shared. */
S-10 }
S-11
S-12 void f2(int *q)
S-13 {
S-14     #pragma omp barrier
S-15     *q = 2;
S-16     #pragma omp barrier
S-17
S-18     /* a barrier implies a flush */
S-19     /* x, p, and *q are flushed */
S-20     /* because they are shared and accessible */
S-21     /* q is not flushed because it is not shared. */
S-22 }
S-23
S-24 int g(int n)
S-25 {
S-26     int i = 1, j, sum = 0;
S-27     *p = 1;
S-28     #pragma omp parallel reduction(+: sum) num_threads(10)
S-29     {
S-30         f1(&j);
S-31
S-32         /* i, n and sum were not flushed */
S-33         /* because they were not accessible in f1 */
S-34         /* j was flushed because it was accessible */
S-35         sum += j;
S-36
S-37         f2(&j);
S-38
S-39         /* i, n, and sum were not flushed */
```

```

S-40     /* because they were not accessible in f2 */
S-41     /* j was flushed because it was accessible */
S-42     sum += i + j + *p + n;
S-43     }
S-44     return sum;
S-45 }
S-46
S-47 int main()
S-48 {
S-49     int result = g(7);
S-50     return result;
S-51 }

```



1

Example flush_nolist.f

```

S-1      SUBROUTINE F1(Q)
S-2      COMMON /DATA/ X, P
S-3      INTEGER, TARGET :: X
S-4      INTEGER, POINTER :: P
S-5      INTEGER Q
S-6
S-7      Q = 1
S-8      !$OMP FLUSH
S-9      ! X, P and Q are flushed
S-10     ! because they are shared and accessible
S-11     END SUBROUTINE F1
S-12
S-13     SUBROUTINE F2(Q)
S-14     COMMON /DATA/ X, P
S-15     INTEGER, TARGET :: X
S-16     INTEGER, POINTER :: P
S-17     INTEGER Q
S-18
S-19     !$OMP BARRIER
S-20     Q = 2
S-21     !$OMP BARRIER
S-22     ! a barrier implies a flush
S-23     ! X, P and Q are flushed
S-24     ! because they are shared and accessible
S-25     END SUBROUTINE F2
S-26
S-27     INTEGER FUNCTION G(N)
S-28     COMMON /DATA/ X, P
S-29     INTEGER, TARGET :: X
S-30     INTEGER, POINTER :: P

```

```

S-31      INTEGER N
S-32      INTEGER I, J, SUM
S-33
S-34      I = 1
S-35      SUM = 0
S-36      P = 1
S-37      !$OMP PARALLEL REDUCTION(+: SUM) NUM_THREADS(10)
S-38          CALL F1(J)
S-39              ! I, N and SUM were not flushed
S-40              !   because they were not accessible in F1
S-41              ! J was flushed because it was accessible
S-42          SUM = SUM + J
S-43
S-44          CALL F2(J)
S-45              ! I, N, and SUM were not flushed
S-46              !   because they were not accessible in f2
S-47              ! J was flushed because it was accessible
S-48          SUM = SUM + I + J + P + N
S-49      !$OMP END PARALLEL
S-50
S-51      G = SUM
S-52      END FUNCTION G
S-53
S-54      PROGRAM FLUSH_NOLIST
S-55          COMMON /DATA/ X, P
S-56          INTEGER, TARGET :: X
S-57          INTEGER, POINTER :: P
S-58          INTEGER RESULT, G
S-59
S-60          P => X
S-61          RESULT = G(7)
S-62          PRINT *, RESULT
S-63      END PROGRAM FLUSH_NOLIST

```

Fortran

1 6.7 The ordered Clause and the ordered Construct

2 Ordered constructs are useful for sequentially ordering the output from work that is done in
3 parallel. The following program prints out the indices in sequential order:

▼ C / C++ ▼

4 *Example ordered.1.c*

```
S-1 #include <stdio.h>
S-2
S-3 void work(int k)
S-4 {
S-5     #pragma omp ordered
S-6     printf(" %d\n", k);
S-7 }
S-8
S-9 void ordered_example(int lb, int ub, int stride)
S-10 {
S-11     int i;
S-12
S-13     #pragma omp parallel for ordered schedule(dynamic)
S-14     for (i=lb; i<ub; i+=stride)
S-15         work(i);
S-16 }
S-17
S-18 int main()
S-19 {
S-20     ordered_example(0, 100, 5);
S-21     return 0;
S-22 }
```

▲ C / C++ ▲

▼ Fortran ▼

5 *Example ordered.1.f*

```
S-1         SUBROUTINE WORK(K)
S-2             INTEGER k
S-3
S-4         !$OMP ORDERED
S-5             WRITE(*,*) K
S-6         !$OMP END ORDERED
S-7
S-8         END SUBROUTINE WORK
S-9
S-10        SUBROUTINE SUB(LB, UB, STRIDE)
S-11            INTEGER LB, UB, STRIDE
S-12            INTEGER I
```

```

S-13
S-14 !$OMP PARALLEL DO ORDERED SCHEDULE(DYNAMIC)
S-15     DO I=LB,UB,STRIDE
S-16         CALL WORK(I)
S-17     END DO
S-18 !$OMP END PARALLEL DO
S-19
S-20     END SUBROUTINE SUB
S-21
S-22     PROGRAM ORDERED_EXAMPLE
S-23         CALL SUB(1,100,5)
S-24     END PROGRAM ORDERED_EXAMPLE

```

Fortran

It is possible to have multiple **ordered** constructs within a loop region with the **ordered** clause specified. The first example is non-conforming because all iterations execute two **ordered** regions. An iteration of a loop must not execute more than one **ordered** region:

C / C++

Example ordered.2.c

```

S-1 void work(int i) {}
S-2
S-3 void ordered_wrong(int n)
S-4 {
S-5     int i;
S-6     #pragma omp for ordered
S-7     for (i=0; i<n; i++) {
S-8 /* incorrect because an iteration may not execute more than one
S-9     ordered region */
S-10    #pragma omp ordered
S-11        work(i);
S-12    #pragma omp ordered
S-13        work(i+1);
S-14    }
S-15 }

```

C / C++

Example ordered.2.f

```

S-1      SUBROUTINE WORK(I)
S-2      INTEGER I
S-3      END SUBROUTINE WORK
S-4
S-5      SUBROUTINE ORDERED_WRONG(N)
S-6      INTEGER N
S-7
S-8      INTEGER I
S-9      !$OMP DO ORDERED
S-10     DO I = 1, N
S-11     ! incorrect because an iteration may not execute more than one
S-12     ! ordered region
S-13     !$OMP ORDERED
S-14         CALL WORK(I)
S-15     !$OMP END ORDERED
S-16
S-17     !$OMP ORDERED
S-18         CALL WORK(I+1)
S-19     !$OMP END ORDERED
S-20     END DO
S-21     END SUBROUTINE ORDERED_WRONG

```

The following is a conforming example with more than one **ordered** construct. Each iteration will execute only one **ordered** region:

Example ordered.3.c

```

S-1      void work(int i) {}
S-2      void ordered_good(int n)
S-3      {
S-4          int i;
S-5          #pragma omp for ordered
S-6          for (i=0; i<n; i++) {
S-7              if (i <= 10) {
S-8                  #pragma omp ordered
S-9                  work(i);
S-10             }
S-11             if (i > 10) {
S-12                 #pragma omp ordered
S-13                 work(i+1);
S-14             }

```

S-15 }
S-16 }

C / C++

Fortran

1

Example ordered.3.f

```
S-1          SUBROUTINE ORDERED_GOOD (N)
S-2          INTEGER N
S-3
S-4          !$OMP DO ORDERED
S-5              DO I = 1,N
S-6                  IF (I <= 10) THEN
S-7                      !$OMP ORDERED
S-8                          CALL WORK(I)
S-9                      END ORDERED
S-10                 ENDIF
S-11
S-12                 IF (I > 10) THEN
S-13                     !$OMP ORDERED
S-14                         CALL WORK(I+1)
S-15                     END ORDERED
S-16                 ENDIF
S-17             ENDDO
S-18         END SUBROUTINE ORDERED_GOOD
```

Fortran

1 6.8 Doacross Loop Nest

2 An **ordered** clause can be used on a loop construct with an integer parameter argument to define
3 the number of associated loops within a *doacross loop nest* where cross-iteration dependences
4 exist. A **depend** clause on an **ordered** construct within an ordered loop describes the
5 dependences of the *doacross* loops.

6 In the code below, the **depend(sink:i-1)** clause defines an *i-1* to *i* cross-iteration dependence
7 that specifies a wait point for the completion of computation from iteration *i-1* before proceeding to
8 the subsequent statements. The **depend(source)** clause indicates the completion of
9 computation from the current iteration (*i*) to satisfy the cross-iteration dependence that arises from
10 the iteration. For this example the same sequential ordering could have been achieved with an
11 **ordered** clause without a parameter, on the loop directive, and a single **ordered** directive
12 without the **depend** clause specified for the statement executing the *bar* function.

▼ C / C++ ▼

13 *Example doacross.1.c*

```
S-1 float foo(int i);  
S-2 float bar(float a, float b);  
S-3 float baz(float b);  
S-4  
S-5 void work( int N, float *A, float *B, float *C )  
S-6 {  
S-7     int i;  
S-8  
S-9     #pragma omp for ordered(1)  
S-10    for (i=1; i<N; i++)  
S-11    {  
S-12        A[i] = foo(i);  
S-13  
S-14        #pragma omp ordered depend(sink: i-1)  
S-15        B[i] = bar(A[i], B[i-1]);  
S-16        #pragma omp ordered depend(source)  
S-17  
S-18        C[i] = baz(B[i]);  
S-19    }  
S-20 }
```

▲ C / C++ ▲

Example *doacross.1.f90*

```

S-1  subroutine work( N, A, B, C )
S-2      integer :: N, i
S-3      real, dimension(N) :: A, B, C
S-4      real, external :: foo, bar, baz
S-5
S-6      !$omp do ordered(1)
S-7      do i=2, N
S-8          A(i) = foo(i)
S-9
S-10         !$omp ordered depend(sink: i-1)
S-11         B(i) = bar(A(i), B(i-1))
S-12         !$omp ordered depend(source)
S-13
S-14         C(i) = baz(B(i))
S-15     end do
S-16 end subroutine

```

The following code is similar to the previous example but with *doacross loop nest* extended to two nested loops, *i* and *j*, as specified by the **ordered(2)** clause on the loop directive. In the C/C++ code, the *i* and *j* loops are the first and second associated loops, respectively, whereas in the Fortran code, the *j* and *i* loops are the first and second associated loops, respectively. The **depend(sink:i-1, j)** and **depend(sink:i, j-1)** clauses in the C/C++ code define cross-iteration dependencies in two dimensions from iterations (*i-1, j*) and (*i, j-1*) to iteration (*i, j*). Likewise, the **depend(sink: j-1, i)** and **depend(sink: j, i-1)** clauses in the Fortran code define cross-iteration dependencies from iterations (*j-1, i*) and (*j, i-1*) to iteration (*j, i*).

Example *doacross.2.c*

```

S-1  float foo(int i, int j);
S-2  float bar(float a, float b, float c);
S-3  float baz(float b);
S-4
S-5  void work( int N, int M, float **A, float **B, float **C )
S-6  {
S-7      int i, j;
S-8
S-9      #pragma omp for ordered(2)
S-10     for (i=1; i<N; i++)
S-11     {
S-12         for (j=1; j<M; j++)
S-13         {

```

```

S-14         A[i][j] = foo(i, j);
S-15
S-16         #pragma omp ordered depend(sink: i-1,j) depend(sink: i,j-1)
S-17         B[i][j] = bar(A[i][j], B[i-1][j], B[i][j-1]);
S-18         #pragma omp ordered depend(source)
S-19
S-20         C[i][j] = baz(B[i][j]);
S-21     }
S-22 }
S-23 }

```

1

Example doacross.2.f90

```

S-1  subroutine work( N, M, A, B, C )
S-2      integer :: N, M, i, j
S-3      real, dimension(M,N) :: A, B, C
S-4      real, external :: foo, bar, baz
S-5
S-6      !$omp do ordered(2)
S-7      do j=2, N
S-8          do i=2, M
S-9              A(i,j) = foo(i, j)
S-10
S-11              !$omp ordered depend(sink: j-1,i) depend(sink: j,i-1)
S-12              B(i,j) = bar(A(i,j), B(i-1,j), B(i,j-1))
S-13              !$omp ordered depend(source)
S-14
S-15              C(i,j) = baz(B(i,j))
S-16          end do
S-17      end do
S-18  end subroutine

```

2
3
4
5
6
7

The following example shows the incorrect use of the **ordered** directive with a **depend** clause. There are two issues with the code. The first issue is a missing **ordered depend(source)** directive, which could cause a deadlock. The second issue is the **depend(sink:i+1, j)** and **depend(sink:i, j+1)** clauses define dependences on lexicographically later source iterations $(i+1, j)$ and $(i, j+1)$, which could cause a deadlock as well since they may not start to execute until the current iteration completes.

1

Example doacross.3.c

```

S-1  #define N 100
S-2
S-3  void work_wrong(double p[][N][N])
S-4  {
S-5      int i, j, k;
S-6
S-7      #pragma omp parallel for ordered(2) private(i,j,k)
S-8          for (i=1; i<N-1; i++)
S-9          {
S-10             for (j=1; j<N-1; j++)
S-11             {
S-12                 #pragma omp ordered depend(sink: i-1,j) depend(sink: i+1,j) \
S-13                     depend(sink: i,j-1) depend(sink: i,j+1)
S-14                 for (k=1; k<N-1; k++)
S-15                 {
S-16                     double tmp1 = p[i-1][j][k] + p[i+1][j][k];
S-17                     double tmp2 = p[i][j-1][k] + p[i][j+1][k];
S-18                     double tmp3 = p[i][j][k-1] + p[i][j][k+1];
S-19                     p[i][j][k] = (tmp1 + tmp2 + tmp3) / 6.0;
S-20                 }
S-21             /* missing #pragma omp ordered depend(source) */
S-22             }
S-23         }
S-24     }

```

2

Example doacross.3.f90

```

S-1  subroutine work_wrong(N, p)
S-2      integer :: N
S-3      real(8), dimension(N,N,N) :: p
S-4      integer :: i, j, k
S-5      real(8) :: tmp1, tmp2, tmp3
S-6
S-7      !$omp parallel do ordered(2) private(i,j,k,tmp1,tmp2,tmp3)
S-8          do i=2, N-1
S-9              do j=2, N-1
S-10                 !$omp ordered depend(sink: i-1,j) depend(sink: i+1,j) &
S-11                 !$omp& depend(sink: i,j-1) depend(sink: i,j+1)
S-12                 do k=2, N-1
S-13                     tmp1 = p(k-1,j,i) + p(k+1,j,i)
S-14                     tmp2 = p(k,j-1,i) + p(k,j+1,i)
S-15                     tmp3 = p(k,j,i-1) + p(k,j,i+1)

```

```

S-16         p(k,j,i) = (tmp1 + tmp2 + tmp3) / 6.0
S-17     end do
S-18     ! missing !$omp ordered depend(source)
S-19     end do
S-20     end do
S-21 end subroutine

```

Fortran

1 The following example illustrates the use of the **collapse** clause for a *doacross loop nest*. The *i*
2 and *j* loops are the associated loops for the collapsed loop as well as for the *doacross loop nest*. The
3 example also shows a compliant usage of the dependence source directive placed before the
4 corresponding sink directive. Checking the completion of computation from previous iterations at
5 the sink point can occur after the source statement.

C / C++

6 *Example doacross.4.c*

```

S-1 double foo(int i, int j);
S-2
S-3 void work( int N, int M, double **A, double **B, double **C )
S-4 {
S-5     int i, j;
S-6     double alpha = 1.2;
S-7
S-8     #pragma omp for collapse(2) ordered(2)
S-9     for (i = 1; i < N-1; i++)
S-10    {
S-11        for (j = 1; j < M-1; j++)
S-12        {
S-13            A[i][j] = foo(i, j);
S-14            #pragma omp ordered depend(source)
S-15
S-16            B[i][j] = alpha * A[i][j];
S-17
S-18            #pragma omp ordered depend(sink: i-1,j) depend(sink: i,j-1)
S-19            C[i][j] = 0.2 * (A[i-1][j] + A[i+1][j] +
S-20                A[i][j-1] + A[i][j+1] + A[i][j]);
S-21        }
S-22    }
S-23 }

```

C / C++

1

Example doacross.4.f90

```

S-1  subroutine work( N, M, A, B, C )
S-2      integer :: N, M
S-3      real(8), dimension(M, N) :: A, B, C
S-4      real(8), external :: foo
S-5      integer :: i, j
S-6      real(8) :: alpha = 1.2
S-7
S-8      !$omp do collapse(2) ordered(2)
S-9      do j=2, N-1
S-10         do i=2, M-1
S-11             A(i,j) = foo(i, j)
S-12             !$omp ordered depend(source)
S-13
S-14             B(i,j) = alpha * A(i,j)
S-15
S-16             !$omp ordered depend(sink: j,i-1) depend(sink: j-1,i)
S-17             C(i,j) = 0.2 * (A(i-1,j) + A(i+1,j) + &
S-18                 A(i,j-1) + A(i,j+1) + A(i,j))
S-19         end do
S-20     end do
S-21 end subroutine

```


1 6.9 Lock Routines

2 This section is about the use of lock routines for synchronization.

3 6.9.1 The `omp_init_lock` Routine

4 The following example demonstrates how to initialize an array of locks in a **parallel** region by
5 using `omp_init_lock`.

C++

6 *Example `init_lock.1.cpp`*

```
S-1 #include <omp.h>
S-2
S-3 omp_lock_t *new_locks()
S-4 {
S-5     int i;
S-6     omp_lock_t *lock = new omp_lock_t[1000];
S-7
S-8     #pragma omp parallel for private(i)
S-9         for (i=0; i<1000; i++)
S-10         {
S-11             omp_init_lock(&lock[i]);
S-12         }
S-13     return lock;
S-14 }
```

C++

Fortran

7 *Example `init_lock.1.f`*

```
S-1      FUNCTION NEW_LOCKS()
S-2          USE OMP_LIB          ! or INCLUDE "omp_lib.h"
S-3          INTEGER(OMP_LOCK_KIND), DIMENSION(1000) :: NEW_LOCKS
S-4
S-5          INTEGER I
S-6
S-7      !$OMP   PARALLEL DO PRIVATE(I)
S-8          DO I=1,1000
S-9              CALL OMP_INIT_LOCK(NEW_LOCKS(I))
S-10          END DO
S-11      !$OMP   END PARALLEL DO
S-12
S-13      END FUNCTION NEW_LOCKS
```

6.9.2 The `omp_init_lock_with_hint` Routine

The following example demonstrates how to initialize an array of locks in a **parallel** region by using `omp_init_lock_with_hint`. Note, hints are combined with an `|` or `+` operator in C/C++ and a `+` operator in Fortran.

C++

Example `init_lock_with_hint.1.cpp`

```
S-1  #include <omp.h>
S-2
S-3  omp_lock_t *new_locks()
S-4  {
S-5      int i;
S-6      omp_lock_t *lock = new omp_lock_t[1000];
S-7
S-8      #pragma omp parallel for private(i)
S-9          for (i=0; i<1000; i++)
S-10         {
S-11             omp_init_lock_with_hint(&lock[i],
S-12                 omp_lock_hint_contended | omp_lock_hint_speculative);
S-13         }
S-14         return lock;
S-15     }
```

C++

Fortran

Example `init_lock_with_hint.1.f`

```
S-1      FUNCTION NEW_LOCKS()
S-2          USE OMP_LIB          ! or INCLUDE "omp_lib.h"
S-3          INTEGER(OMP_LOCK_KIND), DIMENSION(1000) :: NEW_LOCKS
S-4
S-5          INTEGER I
S-6
S-7      !$OMP  PARALLEL DO PRIVATE(I)
S-8          DO I=1,1000
S-9              CALL OMP_INIT_LOCK_WITH_HINT(NEW_LOCKS(I),
S-10          &              OMP_LOCK_HINT_CONTENDED + OMP_LOCK_HINT_SPECULATIVE)
S-11          END DO
S-12      !$OMP  END PARALLEL DO
S-13
S-14      END FUNCTION NEW_LOCKS
```

6.9.3 Ownership of Locks

Ownership of locks has changed since OpenMP 2.5. In OpenMP 2.5, locks are owned by threads; so a lock released by the `omp_unset_lock` routine must be owned by the same thread executing the routine. Beginning with OpenMP 3.0, locks are owned by task regions; so a lock released by the `omp_unset_lock` routine in a task region must be owned by the same task region.

This change in ownership requires extra care when using locks. The following program is conforming in OpenMP 2.5 because the thread that releases the lock `lck` in the parallel region is the same thread that acquired the lock in the sequential part of the program (master thread of parallel region and the initial thread are the same). However, it is not conforming beginning with OpenMP 3.0, because the task region that releases the lock `lck` is different from the task region that acquires the lock.

C / C++

Example lock_owner.1.c

```
S-1  #include <stdlib.h>
S-2  #include <stdio.h>
S-3  #include <omp.h>
S-4
S-5  int main()
S-6  {
S-7      int x;
S-8      omp_lock_t lck;
S-9
S-10     omp_init_lock (&lck);
S-11     omp_set_lock (&lck);
S-12     x = 0;
S-13
S-14     #pragma omp parallel shared (x)
S-15     {
S-16         #pragma omp master
S-17         {
S-18             x = x + 1;
S-19             omp_unset_lock (&lck);
S-20         }
S-21
S-22         /* Some more stuff. */
S-23     }
S-24     omp_destroy_lock (&lck);
S-25     return 0;
S-26 }
```

Example *lock_owner.f*

```

S-1      program lock
S-2      use omp_lib
S-3      integer :: x
S-4      integer (kind=omp_lock_kind) :: lck
S-5
S-6      call omp_init_lock (lck)
S-7      call omp_set_lock(lck)
S-8      x = 0
S-9
S-10     !$omp parallel shared (x)
S-11     !$omp master
S-12         x = x + 1
S-13         call omp_unset_lock(lck)
S-14     !$omp end master
S-15
S-16     !      Some more stuff.
S-17     !$omp end parallel
S-18
S-19         call omp_destroy_lock(lck)
S-20     end

```

2 6.9.4 Simple Lock Routines

In the following example, the lock routines cause the threads to be idle while waiting for entry to the first critical section, but to do other work while waiting for entry to the second. The **omp_set_lock** function blocks, but the **omp_test_lock** function does not, allowing the work in **skip** to be done.

Note that the argument to the lock routines should have type **omp_lock_t**, and that there is no need to flush it.

1

Example simple_lock.1.c

```

S-1  #include <stdio.h>
S-2  #include <omp.h>
S-3  void skip(int i) {}
S-4  void work(int i) {}
S-5  int main()
S-6  {
S-7      omp_lock_t lck;
S-8      int id;
S-9      omp_init_lock(&lck);
S-10
S-11      #pragma omp parallel shared(lck) private(id)
S-12      {
S-13          id = omp_get_thread_num();
S-14
S-15          omp_set_lock(&lck);
S-16          /* only one thread at a time can execute this printf */
S-17          printf("My thread id is %d.\n", id);
S-18          omp_unset_lock(&lck);
S-19
S-20          while (! omp_test_lock(&lck)) {
S-21              skip(id);    /* we do not yet have the lock,
S-22                          so we must do something else */
S-23          }
S-24
S-25          work(id);        /* we now have the lock
S-26                          and can do the work */
S-27
S-28          omp_unset_lock(&lck);
S-29      }
S-30      omp_destroy_lock(&lck);
S-31
S-32      return 0;
S-33  }

```

2

Note that there is no need to flush the lock variable.

1

Example simple_lock.1.f

```

S-1      SUBROUTINE SKIP(ID)
S-2      END SUBROUTINE SKIP
S-3
S-4      SUBROUTINE WORK(ID)
S-5      END SUBROUTINE WORK
S-6
S-7      PROGRAM SIMPLELOCK
S-8
S-9          INCLUDE "omp_lib.h"      ! or USE OMP_LIB
S-10
S-11          INTEGER(OMP_LOCK_KIND) LCK
S-12          INTEGER ID
S-13
S-14          CALL OMP_INIT_LOCK(LCK)
S-15
S-16      !$OMP  PARALLEL SHARED(LCK) PRIVATE(ID)
S-17              ID = OMP_GET_THREAD_NUM()
S-18              CALL OMP_SET_LOCK(LCK)
S-19              PRINT *, 'My thread id is ', ID
S-20              CALL OMP_UNSET_LOCK(LCK)
S-21
S-22              DO WHILE (.NOT. OMP_TEST_LOCK(LCK))
S-23                  CALL SKIP(ID)      ! We do not yet have the lock
S-24                                  ! so we must do something else
S-25              END DO
S-26
S-27              CALL WORK(ID)          ! We now have the lock
S-28                                  ! and can do the work
S-29
S-30              CALL OMP_UNSET_LOCK( LCK )
S-31
S-32      !$OMP  END PARALLEL
S-33
S-34          CALL OMP_DESTROY_LOCK( LCK )
S-35
S-36      END PROGRAM SIMPLELOCK

```

1 6.9.5 Nestable Lock Routines

2 The following example demonstrates how a nestable lock can be used to synchronize updates both
3 to a whole structure and to one of its members.

C / C++

4 *Example nestable_lock.1.c*

```
S-1  #include <omp.h>
S-2  typedef struct {
S-3      int a,b;
S-4      omp_nest_lock_t lck; } pair;
S-5
S-6  int work1();
S-7  int work2();
S-8  int work3();
S-9  void incr_a(pair *p, int a)
S-10 {
S-11     /* Called only from incr_pair, no need to lock. */
S-12     p->a += a;
S-13 }
S-14 void incr_b(pair *p, int b)
S-15 {
S-16     /* Called both from incr_pair and elsewhere, */
S-17     /* so need a nestable lock. */
S-18
S-19     omp_set_nest_lock(&p->lck);
S-20     p->b += b;
S-21     omp_unset_nest_lock(&p->lck);
S-22 }
S-23 void incr_pair(pair *p, int a, int b)
S-24 {
S-25     omp_set_nest_lock(&p->lck);
S-26     incr_a(p, a);
S-27     incr_b(p, b);
S-28     omp_unset_nest_lock(&p->lck);
S-29 }
S-30 void nestlock(pair *p)
S-31 {
S-32     #pragma omp parallel sections
S-33     {
S-34         #pragma omp section
S-35         incr_pair(p, work1(), work2());
S-36         #pragma omp section
S-37         incr_b(p, work3());
S-38     }
S-39 }
```

1

Example nestable_lock.1.f

```

S-1      MODULE DATA
S-2          USE OMP_LIB, ONLY: OMP_NEST_LOCK_KIND
S-3          TYPE LOCKED_PAIR
S-4              INTEGER A
S-5              INTEGER B
S-6              INTEGER (OMP_NEST_LOCK_KIND) LCK
S-7      END TYPE
S-8      END MODULE DATA
S-9
S-10     SUBROUTINE INCR_A(P, A)
S-11         ! called only from INCR_PAIR, no need to lock
S-12         USE DATA
S-13         TYPE (LOCKED_PAIR) :: P
S-14         INTEGER A
S-15         P%A = P%A + A
S-16     END SUBROUTINE INCR_A
S-17
S-18     SUBROUTINE INCR_B(P, B)
S-19         ! called from both INCR_PAIR and elsewhere,
S-20         ! so we need a nestable lock
S-21         USE OMP_LIB          ! or INCLUDE "omp_lib.h"
S-22         USE DATA
S-23         TYPE (LOCKED_PAIR) :: P
S-24         INTEGER B
S-25         CALL OMP_SET_NEST_LOCK(P%LCK)
S-26         P%B = P%B + B
S-27         CALL OMP_UNSET_NEST_LOCK(P%LCK)
S-28     END SUBROUTINE INCR_B
S-29
S-30     SUBROUTINE INCR_PAIR(P, A, B)
S-31         USE OMP_LIB          ! or INCLUDE "omp_lib.h"
S-32         USE DATA
S-33         TYPE (LOCKED_PAIR) :: P
S-34         INTEGER A
S-35         INTEGER B
S-36
S-37         CALL OMP_SET_NEST_LOCK(P%LCK)
S-38         CALL INCR_A(P, A)
S-39         CALL INCR_B(P, B)
S-40         CALL OMP_UNSET_NEST_LOCK(P%LCK)
S-41     END SUBROUTINE INCR_PAIR
S-42

```



```

S-43      SUBROUTINE NESTLOCK(P)
S-44          USE OMP_LIB          ! or INCLUDE "omp_lib.h"
S-45          USE DATA
S-46          TYPE(LOCKED_PAIR) :: P
S-47          INTEGER WORK1, WORK2, WORK3
S-48          EXTERNAL WORK1, WORK2, WORK3
S-49
S-50      !$OMP    PARALLEL SECTIONS
S-51
S-52      !$OMP    SECTION
S-53          CALL INCR_PAIR(P, WORK1(), WORK2())
S-54      !$OMP    SECTION
S-55          CALL INCR_B(P, WORK3())
S-56      !$OMP    END PARALLEL SECTIONS
S-57
S-58      END SUBROUTINE NESTLOCK

```

Fortran

Data Environment

The OpenMP *data environment* contains data attributes of variables and objects. Many constructs (such as **parallel**, **simd**, **task**) accept clauses to control *data-sharing* attributes of referenced variables in the construct, where *data-sharing* applies to whether the attribute of the variable is *shared*, is *private* storage, or has special operational characteristics (as found in the **firstprivate**, **lastprivate**, **linear**, or **reduction** clause).

The data environment for a device (distinguished as a *device data environment*) is controlled on the host by *data-mapping* attributes, which determine the relationship of the data on the host, the *original* data, and the data on the device, the *corresponding* data.

DATA-SHARING ATTRIBUTES

Data-sharing attributes of variables can be classified as being *predetermined*, *explicitly determined* or *implicitly determined*.

Certain variables and objects have predetermined attributes. A commonly found case is the loop iteration variable in associated loops of a **for** or **do** construct. It has a private data-sharing attribute. Variables with predetermined data-sharing attributes can not be listed in a data-sharing clause; but there are some exceptions (mainly concerning loop iteration variables).

Variables with explicitly determined data-sharing attributes are those that are referenced in a given construct and are listed in a data-sharing attribute clause on the construct. Some of the common data-sharing clauses are: **shared**, **private**, **firstprivate**, **lastprivate**, **linear**, and **reduction**.

Variables with implicitly determined data-sharing attributes are those that are referenced in a given construct, do not have predetermined data-sharing attributes, and are not listed in a data-sharing attribute clause of an enclosing construct. For a complete list of variables and objects with predetermined and implicitly determined attributes, please refer to the *Data-sharing Attribute Rules for Variables Referenced in a Construct* subsection of the OpenMP Specifications document.

DATA-MAPPING ATTRIBUTES

The **map** clause on a device construct explicitly specifies how the list items in the clause are mapped from the encountering task's data environment (on the host) to the corresponding item in the device data environment (on the device). The common *list items* are arrays, array sections, scalars, pointers, and structure elements (members).

Procedures and global variables have predetermined data mapping if they appear within the list or block of a **declare target** directive. Also, a C/C++ pointer is mapped as a zero-length array section, as is a C++ variable that is a reference to a pointer.

Without explicit mapping, non-scalar and non-pointer variables within the scope of the **target** construct are implicitly mapped with a *map-type* of **tofrom**. Without explicit mapping, scalar variables within the scope of the **target** construct are not mapped, but have an implicit firstprivate data-sharing attribute. (That is, the value of the original variable is given to a private variable of the same name on the device.) This behavior can be changed with the **defaultmap** clause.

The **map** clause can appear on **target**, **target data** and **target enter/exit data** constructs. The operations of creation and removal of device storage as well as assignment of the original list item values to the corresponding list items may be complicated when the list item appears on multiple constructs or when the host and device storage is shared. In these cases the item's reference count, the number of times it has been referenced (+1 on entry and -1 on exited) in nested (structured) map regions and/or accumulative (unstructured) mappings, determines the operation. Details of the **map** clause and reference count operation are specified in the *map Clause* subsection of the OpenMP Specifications document.

1 7.1 The `threadprivate` Directive

2 The following examples demonstrate how to use the **threadprivate** directive to give each
3 thread a separate counter.

▼ C / C++ ▼

4 *Example threadprivate.1.c*

```
S-1 int counter = 0;  
S-2 #pragma omp threadprivate(counter)  
S-3  
S-4 int increment_counter()  
S-5 {  
S-6     counter++;  
S-7     return(counter);  
S-8 }
```

▲ C / C++ ▲
▼ Fortran ▼

5 *Example threadprivate.1.f*

```
S-1     INTEGER FUNCTION INCREMENT_COUNTER()  
S-2         COMMON/INC_COMMON/COUNTER  
S-3     !$OMP    THREADPRIVATE (/INC_COMMON/)  
S-4  
S-5         COUNTER = COUNTER +1  
S-6         INCREMENT_COUNTER = COUNTER  
S-7         RETURN  
S-8     END FUNCTION INCREMENT_COUNTER
```

▲ Fortran ▲
▼ C / C++ ▼

6 The following example uses **threadprivate** on a static variable:

7 *Example threadprivate.2.c*

```
S-1 int increment_counter_2()  
S-2 {  
S-3     static int counter = 0;  
S-4     #pragma omp threadprivate(counter)  
S-5     counter++;  
S-6     return(counter);  
S-7 }
```

The following example demonstrates unspecified behavior for the initialization of a **threadprivate** variable. A **threadprivate** variable is initialized once at an unspecified point before its first reference. Because **a** is constructed using the value of **x** (which is modified by the statement **x++**), the value of **a.val** at the start of the **parallel** region could be either 1 or 2. This problem is avoided for **b**, which uses an auxiliary **const** variable and a copy-constructor.

Example threadprivate.3.cpp

```
S-1  class T {
S-2      public:
S-3          int val;
S-4          T (int);
S-5          T (const T&);
S-6      };
S-7
S-8  T :: T (int v){
S-9      val = v;
S-10 }
S-11
S-12 T :: T (const T& t) {
S-13     val = t.val;
S-14 }
S-15
S-16 void g(T a, T b){
S-17     a.val += b.val;
S-18 }
S-19
S-20 int x = 1;
S-21 T a(x);
S-22 const T b_aux(x); /* Capture value of x = 1 */
S-23 T b(b_aux);
S-24 #pragma omp threadprivate(a, b)
S-25
S-26 void f(int n) {
S-27     x++;
S-28     #pragma omp parallel for
S-29     /* In each thread:
S-30      * a is constructed from x (with value 1 or 2?)
S-31      * b is copy-constructed from b_aux
S-32      */
S-33
S-34     for (int i=0; i<n; i++) {
S-35         g(a, b); /* Value of a is unspecified. */
S-36     }
S-37 }
```

— C / C++ —

The following examples show non-conforming uses and correct uses of the **threadprivate** directive.

Fortran

The following example is non-conforming because the common block is not declared local to the subroutine that refers to it:

Example threadprivate.2.f

```
S-1      MODULE INC_MODULE
S-2      COMMON /T/ A
S-3      END MODULE INC_MODULE
S-4
S-5      SUBROUTINE INC_MODULE_WRONG()
S-6      USE INC_MODULE
S-7      !$OMP THREADPRIVATE (/T/)
S-8      !non-conforming because /T/ not declared in INC_MODULE_WRONG
S-9      END SUBROUTINE INC_MODULE_WRONG
```

The following example is also non-conforming because the common block is not declared local to the subroutine that refers to it:

Example threadprivate.3.f

```
S-1      SUBROUTINE INC_WRONG()
S-2      COMMON /T/ A
S-3      !$OMP THREADPRIVATE (/T/)
S-4
S-5      CONTAINS
S-6      SUBROUTINE INC_WRONG_SUB()
S-7      !$OMP PARALLEL COPYIN (/T/)
S-8      !non-conforming because /T/ not declared in INC_WRONG_SUB
S-9      !$OMP END PARALLEL
S-10     END SUBROUTINE INC_WRONG_SUB
S-11     END SUBROUTINE INC_WRONG
```

The following example is a correct rewrite of the previous example:

Example threadprivate.4.f

```
S-1      SUBROUTINE INC_GOOD()
S-2      COMMON /T/ A
S-3      !$OMP THREADPRIVATE (/T/)
S-4
S-5      CONTAINS
S-6      SUBROUTINE INC_GOOD_SUB()
S-7      COMMON /T/ A
S-8      !$OMP THREADPRIVATE (/T/)
S-9
S-10     !$OMP PARALLEL COPYIN (/T/)
```

```
S-11      !$OMP          END PARALLEL
S-12      END SUBROUTINE INC_GOOD_SUB
S-13      END SUBROUTINE INC_GOOD
```

The following is an example of the use of **threadprivate** for local variables:

Example threadprivate.5.f

```
S-1      PROGRAM INC_GOOD2
S-2          INTEGER, ALLOCATABLE, SAVE :: A(:)
S-3          INTEGER, POINTER, SAVE :: PTR
S-4          INTEGER, SAVE :: I
S-5          INTEGER, TARGET :: TARG
S-6          LOGICAL :: FIRSTIN = .TRUE.
S-7      !$OMP  THREADPRIVATE(A, I, PTR)
S-8
S-9          ALLOCATE (A(3))
S-10         A = (/1,2,3/)
S-11         PTR => TARG
S-12         I = 5
S-13
S-14      !$OMP  PARALLEL COPYIN(I, PTR)
S-15      !$OMP  CRITICAL
S-16          IF (FIRSTIN) THEN
S-17              TARG = 4          ! Update target of ptr
S-18              I = I + 10
S-19              IF (ALLOCATED(A)) A = A + 10
S-20              FIRSTIN = .FALSE.
S-21          END IF
S-22
S-23          IF (ALLOCATED(A)) THEN
S-24              PRINT *, 'a = ', A
S-25          ELSE
S-26              PRINT *, 'A is not allocated'
S-27          END IF
S-28
S-29          PRINT *, 'ptr = ', PTR
S-30          PRINT *, 'i = ', I
S-31          PRINT *
S-32
S-33      !$OMP  END CRITICAL
S-34      !$OMP  END PARALLEL
S-35      END PROGRAM INC_GOOD2
```

The above program, if executed by two threads, will print one of the following two sets of output:

```

1      a = 11 12 13
2      ptr = 4
3      i = 15

4      A is not allocated
5      ptr = 4
6      i = 5

7      or

8      A is not allocated
9      ptr = 4
10     i = 15

11     a = 1 2 3
12     ptr = 4
13     i = 5

```

14 The following is an example of the use of **threadprivate** for module variables:

15 *Example threadprivate.6.f*

```

S-1      MODULE INC_MODULE_GOOD3
S-2          REAL, POINTER :: WORK(:)
S-3          SAVE WORK
S-4      !$OMP    THREADPRIVATE(WORK)
S-5          END MODULE INC_MODULE_GOOD3
S-6
S-7      SUBROUTINE SUB1(N)
S-8          USE INC_MODULE_GOOD3
S-9      !$OMP    PARALLEL PRIVATE(THE_SUM)
S-10         ALLOCATE(WORK(N))
S-11         CALL SUB2(THE_SUM)
S-12         WRITE(*,*)THE_SUM
S-13     !$OMP    END PARALLEL
S-14     END SUBROUTINE SUB1
S-15
S-16     SUBROUTINE SUB2(THE_SUM)
S-17         USE INC_MODULE_GOOD3
S-18         WORK(:) = 10
S-19         THE_SUM=SUM(WORK)
S-20     END SUBROUTINE SUB2
S-21
S-22     PROGRAM INC_GOOD3
S-23         N = 10
S-24         CALL SUB1(N)

```


S-25

END PROGRAM INC_GOOD3

Fortran

C++

1 The following example illustrates initialization of **threadprivate** variables for class-type **T**. **t1**
2 is default constructed, **t2** is constructed taking a constructor accepting one argument of integer
3 type, **t3** is copy constructed with argument **f()** :

4 *Example threadprivate.4.cpp*

```
S-1 static T t1;  
S-2 #pragma omp threadprivate(t1)  
S-3 static T t2( 23 );  
S-4 #pragma omp threadprivate(t2)  
S-5 static T t3 = f();  
S-6 #pragma omp threadprivate(t3)
```

5 The following example illustrates the use of **threadprivate** for static class members. The
6 **threadprivate** directive for a static class member must be placed inside the class definition.

7 *Example threadprivate.5.cpp*

```
S-1 class T {  
S-2     public:  
S-3         static int i;  
S-4     #pragma omp threadprivate(i)  
S-5 };
```

C++

1 7.2 The default (none) Clause

2 The following example distinguishes the variables that are affected by the **default (none)**
3 clause from those that are not.

▼ C / C++ ▼

4 Beginning with OpenMP 4.0, variables with **const**-qualified type and no mutable member are no
5 longer predetermined shared. Thus, these variables (variable *c* in the example) need to be explicitly
6 listed in data-sharing attribute clauses when the **default (none)** clause is specified.

7 *Example default_none.1.c*

```
S-1 #include <omp.h>
S-2 int x, y, z[1000];
S-3 #pragma omp threadprivate(x)
S-4
S-5 void default_none(int a) {
S-6     const int c = 1;
S-7     int i = 0;
S-8
S-9     #pragma omp parallel default(none) private(a) shared(z, c)
S-10    {
S-11        int j = omp_get_num_threads();
S-12        /* O.K. - j is declared within parallel region */
S-13        a = z[j]; /* O.K. - a is listed in private clause */
S-14                /* - z is listed in shared clause */
S-15        x = c; /* O.K. - x is threadprivate */
S-16                /* - c has const-qualified type and
S-17                is listed in shared clause */
S-18        z[i] = y; /* Error - cannot reference i or y here */
S-19
S-20        #pragma omp for firstprivate(y)
S-21        /* Error - Cannot reference y in the firstprivate clause */
S-22        for (i=0; i<10 ; i++) {
S-23            z[i] = i; /* O.K. - i is the loop iteration variable */
S-24        }
S-25
S-26        z[i] = y; /* Error - cannot reference i or y here */
S-27    }
S-28 }
```

▲ C / C++ ▲

1

Example default_none.f

```

S-1      SUBROUTINE DEFAULT_NONE(A)
S-2      INCLUDE "omp_lib.h"      ! or USE OMP_LIB
S-3
S-4      INTEGER A
S-5
S-6      INTEGER X, Y, Z(1000)
S-7      COMMON/BLOCKX/X
S-8      COMMON/BLOCKY/Y
S-9      COMMON/BLOCKZ/Z
S-10     !$OMP THREADPRIVATE (/BLOCKX/)
S-11
S-12      INTEGER I, J
S-13      i = 1
S-14
S-15     !$OMP  PARALLEL DEFAULT(NONE) PRIVATE(A) SHARED(Z) PRIVATE(J)
S-16          J = OMP_GET_NUM_THREADS();
S-17              ! O.K. - J is listed in PRIVATE clause
S-18          A = Z(J) ! O.K. - A is listed in PRIVATE clause
S-19              ! - Z is listed in SHARED clause
S-20          X = 1    ! O.K. - X is THREADPRIVATE
S-21          Z(I) = Y ! Error - cannot reference I or Y here
S-22
S-23     !$OMP DO firstprivate(y)
S-24         ! Error - Cannot reference y in the firstprivate clause
S-25         DO I = 1,10
S-26             Z(I) = I ! O.K. - I is the loop iteration variable
S-27         END DO
S-28
S-29
S-30         Z(I) = Y    ! Error - cannot reference I or Y here
S-31     !$OMP  END PARALLEL
S-32     END SUBROUTINE DEFAULT_NONE

```

1 7.3 The private Clause

2 In the following example, the values of original list items *i* and *j* are retained on exit from the
3 **parallel** region, while the private list items *i* and *j* are modified within the **parallel**
4 construct.

C / C++

5 *Example private.1.c*

```
S-1 #include <stdio.h>
S-2 #include <assert.h>
S-3
S-4 int main()
S-5 {
S-6     int i, j;
S-7     int *ptr_i, *ptr_j;
S-8
S-9     i = 1;
S-10    j = 2;
S-11
S-12    ptr_i = &i;
S-13    ptr_j = &j;
S-14
S-15    #pragma omp parallel private(i) firstprivate(j)
S-16    {
S-17        i = 3;
S-18        j = j + 2;
S-19        assert (*ptr_i == 1 && *ptr_j == 2);
S-20    }
S-21
S-22    assert(i == 1 && j == 2);
S-23
S-24    return 0;
S-25 }
```

C / C++

Fortran

Example private.1.f

```

S-1      PROGRAM PRIV_EXAMPLE
S-2      INTEGER I, J
S-3
S-4      I = 1
S-5      J = 2
S-6
S-7      !$OMP  PARALLEL PRIVATE(I) FIRSTPRIVATE(J)
S-8          I = 3
S-9          J = J + 2
S-10     !$OMP  END PARALLEL
S-11
S-12     PRINT *, I, J ! I .eq. 1 .and. J .eq. 2
S-13     END PROGRAM PRIV_EXAMPLE

```

Fortran

In the following example, all uses of the variable *a* within the loop construct in the routine *f* refer to a private list item *a*, while it is unspecified whether references to *a* in the routine *g* are to a private list item or the original list item.

C / C++

Example private.2.c

```

S-1      int a;
S-2
S-3      void g(int k) {
S-4          a = k; /* Accessed in the region but outside of the construct;
S-5                  * therefore unspecified whether original or private list
S-6                  * item is modified. */
S-7      }
S-8
S-9
S-10     void f(int n) {
S-11         int a = 0;
S-12
S-13         #pragma omp parallel for private(a)
S-14         for (int i=1; i<n; i++) {
S-15             a = i;
S-16             g(a*2); /* Private copy of "a" */
S-17         }
S-18     }

```

C / C++

Example private.2.f

```

S-1      MODULE PRIV_EXAMPLE2
S-2      REAL A
S-3
S-4      CONTAINS
S-5
S-6      SUBROUTINE G(K)
S-7      REAL K
S-8      A = K ! Accessed in the region but outside of the
S-9             ! construct; therefore unspecified whether
S-10            ! original or private list item is modified.
S-11      END SUBROUTINE G
S-12
S-13      SUBROUTINE F(N)
S-14      INTEGER N
S-15      REAL A
S-16
S-17      INTEGER I
S-18  !$OMP  PARALLEL DO PRIVATE(A)
S-19          DO I = 1,N
S-20              A = I
S-21              CALL G(A*2)
S-22          ENDDO
S-23  !$OMP  END PARALLEL DO
S-24      END SUBROUTINE F
S-25
S-26      END MODULE PRIV_EXAMPLE2

```

The following example demonstrates that a list item that appears in a **private** clause in a **parallel** construct may also appear in a **private** clause in an enclosed worksharing construct, which results in an additional private copy.

Example private.3.c

```

S-1  #include <assert.h>
S-2  void priv_example3()
S-3  {
S-4      int i, a;
S-5
S-6      #pragma omp parallel private(a)
S-7      {
S-8          a = 1;
S-9      #pragma omp parallel for private(a)

```

```

S-10      for (i=0; i<10; i++)
S-11      {
S-12          a = 2;
S-13      }
S-14      assert(a == 1);
S-15  }
S-16  }

```

C / C++

Fortran

1

Example private.3.f

```

S-1      SUBROUTINE PRIV_EXAMPLE3()
S-2      INTEGER I, A
S-3
S-4      !$OMP PARALLEL PRIVATE(A)
S-5          A = 1
S-6      !$OMP PARALLEL DO PRIVATE(A)
S-7          DO I = 1, 10
S-8              A = 2
S-9          END DO
S-10     !$OMP END PARALLEL DO
S-11     PRINT *, A ! Outer A still has value 1
S-12     !$OMP END PARALLEL
S-13     END SUBROUTINE PRIV_EXAMPLE3

```

Fortran

1 7.4 Fortran Private Loop Iteration Variables

Fortran

2 In general loop iteration variables will be private, when used in the *do-loop* of a **do** and
3 **parallel do** construct or in sequential loops in a **parallel** construct (see Section 2.7.1 and
4 Section 2.14.1 of the OpenMP 4.0 specification). In the following example of a sequential loop in a
5 **parallel** construct the loop iteration variable *I* will be private.

6 *Example fort_loopvar.1.f90*

```
S-1 SUBROUTINE PLOOP_1(A,N)
S-2 INCLUDE "omp_lib.h"          ! or USE OMP_LIB
S-3
S-4 REAL A(*)
S-5 INTEGER I, MYOFFSET, N
S-6
S-7 !$OMP PARALLEL PRIVATE(MYOFFSET)
S-8     MYOFFSET = OMP_GET_THREAD_NUM() * N
S-9     DO I = 1, N
S-10        A(MYOFFSET+I) = FLOAT(I)
S-11     ENDDO
S-12 !$OMP END PARALLEL
S-13
S-14 END SUBROUTINE PLOOP_1
```

7 In exceptional cases, loop iteration variables can be made shared, as in the following example:

8 *Example fort_loopvar.2.f90*

```
S-1 SUBROUTINE PLOOP_2(A,B,N,I1,I2)
S-2 REAL A(*), B(*)
S-3 INTEGER I1, I2, N
S-4
S-5 !$OMP PARALLEL SHARED(A,B,I1,I2)
S-6 !$OMP SECTIONS
S-7 !$OMP SECTION
S-8     DO I1 = I1, N
S-9         IF (A(I1).NE.0.0) EXIT
S-10    ENDDO
S-11 !$OMP SECTION
S-12     DO I2 = I2, N
S-13         IF (B(I2).NE.0.0) EXIT
S-14    ENDDO
S-15 !$OMP END SECTIONS
S-16 !$OMP SINGLE
S-17     IF (I1.LE.N) PRINT *, 'ITEMS IN A UP TO ', I1, 'ARE ALL ZERO.'
S-18     IF (I2.LE.N) PRINT *, 'ITEMS IN B UP TO ', I2, 'ARE ALL ZERO.'
```



```
S-19  !$OMP END SINGLE
S-20  !$OMP END PARALLEL
S-21
S-22  END SUBROUTINE PLOOP_2
```

1 Note however that the use of shared loop iteration variables can easily lead to race conditions.

▲ Fortran ▲

7.5 Fortran Restrictions on shared and private Clauses with Common Blocks

Fortran

When a named common block is specified in a **private**, **firstprivate**, or **lastprivate** clause of a construct, none of its members may be declared in another data-sharing attribute clause on that construct. The following examples illustrate this point.

The following example is conforming:

Example fort_sp_common.1.f

```
S-1      SUBROUTINE COMMON_GOOD ()
S-2      COMMON /C/ X,Y
S-3      REAL X, Y
S-4
S-5      !$OMP PARALLEL PRIVATE (/C/)
S-6      ! do work here
S-7      !$OMP END PARALLEL
S-8      !$OMP PARALLEL SHARED (X,Y)
S-9      ! do work here
S-10     !$OMP END PARALLEL
S-11     END SUBROUTINE COMMON_GOOD
```

The following example is also conforming:

Example fort_sp_common.2.f

```
S-1      SUBROUTINE COMMON_GOOD2 ()
S-2      COMMON /C/ X,Y
S-3      REAL X, Y
S-4      INTEGER I
S-5      !$OMP PARALLEL
S-6      !$OMP DO PRIVATE (/C/)
S-7      DO I=1,1000
S-8      ! do work here
S-9      ENDDO
S-10     !$OMP END DO
S-11     !$OMP DO PRIVATE (X)
S-12     DO I=1,1000
S-13     ! do work here
S-14     ENDDO
S-15     !$OMP END DO
S-16     !$OMP END PARALLEL
S-17     END SUBROUTINE COMMON_GOOD2
```

The following example is conforming:

Example fort_sp_common.3.f

```
S-1      SUBROUTINE COMMON_GOOD3 ()
S-2      COMMON /C/ X,Y
S-3      !$OMP  PARALLEL PRIVATE (/C/)
S-4      ! do work here
S-5      !$OMP  END PARALLEL
S-6      !$OMP  PARALLEL SHARED (/C/)
S-7      ! do work here
S-8      !$OMP  END PARALLEL
S-9      END SUBROUTINE COMMON_GOOD3
```

The following example is non-conforming because **x** is a constituent element of **c**:

Example fort_sp_common.4.f

```
S-1      SUBROUTINE COMMON_WRONG ()
S-2      COMMON /C/ X,Y
S-3      ! Incorrect because X is a constituent element of C
S-4      !$OMP  PARALLEL PRIVATE (/C/), SHARED(X)
S-5      ! do work here
S-6      !$OMP  END PARALLEL
S-7      END SUBROUTINE COMMON_WRONG
```

The following example is non-conforming because a common block may not be declared both shared and private:

Example fort_sp_common.5.f

```
S-1      SUBROUTINE COMMON_WRONG2 ()
S-2      COMMON /C/ X,Y
S-3      ! Incorrect: common block C cannot be declared both
S-4      ! shared and private
S-5      !$OMP  PARALLEL PRIVATE (/C/), SHARED (/C/)
S-6      ! do work here
S-7      !$OMP  END PARALLEL
S-8
S-9      END SUBROUTINE COMMON_WRONG2
```

7.6 Fortran Restrictions on Storage Association with the `private` Clause

Fortran

The following non-conforming examples illustrate the implications of the **private** clause rules with regard to storage association.

Example fort_sa_private.1.f

```
S-1      SUBROUTINE SUB ()
S-2      COMMON /BLOCK/ X
S-3      PRINT *,X                ! X is undefined
S-4      END SUBROUTINE SUB
S-5
S-6      PROGRAM PRIV_RESTRICT
S-7      COMMON /BLOCK/ X
S-8      X = 1.0
S-9      !$OMP PARALLEL PRIVATE (X)
S-10     X = 2.0
S-11     CALL SUB ()
S-12     !$OMP END PARALLEL
S-13     END PROGRAM PRIV_RESTRICT
```

Example fort_sa_private.2.f

```
S-1      PROGRAM PRIV_RESTRICT2
S-2      COMMON /BLOCK2/ X
S-3      X = 1.0
S-4
S-5      !$OMP PARALLEL PRIVATE (X)
S-6      X = 2.0
S-7      CALL SUB ()
S-8      !$OMP END PARALLEL
S-9
S-10     CONTAINS
S-11
S-12     SUBROUTINE SUB ()
S-13     COMMON /BLOCK2/ Y
S-14
S-15     PRINT *,X                ! X is undefined
S-16     PRINT *,Y                ! Y is undefined
S-17     END SUBROUTINE SUB
S-18
S-19     END PROGRAM PRIV_RESTRICT2
```

Example fort_sa_private.3.f

```

S-1      PROGRAM PRIV_RESTRICT3
S-2      EQUIVALENCE (X,Y)
S-3      X = 1.0
S-4
S-5      !$OMP  PARALLEL PRIVATE(X)
S-6          PRINT *,Y                ! Y is undefined
S-7          Y = 10
S-8          PRINT *,X                ! X is undefined
S-9      !$OMP  END PARALLEL
S-10     END PROGRAM PRIV_RESTRICT3

```

1 *Example fort_sa_private.4.f*

```

S-1      PROGRAM PRIV_RESTRICT4
S-2      INTEGER I, J
S-3      INTEGER A(100), B(100)
S-4      EQUIVALENCE (A(51), B(1))
S-5
S-6      !$OMP PARALLEL DO DEFAULT(PRIVATE) PRIVATE(I,J) LASTPRIVATE(A)
S-7          DO I=1,100
S-8              DO J=1,100
S-9                  B(J) = J - 1
S-10             ENDDO
S-11
S-12             DO J=1,100
S-13                 A(J) = J      ! B becomes undefined at this point
S-14             ENDDO
S-15
S-16             DO J=1,50
S-17                 B(J) = B(J) + 1 ! B is undefined
S-18                                 ! A becomes undefined at this point
S-19             ENDDO
S-20         ENDDO
S-21     !$OMP END PARALLEL DO      ! The LASTPRIVATE write for A has
S-22                                 ! undefined results
S-23
S-24         PRINT *, B             ! B is undefined since the LASTPRIVATE
S-25                                 ! write of A was not defined
S-26     END PROGRAM PRIV_RESTRICT4

```

2 *Example fort_sa_private.5.f*

```

S-1      SUBROUTINE SUB1(X)
S-2      DIMENSION X(10)
S-3
S-4      ! This use of X does not conform to the

```

```

S-5      ! specification. It would be legal Fortran 90,
S-6      ! but the OpenMP private directive allows the
S-7      ! compiler to break the sequence association that
S-8      ! A had with the rest of the common block.
S-9
S-10     forall (I = 1:10) X(I) = I
S-11     end subroutine SUB1
S-12
S-13     PROGRAM PRIV_RESTRICT5
S-14     COMMON /BLOCK5/ A
S-15
S-16     DIMENSION B(10)
S-17     EQUIVALENCE (A,B(1))
S-18
S-19     ! the common block has to be at least 10 words
S-20     A = 0
S-21
S-22     !$OMP PARALLEL PRIVATE(/BLOCK5/)
S-23
S-24         ! Without the private clause,
S-25         ! we would be passing a member of a sequence
S-26         ! that is at least ten elements long.
S-27         ! With the private clause, A may no longer be
S-28         ! sequence-associated.
S-29
S-30         CALL SUB1(A)
S-31     !$OMP MASTER
S-32         PRINT *, A
S-33     !$OMP END MASTER
S-34
S-35     !$OMP END PARALLEL
S-36     END PROGRAM PRIV_RESTRICT5

```

Fortran

1 7.7 C/C++ Arrays in a `firstprivate` Clause

▼ C / C++ ▼

2 The following example illustrates the size and value of list items of array or pointer type in a
3 **firstprivate** clause . The size of new list items is based on the type of the corresponding
4 original list item, as determined by the base language.

5 In this example:

- 6 • The type of **A** is array of two arrays of two ints.
- 7 • The type of **B** is adjusted to pointer to array of **n** ints, because it is a function parameter.
- 8 • The type of **C** is adjusted to pointer to int, because it is a function parameter.
- 9 • The type of **D** is array of two arrays of two ints.
- 10 • The type of **E** is array of **n** arrays of **n** ints.

11 Note that **B** and **E** involve variable length array types.

12 The new items of array type are initialized as if each integer element of the original array is
13 assigned to the corresponding element of the new array. Those of pointer type are initialized as if
14 by assignment from the original item to the new item.

15 *Example `carrays_fpriv.1.c`*

```
S-1  #include <assert.h>
S-2
S-3  int A[2][2] = {1, 2, 3, 4};
S-4
S-5  void f(int n, int B[n][n], int C[])
S-6  {
S-7      int D[2][2] = {1, 2, 3, 4};
S-8      int E[n][n];
S-9
S-10     assert(n >= 2);
S-11     E[1][1] = 4;
S-12
S-13     #pragma omp parallel firstprivate(B, C, D, E)
S-14     {
S-15         assert(sizeof(B) == sizeof(int (*)[n]));
S-16         assert(sizeof(C) == sizeof(int*));
S-17         assert(sizeof(D) == 4 * sizeof(int));
S-18         assert(sizeof(E) == n * n * sizeof(int));
S-19
S-20         /* Private B and C have values of original B and C. */
S-21         assert(&B[1][1] == &A[1][1]);
S-22         assert(&C[3] == &A[1][1]);
S-23         assert(D[1][1] == 4);
S-24         assert(E[1][1] == 4);
```

```
S-25     }  
S-26   }  
S-27  
S-28   int main() {  
S-29     f(2, A, A[0]);  
S-30     return 0;  
S-31   }
```

▲————— C / C++ —————▲

1 7.8 The lastprivate Clause

2 Correct execution sometimes depends on the value that the last iteration of a loop assigns to a
3 variable. Such programs must list all such variables in a **lastprivate** clause so that the values
4 of the variables are the same as when the loop is executed sequentially.

▼ C / C++ ▼

5 *Example lastprivate.1.c*

```
S-1 void lastpriv (int n, float *a, float *b)
S-2 {
S-3     int i;
S-4
S-5     #pragma omp parallel
S-6     {
S-7         #pragma omp for lastprivate(i)
S-8         for (i=0; i<n-1; i++)
S-9             a[i] = b[i] + b[i+1];
S-10    }
S-11
S-12    a[i]=b[i];      /* i == n-1 here */
S-13 }
```

▲ C / C++ ▲
▼ Fortran ▼

6 *Example lastprivate.1.f*

```
S-1      SUBROUTINE LASTPRIV(N, A, B)
S-2
S-3      INTEGER N
S-4      REAL A(*), B(*)
S-5      INTEGER I
S-6      !$OMP PARALLEL
S-7      !$OMP DO LASTPRIVATE(I)
S-8
S-9      DO I=1,N-1
S-10         A(I) = B(I) + B(I+1)
S-11      ENDDO
S-12
S-13      !$OMP END PARALLEL
S-14      A(I) = B(I)      ! I has the value of N here
S-15
S-16      END SUBROUTINE LASTPRIV
```

▲ Fortran ▲

1 7.9 The reduction Clause

2 The following example demonstrates the **reduction** clause ; note that some reductions can be
3 expressed in the loop in several ways, as shown for the **max** and **min** reductions below:

C / C++

4 *Example reduction.1.c*

```
S-1 #include <math.h>
S-2 void reduction1(float *x, int *y, int n)
S-3 {
S-4     int i, b, c;
S-5     float a, d;
S-6     a = 0.0;
S-7     b = 0;
S-8     c = y[0];
S-9     d = x[0];
S-10    #pragma omp parallel for private(i) shared(x, y, n) \
S-11                                   reduction(+:a) reduction(^:b) \
S-12                                   reduction(min:c) reduction(max:d)
S-13    for (i=0; i<n; i++) {
S-14        a += x[i];
S-15        b ^= y[i];
S-16        if (c > y[i]) c = y[i];
S-17        d = fmaxf(d, x[i]);
S-18    }
S-19 }
```

C / C++

Fortran

5 *Example reduction.1.f90*

```
S-1 SUBROUTINE REDUCTION1(A, B, C, D, X, Y, N)
S-2     REAL :: X(*), A, D
S-3     INTEGER :: Y(*), N, B, C
S-4     INTEGER :: I
S-5     A = 0
S-6     B = 0
S-7     C = Y(1)
S-8     D = X(1)
S-9     !$OMP PARALLEL DO PRIVATE(I) SHARED(X, Y, N) REDUCTION(+:A) &
S-10    !$OMP& REDUCTION(IEOR:B) REDUCTION(MIN:C) REDUCTION(MAX:D)
S-11     DO I=1,N
S-12         A = A + X(I)
S-13         B = IEOR(B, Y(I))
S-14         C = MIN(C, Y(I))
S-15         IF (D < X(I)) D = X(I)
```

```

S-16         END DO
S-17
S-18     END SUBROUTINE REDUCTION1

```

Fortran

1 A common implementation of the preceding example is to treat it as if it had been written as
 2 follows:

C / C++

3 *Example reduction.2.c*

```

S-1  #include <limits.h>
S-2  #include <math.h>
S-3  void reduction2(float *x, int *y, int n)
S-4  {
S-5      int i, b, b_p, c, c_p;
S-6      float a, a_p, d, d_p;
S-7      a = 0.0f;
S-8      b = 0;
S-9      c = y[0];
S-10     d = x[0];
S-11     #pragma omp parallel shared(a, b, c, d, x, y, n) \
S-12                          private(a_p, b_p, c_p, d_p)
S-13     {
S-14         a_p = 0.0f;
S-15         b_p = 0;
S-16         c_p = INT_MAX;
S-17         d_p = -HUGE_VALF;
S-18         #pragma omp for private(i)
S-19         for (i=0; i<n; i++) {
S-20             a_p += x[i];
S-21             b_p ^= y[i];
S-22             if (c_p > y[i]) c_p = y[i];
S-23             d_p = fmaxf(d_p, x[i]);
S-24         }
S-25         #pragma omp critical
S-26         {
S-27             a += a_p;
S-28             b ^= b_p;
S-29             if( c > c_p ) c = c_p;
S-30             d = fmaxf(d, d_p);
S-31         }
S-32     }
S-33 }

```

C / C++

Example reduction.2.f90

```

S-1      SUBROUTINE REDUCTION2(A, B, C, D, X, Y, N)
S-2      REAL :: X(*), A, D
S-3      INTEGER :: Y(*), N, B, C
S-4      REAL :: A_P, D_P
S-5      INTEGER :: I, B_P, C_P
S-6      A = 0
S-7      B = 0
S-8      C = Y(1)
S-9      D = X(1)
S-10     !$OMP PARALLEL SHARED(X, Y, A, B, C, D, N) &
S-11     !$OMP&          PRIVATE(A_P, B_P, C_P, D_P)
S-12     A_P = 0.0
S-13     B_P = 0
S-14     C_P = HUGE(C_P)
S-15     D_P = -HUGE(D_P)
S-16     !$OMP DO PRIVATE(I)
S-17     DO I=1,N
S-18     A_P = A_P + X(I)
S-19     B_P = IEOR(B_P, Y(I))
S-20     C_P = MIN(C_P, Y(I))
S-21     IF (D_P < X(I)) D_P = X(I)
S-22     END DO
S-23     !$OMP CRITICAL
S-24     A = A + A_P
S-25     B = IEOR(B, B_P)
S-26     C = MIN(C, C_P)
S-27     D = MAX(D, D_P)
S-28     !$OMP END CRITICAL
S-29     !$OMP END PARALLEL
S-30     END SUBROUTINE REDUCTION2

```

The following program is non-conforming because the reduction is on the *intrinsic procedure name* **MAX** but that name has been redefined to be the variable named **MAX**.

Example reduction.3.f90

```

S-1      PROGRAM REDUCTION_WRONG
S-2      MAX = HUGE(0)
S-3      M = 0
S-4
S-5      !$OMP PARALLEL DO REDUCTION(MAX: M)
S-6      ! MAX is no longer the intrinsic so this is non-conforming
S-7      DO I = 1, 100
S-8      CALL SUB(M, I)
S-9      END DO

```

```
S-10
S-11     END PROGRAM REDUCTION_WRONG
S-12
S-13     SUBROUTINE SUB(M, I)
S-14         M = MAX(M, I)
S-15     END SUBROUTINE SUB
```

The following conforming program performs the reduction using the *intrinsic procedure name* **MAX** even though the intrinsic **MAX** has been renamed to **REN**.

Example reduction.4.f90

```
S-1     MODULE M
S-2         INTRINSIC MAX
S-3     END MODULE M
S-4
S-5     PROGRAM REDUCTION3
S-6         USE M, REN => MAX
S-7         N = 0
S-8         !$OMP PARALLEL DO REDUCTION(REN: N)      ! still does MAX
S-9             DO I = 1, 100
S-10                 N = MAX(N, I)
S-11             END DO
S-12     END PROGRAM REDUCTION3
```

The following conforming program performs the reduction using *intrinsic procedure name* **MAX** even though the intrinsic **MAX** has been renamed to **MIN**.

Example reduction.5.f90

```
S-1     MODULE MOD
S-2         INTRINSIC MAX, MIN
S-3     END MODULE MOD
S-4
S-5     PROGRAM REDUCTION4
S-6         USE MOD, MIN=>MAX, MAX=>MIN
S-7         REAL :: R
S-8         R = -HUGE(0.0)
S-9
S-10        !$OMP PARALLEL DO REDUCTION(MIN: R)      ! still does MAX
S-11            DO I = 1, 1000
S-12                R = MIN(R, SIN(REAL(I)))
S-13            END DO
S-14            PRINT *, R
S-15    END PROGRAM REDUCTION4
```

Fortran

The following example is non-conforming because the initialization (**a = 0**) of the original list item **a** is not synchronized with the update of **a** as a result of the reduction computation in the **for** loop. Therefore, the example may print an incorrect value for **a**.

To avoid this problem, the initialization of the original list item **a** should complete before any update of **a** as a result of the **reduction** clause. This can be achieved by adding an explicit barrier after the assignment **a = 0**, or by enclosing the assignment **a = 0** in a **single** directive (which has an implied barrier), or by initializing **a** before the start of the **parallel** region.

C / C++

Example reduction.6.c

```
S-1  #include <stdio.h>
S-2
S-3  int main (void)
S-4  {
S-5      int a, i;
S-6
S-7      #pragma omp parallel shared(a) private(i)
S-8      {
S-9          #pragma omp master
S-10         a = 0;
S-11
S-12         // To avoid race conditions, add a barrier here.
S-13
S-14         #pragma omp for reduction(+:a)
S-15         for (i = 0; i < 10; i++) {
S-16             a += i;
S-17         }
S-18
S-19         #pragma omp single
S-20         printf ("Sum is %d\n", a);
S-21     }
S-22     return 0;
S-23 }
```

C / C++

Fortran

Example reduction.6.f

```
S-1      INTEGER A, I
S-2
S-3      !$OMP PARALLEL SHARED(A) PRIVATE(I)
S-4
S-5      !$OMP MASTER
S-6          A = 0
S-7      !$OMP END MASTER
S-8
S-9          ! To avoid race conditions, add a barrier here.
S-10
S-11      !$OMP DO REDUCTION(+:A)
S-12          DO I= 0, 9
S-13              A = A + I
S-14          END DO
S-15
S-16      !$OMP SINGLE
S-17          PRINT *, "Sum is ", A
S-18      !$OMP END SINGLE
S-19
S-20      !$OMP END PARALLEL
S-21          END
```

Fortran

The following example demonstrates the reduction of array *a*. In C/C++ this is illustrated by the explicit use of an array section *a[0:N]* in the **reduction** clause. The corresponding Fortran example uses array syntax supported in the base language. As of the OpenMP 4.5 specification the explicit use of array section in the **reduction** clause in Fortran is not permitted. But this oversight will be fixed in the next release of the specification.

C / C++

Example reduction.7.c

```
S-1      #include <stdio.h>
S-2
S-3      #define N 100
S-4      void init(int n, float (*b) [N]);
S-5
S-6      int main() {
S-7
S-8          int i, j;
S-9          float a[N], b[N] [N];
S-10
S-11          init(N, b);
```

```

S-12
S-13     for(i=0; i<N; i++) a[i]=0.0e0;
S-14
S-15     #pragma omp parallel for reduction(+:a[0:N]) private(j)
S-16     for(i=0; i<N; i++){
S-17         for(j=0; j<N; j++){
S-18             a[j] += b[i][j];
S-19         }
S-20     }
S-21     printf(" a[0] a[N-1]: %f %f\n", a[0], a[N-1]);
S-22
S-23     return 0;
S-24 }

```

▲ C / C++ ▲

▼ Fortran ▼

1 *Example reduction.7.f90*

```

S-1  program array_red
S-2
S-3      integer,parameter :: n=100
S-4      integer           :: j
S-5      real              :: a(n), b(n,n)
S-6
S-7      call init(n,b)
S-8
S-9      a(:) = 0.0e0
S-10
S-11      !$omp parallel do reduction(+:a)
S-12      do j = 1, n
S-13          a(:) = a(:) + b(:,j)
S-14      end do
S-15
S-16      print*, " a(1) a(n): ", a(1), a(n)
S-17
S-18  end program

```

▲ Fortran ▲

1 7.10 The copyin Clause

2 The **copyin** clause is used to initialize threadprivate data upon entry to a **parallel** region. The
3 value of the threadprivate variable in the master thread is copied to the threadprivate variable of
4 each other team member.

▼ C / C++ ▼

5 *Example copyin.1.c*

```
S-1  #include <stdlib.h>
S-2
S-3  float* work;
S-4  int size;
S-5  float tol;
S-6
S-7  #pragma omp threadprivate(work,size,tol)
S-8
S-9  void build()
S-10 {
S-11     int i;
S-12     work = (float*)malloc( sizeof(float)*size );
S-13     for( i = 0; i < size; ++i ) work[i] = tol;
S-14 }
S-15
S-16 void copyin_example( float t, int n )
S-17 {
S-18     tol = t;
S-19     size = n;
S-20     #pragma omp parallel copyin(tol,size)
S-21     {
S-22         build();
S-23     }
S-24 }
```

▲ C / C++ ▲

1

Example copyin.1.f

```

S-1      MODULE M
S-2          REAL, POINTER, SAVE :: WORK(:)
S-3          INTEGER :: SIZE
S-4          REAL :: TOL
S-5      !$OMP   THREADPRIVATE(WORK,SIZE,TOL)
S-6      END MODULE M
S-7
S-8      SUBROUTINE COPYIN_EXAMPLE( T, N )
S-9          USE M
S-10         REAL :: T
S-11         INTEGER :: N
S-12         TOL = T
S-13         SIZE = N
S-14     !$OMP   PARALLEL COPYIN(TOL,SIZE)
S-15         CALL BUILD
S-16     !$OMP   END PARALLEL
S-17     END SUBROUTINE COPYIN_EXAMPLE
S-18
S-19     SUBROUTINE BUILD
S-20         USE M
S-21         ALLOCATE(WORK(SIZE))
S-22         WORK = TOL
S-23     END SUBROUTINE BUILD

```

1 7.11 The copyprivate Clause

2 The **copyprivate** clause can be used to broadcast values acquired by a single thread directly to
3 all instances of the private variables in the other threads. In this example, if the routine is called
4 from the sequential part, its behavior is not affected by the presence of the directives. If it is called
5 from a **parallel** region, then the actual arguments with which **a** and **b** are associated must be
6 private.

7 The thread that executes the structured block associated with the **single** construct broadcasts the
8 values of the private variables **a**, **b**, **x**, and **y** from its implicit task's data environment to the data
9 environments of the other implicit tasks in the thread team. The broadcast completes before any of
10 the threads have left the barrier at the end of the construct.

▼ C / C++ ▼

11 *Example copyprivate.l.c*

```
S-1 #include <stdio.h>
S-2 float x, y;
S-3 #pragma omp threadprivate(x, y)
S-4
S-5 void init(float a, float b ) {
S-6     #pragma omp single copyprivate(a,b,x,y)
S-7     {
S-8         scanf("%f %f %f %f", &a, &b, &x, &y);
S-9     }
S-10 }
```

▲ C / C++ ▲
▼ Fortran ▼

12 *Example copyprivate.l.f*

```
S-1 SUBROUTINE INIT(A,B)
S-2 REAL A, B
S-3 COMMON /XY/ X,Y
S-4 !$OMP THREADPRIVATE (/XY/)
S-5
S-6 !$OMP SINGLE
S-7 READ (11) A,B,X,Y
S-8 !$OMP END SINGLE COPYPRIVATE (A,B,/XY/)
S-9
S-10 END SUBROUTINE INIT
```

▲ Fortran ▲

13 In this example, assume that the input must be performed by the master thread. Since the **master**
14 construct does not support the **copyprivate** clause, it cannot broadcast the input value that is
15 read. However, **copyprivate** is used to broadcast an address where the input value is stored.

1 *Example copyprivate.2.c*

```

S-1  #include <stdio.h>
S-2  #include <stdlib.h>
S-3
S-4  float read_next( ) {
S-5      float * tmp;
S-6      float return_val;
S-7
S-8      #pragma omp single copyprivate(tmp)
S-9      {
S-10         tmp = (float *) malloc(sizeof(float));
S-11     } /* copies the pointer only */
S-12
S-13
S-14     #pragma omp master
S-15     {
S-16         scanf("%f", tmp);
S-17     }
S-18
S-19     #pragma omp barrier
S-20     return_val = *tmp;
S-21     #pragma omp barrier
S-22
S-23     #pragma omp single nowait
S-24     {
S-25         free(tmp);
S-26     }
S-27
S-28     return return_val;
S-29 }

```

2 *Example copyprivate.2.f*

```

S-1      REAL FUNCTION READ_NEXT()
S-2      REAL, POINTER :: TMP
S-3
S-4      !$OMP SINGLE
S-5          ALLOCATE (TMP)
S-6      !$OMP END SINGLE COPYPRIVATE (TMP) ! copies the pointer only
S-7
S-8      !$OMP MASTER
S-9          READ (11) TMP
S-10     !$OMP END MASTER

```

```

S-11
S-12  !$OMP  BARRIER
S-13      READ_NEXT = TMP
S-14  !$OMP  BARRIER
S-15
S-16  !$OMP  SINGLE
S-17      DEALLOCATE (TMP)
S-18  !$OMP  END SINGLE NOWAIT
S-19  END FUNCTION READ_NEXT

```

Fortran

1 Suppose that the number of lock variables required within a **parallel** region cannot easily be
 2 determined prior to entering it. The **copyprivate** clause can be used to provide access to shared
 3 lock variables that are allocated within that **parallel** region.

C / C++

4 *Example copyprivate.3.c*

```

S-1  #include <stdio.h>
S-2  #include <stdlib.h>
S-3  #include <omp.h>
S-4
S-5  omp_lock_t *new_lock()
S-6  {
S-7      omp_lock_t *lock_ptr;
S-8
S-9      #pragma omp single copyprivate(lock_ptr)
S-10     {
S-11         lock_ptr = (omp_lock_t *) malloc(sizeof(omp_lock_t));
S-12         omp_init_lock( lock_ptr );
S-13     }
S-14
S-15     return lock_ptr;
S-16 }

```

C / C++

Example copyprivate.3.f

```

S-1      FUNCTION NEW_LOCK()
S-2      USE OMP_LIB      ! or INCLUDE "omp_lib.h"
S-3      INTEGER(OMP_LOCK_KIND), POINTER :: NEW_LOCK
S-4
S-5      !$OMP    SINGLE
S-6          ALLOCATE (NEW_LOCK)
S-7          CALL OMP_INIT_LOCK(NEW_LOCK)
S-8      !$OMP    END SINGLE COPYPRIVATE(NEW_LOCK)
S-9      END FUNCTION NEW_LOCK

```

Note that the effect of the **copyprivate** clause on a variable with the **allocatable** attribute is different than on a variable with the **pointer** attribute. The value of **A** is copied (as if by intrinsic assignment) and the pointer **B** is copied (as if by pointer assignment) to the corresponding list items in the other implicit tasks belonging to the **parallel** region.

Example copyprivate.4.f

```

S-1      SUBROUTINE S(N)
S-2      INTEGER N
S-3
S-4      REAL, DIMENSION(:), ALLOCATABLE :: A
S-5      REAL, DIMENSION(:), POINTER :: B
S-6
S-7      ALLOCATE (A(N))
S-8      !$OMP    SINGLE
S-9          ALLOCATE (B(N))
S-10         READ (11) A,B
S-11      !$OMP    END SINGLE COPYPRIVATE(A,B)
S-12          ! Variable A is private and is
S-13          ! assigned the same value in each thread
S-14          ! Variable B is shared
S-15
S-16      !$OMP    BARRIER
S-17      !$OMP    SINGLE
S-18          DEALLOCATE (B)
S-19      !$OMP    END SINGLE NOWAIT
S-20      END SUBROUTINE S

```

1 7.12 C++ Reference in Data-Sharing Clauses

C++

C++ reference types are allowed in data-sharing attribute clauses as of OpenMP 4.5, except for the **threadprivate**, **copyin** and **copyprivate** clauses. (See the Data-Sharing Attribute Clauses Section of the 4.5 OpenMP specification.) When a variable with C++ reference type is privatized, the object the reference refers to is privatized in addition to the reference itself. The following example shows the use of reference types in data-sharing clauses in the usual way. Additionally it shows how the data-sharing of formal arguments with a C++ reference type on an orphaned task generating construct is determined implicitly. (See the Data-sharing Attribute Rules for Variables Referenced in a Construct Section of the 4.5 OpenMP specification.)

Example cpp_reference.1.cpp

```
S-1
S-2 void task_body (int &x);
S-3 void gen_task (int &x) { // on orphaned task construct reference argument
S-4     #pragma omp task // x is implicitly determined firstprivate(x)
S-5     task_body (x);
S-6 }
S-7 void test (int &y, int &z) {
S-8     #pragma omp parallel private(y)
S-9     {
S-10         y = z + 2;
S-11         gen_task (y); // no matter if the argument is determined private
S-12         gen_task (z); // or shared in the enclosing context.
S-13
S-14         y++;          // each thread has its own int object y refers to
S-15         gen_task (y);
S-16     }
S-17 }
S-18
```

C++

1 7.13 Fortran ASSOCIATE Construct

Fortran

2 The following is an invalid example of specifying an associate name on a data-sharing attribute
3 clause. The constraint in the Data Sharing Attribute Rules section in the OpenMP 4.0 API
4 Specifications states that an associate name preserves the association with the selector established
5 at the **ASSOCIATE** statement. The associate name *b* is associated with the shared variable *a*. With
6 the predetermined data-sharing attribute rule, the associate name *b* is not allowed to be specified on
7 the **private** clause.

8 *Example associate.1.f*

```
S-1      program example
S-2      real :: a, c
S-3      associate (b => a)
S-4      !$omp parallel private(b, c)          ! invalid to privatize b
S-5      c = 2.0*b
S-6      !$omp end parallel
S-7      end associate
S-8      end program
```

9 In next example, within the **parallel** construct, the association name *thread_id* is associated
10 with the private copy of *i*. The print statement should output the unique thread number.

11 *Example associate.2.f*

```
S-1      program example
S-2      use omp_lib
S-3      integer i
S-4      !$omp parallel private(i)
S-5      i = omp_get_thread_num()
S-6      associate(thread_id => i)
S-7      print *, thread_id          ! print private i value
S-8      end associate
S-9      !$omp end parallel
S-10     end program
```

12 The following example illustrates the effect of specifying a selector name on a data-sharing
13 attribute clause. The associate name *u* is associated with *v* and the variable *v* is specified on the
14 **private** clause of the **parallel** construct. The construct association is established prior to the
15 **parallel** region. The association between *u* and the original *v* is retained (see the Data Sharing
16 Attribute Rules section in the OpenMP 4.0 API Specifications). Inside the **parallel** region, *v*
17 has the value of -1 and *u* has the value of the original *v*.

18 *Example associate.3.f90*


```
S-1  program example
S-2      integer :: v
S-3      v = 15
S-4      associate(u => v)
S-5      !$omp parallel private(v)
S-6          v = -1
S-7          print *, v                ! private v=-1
S-8          print *, u                ! original v=15
S-9      !$omp end parallel
S-10     end associate
S-11     end program
```

Fortran

Memory Model

In this chapter, examples illustrate race conditions on access to variables with shared data-sharing attributes. A race condition can exist when two or more threads are involved in accessing a variable in which not all of the accesses are reads; that is, a WaR, RaW or WaW condition exists (R=read, a=after, W=write). A RaR does not produce a race condition. Ensuring thread execution order at the processor level is not enough to avoid race conditions, because the local storage at the processor level (registers, caches, etc.) must be synchronized so that a consistent view of the variable in the memory hierarchy can be seen by the threads accessing the variable.

OpenMP provides a shared-memory model which allows all threads access to *memory* (shared data). Each thread also has exclusive access to *threadprivate memory* (private data). A private variable referenced in an OpenMP directive's structured block is a new version of the original variable (with the same name) for each task (or SIMD lane) within the code block. A private variable is initially undefined (except for variables in **firstprivate** and **linear** clauses), and the original variable value is unaltered by assignments to the private variable, (except for **reduction**, **lastprivate** and **linear** clauses).

Private variables in an outer **parallel** region can be shared by implicit tasks of an inner **parallel** region (with a **share** clause on the inner **parallel** directive). Likewise, a private variable may be shared in the region of an explicit **task** (through a **shared** clause).

The **flush** directive forces a consistent view of local variables of the thread executing the **flush**. When a list is supplied on the directive, only the items (variables) in the list are guaranteed to be flushed.

Implied flushes exist at prescribed locations of certain constructs. For the complete list of these locations and associated constructs, please refer to the *flush Construct* section of the OpenMP Specifications document.

Examples 1-3 show the difficulty of synchronizing threads through **flush** and **atomic** directives.

1 8.1 The OpenMP Memory Model

2 In the following example, at Print 1, the value of x could be either 2 or 5, depending on the timing
3 of the threads, and the implementation of the assignment to x . There are two reasons that the value
4 at Print 1 might not be 5. First, Print 1 might be executed before the assignment to x is executed.
5 Second, even if Print 1 is executed after the assignment, the value 5 is not guaranteed to be seen by
6 thread 1 because a flush may not have been executed by thread 0 since the assignment.

7 The barrier after Print 1 contains implicit flushes on all threads, as well as a thread synchronization,
8 so the programmer is guaranteed that the value 5 will be printed by both Print 2 and Print 3.

▼ C / C++ ▼

9 *Example mem_model.1.c*

```
S-1 #include <stdio.h>
S-2 #include <omp.h>
S-3
S-4 int main(){
S-5     int x;
S-6
S-7     x = 2;
S-8     #pragma omp parallel num_threads(2) shared(x)
S-9     {
S-10
S-11         if (omp_get_thread_num() == 0) {
S-12             x = 5;
S-13         } else {
S-14             /* Print 1: the following read of x has a race */
S-15             printf("1: Thread# %d: x = %d\n", omp_get_thread_num(), x );
S-16         }
S-17
S-18         #pragma omp barrier
S-19
S-20         if (omp_get_thread_num() == 0) {
S-21             /* Print 2 */
S-22             printf("2: Thread# %d: x = %d\n", omp_get_thread_num(), x );
S-23         } else {
S-24             /* Print 3 */
S-25             printf("3: Thread# %d: x = %d\n", omp_get_thread_num(), x );
S-26         }
S-27     }
S-28     return 0;
S-29 }
```

▲ C / C++ ▲

Example mem_model.1.f90

```

S-1  PROGRAM MEMMODEL
S-2      INCLUDE "omp_lib.h"          ! or USE OMP_LIB
S-3      INTEGER X
S-4
S-5      X = 2
S-6  !$OMP PARALLEL NUM_THREADS(2) SHARED(X)
S-7
S-8      IF (OMP_GET_THREAD_NUM() .EQ. 0) THEN
S-9          X = 5
S-10     ELSE
S-11         ! PRINT 1: The following read of x has a race
S-12         PRINT *, "1: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", X
S-13     ENDIF
S-14
S-15  !$OMP BARRIER
S-16
S-17     IF (OMP_GET_THREAD_NUM() .EQ. 0) THEN
S-18         ! PRINT 2
S-19         PRINT *, "2: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", X
S-20     ELSE
S-21         ! PRINT 3
S-22         PRINT *, "3: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", X
S-23     ENDIF
S-24
S-25  !$OMP END PARALLEL
S-26
S-27  END PROGRAM MEMMODEL

```

The following example demonstrates why synchronization is difficult to perform correctly through variables. The value of flag is undefined in both prints on thread 1 and the value of data is only well-defined in the second print.

1

Example mem_model.2.c

```

S-1  #include <omp.h>
S-2  #include <stdio.h>
S-3  int main()
S-4  {
S-5      int data;
S-6      int flag=0;
S-7      #pragma omp parallel num_threads(2)
S-8      {
S-9          if (omp_get_thread_num()==0)
S-10         {
S-11             /* Write to the data buffer that will be
S-12             read by thread */
S-13             data = 42;
S-14             /* Flush data to thread 1 and strictly order
S-15             the write to data
S-16             relative to the write to the flag */
S-17             #pragma omp flush(flag, data)
S-18             /* Set flag to release thread 1 */
S-19             flag = 1;
S-20             /* Flush flag to ensure that thread 1 sees
S-21             the change */
S-22             #pragma omp flush(flag)
S-23         }
S-24         else if(omp_get_thread_num()==1)
S-25         {
S-26             /* Loop until we see the update to the flag */
S-27             #pragma omp flush(flag, data)
S-28             while (flag < 1)
S-29             {
S-30                 #pragma omp flush(flag, data)
S-31             }
S-32             /* Values of flag and data are undefined */
S-33             printf("flag=%d data=%d\n", flag, data);
S-34             #pragma omp flush(flag, data)
S-35             /* Values data will be 42, value of flag
S-36             still undefined */
S-37             printf("flag=%d data=%d\n", flag, data);
S-38         }
S-39     }
S-40     return 0;
S-41 }

```

Example *mem_model.2.f*

```

S-1      PROGRAM EXAMPLE
S-2      INCLUDE "omp_lib.h" ! or USE OMP_LIB
S-3      INTEGER DATA
S-4      INTEGER FLAG
S-5
S-6      FLAG = 0
S-7      !$OMP PARALLEL NUM_THREADS(2)
S-8          IF(OMP_GET_THREAD_NUM() .EQ. 0) THEN
S-9              ! Write to the data buffer that will be read by thread 1
S-10             DATA = 42
S-11             ! Flush DATA to thread 1 and strictly order the write to DATA
S-12             ! relative to the write to the FLAG
S-13             !$OMP FLUSH(FLAG, DATA)
S-14             ! Set FLAG to release thread 1
S-15             FLAG = 1;
S-16             ! Flush FLAG to ensure that thread 1 sees the change */
S-17             !$OMP FLUSH(FLAG)
S-18         ELSE IF(OMP_GET_THREAD_NUM() .EQ. 1) THEN
S-19             ! Loop until we see the update to the FLAG
S-20             !$OMP FLUSH(FLAG, DATA)
S-21             DO WHILE(FLAG .LT. 1)
S-22                 !$OMP FLUSH(FLAG, DATA)
S-23             ENDDO
S-24
S-25             ! Values of FLAG and DATA are undefined
S-26             PRINT *, 'FLAG=', FLAG, ' DATA=', DATA
S-27             !$OMP FLUSH(FLAG, DATA)
S-28
S-29             !Values DATA will be 42, value of FLAG still undefined */
S-30             PRINT *, 'FLAG=', FLAG, ' DATA=', DATA
S-31         ENDIF
S-32     !$OMP END PARALLEL
S-33 END

```

The next example demonstrates why synchronization is difficult to perform correctly through variables. Because the *write(1)-flush(1)-flush(2)-read(2)* sequence cannot be guaranteed in the example, the statements on thread 0 and thread 1 may execute in either order.

1

Example mem_model.3.c

```

S-1  #include <omp.h>
S-2  #include <stdio.h>
S-3  int main()
S-4  {
S-5      int flag=0;
S-6
S-7      #pragma omp parallel num_threads(3)
S-8      {
S-9          if(omp_get_thread_num()==0)
S-10         {
S-11             /* Set flag to release thread 1 */
S-12             #pragma omp atomic update
S-13             flag++;
S-14             /* Flush of flag is implied by the atomic directive */
S-15         }
S-16         else if(omp_get_thread_num()==1)
S-17         {
S-18             /* Loop until we see that flag reaches 1*/
S-19             #pragma omp flush(flag)
S-20             while(flag < 1)
S-21             {
S-22                 #pragma omp flush(flag)
S-23             }
S-24             printf("Thread 1 awoken\n");
S-25
S-26             /* Set flag to release thread 2 */
S-27             #pragma omp atomic update
S-28             flag++;
S-29             /* Flush of flag is implied by the atomic directive */
S-30         }
S-31         else if(omp_get_thread_num()==2)
S-32         {
S-33             /* Loop until we see that flag reaches 2 */
S-34             #pragma omp flush(flag)
S-35             while(flag < 2)
S-36             {
S-37                 #pragma omp flush(flag)
S-38             }
S-39             printf("Thread 2 awoken\n");
S-40         }
S-41     }
S-42     return 0;
S-43 }

```

1

Example mem_model.3.f

```

S-1      PROGRAM EXAMPLE
S-2      INCLUDE "omp_lib.h" ! or USE OMP_LIB
S-3      INTEGER FLAG
S-4
S-5      FLAG = 0
S-6      !$OMP PARALLEL NUM_THREADS(3)
S-7          IF (OMP_GET_THREAD_NUM() .EQ. 0) THEN
S-8              ! Set flag to release thread 1
S-9              !$OMP ATOMIC UPDATE
S-10                 FLAG = FLAG + 1
S-11                 !Flush of FLAG is implied by the atomic directive
S-12             ELSE IF (OMP_GET_THREAD_NUM() .EQ. 1) THEN
S-13                 ! Loop until we see that FLAG reaches 1
S-14                 !$OMP FLUSH(FLAG, DATA)
S-15                 DO WHILE (FLAG .LT. 1)
S-16                     !$OMP FLUSH(FLAG, DATA)
S-17                 ENDDO
S-18
S-19                 PRINT *, 'Thread 1 awoken'
S-20
S-21                 ! Set FLAG to release thread 2
S-22                 !$OMP ATOMIC UPDATE
S-23                 FLAG = FLAG + 1
S-24                 !Flush of FLAG is implied by the atomic directive
S-25             ELSE IF (OMP_GET_THREAD_NUM() .EQ. 2) THEN
S-26                 ! Loop until we see that FLAG reaches 2
S-27                 !$OMP FLUSH(FLAG, DATA)
S-28                 DO WHILE (FLAG .LT. 2)
S-29                     !$OMP FLUSH(FLAG, DATA)
S-30                 ENDDO
S-31
S-32                 PRINT *, 'Thread 2 awoken'
S-33             ENDIF
S-34         !$OMP END PARALLEL
S-35     END

```


1 8.2 Race Conditions Caused by Implied Copies 2 of Shared Variables in Fortran

Fortran

3 The following example contains a race condition, because the shared variable, which is an array
4 section, is passed as an actual argument to a routine that has an assumed-size array as its dummy
5 argument. The subroutine call passing an array section argument may cause the compiler to copy
6 the argument into a temporary location prior to the call and copy from the temporary location into
7 the original variable when the subroutine returns. This copying would cause races in the
8 **parallel** region.

9 *Example fort_race.f90*

```
S-1 SUBROUTINE SHARED_RACE
S-2
S-3     INCLUDE "omp_lib.h"      ! or USE OMP_LIB
S-4
S-5     REAL A(20)
S-6     INTEGER MYTHREAD
S-7
S-8     !$OMP PARALLEL SHARED(A) PRIVATE(MYTHREAD)
S-9
S-10    MYTHREAD = OMP_GET_THREAD_NUM()
S-11    IF (MYTHREAD .EQ. 0) THEN
S-12        CALL SUB(A(1:10)) ! compiler may introduce writes to A(6:10)
S-13    ELSE
S-14        A(6:10) = 12
S-15    ENDIF
S-16
S-17    !$OMP END PARALLEL
S-18
S-19 END SUBROUTINE SHARED_RACE
S-20
S-21 SUBROUTINE SUB(X)
S-22     REAL X(*)
S-23     X(1:5) = 4
S-24 END SUBROUTINE SUB
```

Fortran

Program Control

Some specific and elementary concepts of controlling program execution are illustrated in the examples of this chapter. Control can be directly managed with conditional control code (`ifdef`'s with the `_OPENMP` macro, and the Fortran sentinel (`!$`) for conditionally compiling). The `if` clause on some constructs can direct the runtime to ignore or alter the behavior of the construct. Of course, the base-language `if` statements can be used to control the "execution" of stand-alone directives (such as `flush`, `barrier`, `taskwait`, and `taskyield`). However, the directives must appear in a block structure, and not as a substatement as shown in examples 1 and 2 of this chapter.

CANCELLATION

Cancellation (termination) of the normal sequence of execution for the threads in an OpenMP region can be accomplished with the `cancel` construct. The construct uses a *construct-type-clause* to set the region-type to activate for the cancellation. That is, inclusion of one of the *construct-type-clause* names `parallel`, `for`, `do`, `sections` or `taskgroup` on the directive line activates the corresponding region. The `cancel` construct is activated by the first encountering thread, and it continues execution at the end of the named region. The `cancel` construct is also a cancellation point for any other thread of the team to also continue execution at the end of the named region.

Also, once the specified region has been activated for cancellation any thread that encounters a `cancellation point` construct with the same named region (*construct-type-clause*), continues execution at the end of the region.

For an activated `cancel taskgroup` construct, the tasks that belong to the taskgroup set of the innermost enclosing taskgroup region will be canceled.

A task that encounters the `cancel taskgroup` construct continues execution at the end of its task region. Any task of the taskgroup that has already begun execution will run to completion, unless it encounters a `cancellation point`; tasks that have not begun execution "may" be discarded as completed tasks.

CONTROL VARIABLES

Internal control variables (ICV) are used by implementations to hold values which control the execution of OpenMP regions. Control (and hence the ICVs) may be set as implementation defaults, or set and adjusted through environment variables, clauses, and API functions. Many of the ICV control values are accessible through API function calls. Also, initial ICV values are reported by the runtime if the **OMP_DISPLAY_ENV** environment variable has been set to **TRUE**.

NESTED CONSTRUCTS

Certain combinations of nested constructs are permitted, giving rise to a *combined* construct consisting of two or more constructs. These can be used when the two (or several) constructs would be used immediately in succession (closely nested). A combined construct can use the clauses of the component constructs without restrictions. A *composite* construct is a combined construct which has one or more clauses with (an often obviously) modified or restricted meaning, relative to when the constructs are uncombined.

Certain nestings are forbidden, and often the reasoning is obvious. Worksharing constructs cannot be nested, and the **barrier** construct cannot be nested inside a worksharing construct, or a **critical** construct. Also, **target** constructs cannot be nested.

The **parallel** construct can be nested, as well as the **task** construct. The parallel execution in the nested **parallel** construct(s) is control by the **OMP_NESTED** and **OMP_MAX_ACTIVE_LEVELS** environment variables, and the **omp_set_nested()** and **omp_set_max_active_levels()** functions.

More details on nesting can be found in the *Nesting of Regions* of the *Directives* chapter in the OpenMP Specifications document.

1 9.1 Conditional Compilation

C / C++

2 The following example illustrates the use of conditional compilation using the OpenMP macro
3 `_OPENMP`. With OpenMP compilation, the `_OPENMP` macro becomes defined.

4 *Example cond_comp.1.c*

```
S-1 #include <stdio.h>
S-2
S-3 int main()
S-4 {
S-5
S-6 # ifdef _OPENMP
S-7     printf("Compiled by an OpenMP-compliant implementation.\n");
S-8 # endif
S-9
S-10     return 0;
S-11 }
```

C / C++

Fortran

5 The following example illustrates the use of the conditional compilation sentinel. With OpenMP
6 compilation, the conditional compilation sentinel `!$` is recognized and treated as two spaces. In
7 fixed form source, statements guarded by the sentinel must start after column 6.

8 *Example cond_comp.1.f*

```
S-1         PROGRAM EXAMPLE
S-2
S-3 C234567890
S-4 !$      PRINT *, "Compiled by an OpenMP-compliant implementation."
S-5
S-6         END PROGRAM EXAMPLE
```

Fortran

1 9.2 Internal Control Variables (ICVs)

2 According to Section 2.3 of the OpenMP 4.0 specification, an OpenMP implementation must act as
3 if there are ICVs that control the behavior of the program. This example illustrates two ICVs,
4 *nthreads-var* and *max-active-levels-var*. The *nthreads-var* ICV controls the number of threads
5 requested for encountered parallel regions; there is one copy of this ICV per task. The
6 *max-active-levels-var* ICV controls the maximum number of nested active parallel regions; there is
7 one copy of this ICV for the whole program.

8 In the following example, the *nest-var*, *max-active-levels-var*, *dyn-var*, and *nthreads-var* ICVs are
9 modified through calls to the runtime library routines **omp_set_nested**,
10 **omp_set_max_active_levels**, **omp_set_dynamic**, and **omp_set_num_threads**
11 respectively. These ICVs affect the operation of **parallel** regions. Each implicit task generated
12 by a **parallel** region has its own copy of the *nest-var*, *dyn-var*, and *nthreads-var* ICVs.

13 In the following example, the new value of *nthreads-var* applies only to the implicit tasks that
14 execute the call to **omp_set_num_threads**. There is one copy of the *max-active-levels-var*
15 ICV for the whole program and its value is the same for all tasks. This example assumes that nested
16 parallelism is supported.

17 The outer **parallel** region creates a team of two threads; each of the threads will execute one of
18 the two implicit tasks generated by the outer **parallel** region.

19 Each implicit task generated by the outer **parallel** region calls **omp_set_num_threads(3)**,
20 assigning the value 3 to its respective copy of *nthreads-var*. Then each implicit task encounters an
21 inner **parallel** region that creates a team of three threads; each of the threads will execute one of
22 the three implicit tasks generated by that inner **parallel** region.

23 Since the outer **parallel** region is executed by 2 threads, and the inner by 3, there will be a total
24 of 6 implicit tasks generated by the two inner **parallel** regions.

25 Each implicit task generated by an inner **parallel** region will execute the call to
26 **omp_set_num_threads(4)**, assigning the value 4 to its respective copy of *nthreads-var*.

27 The print statement in the outer **parallel** region is executed by only one of the threads in the
28 team. So it will be executed only once.

29 The print statement in an inner **parallel** region is also executed by only one of the threads in the
30 team. Since we have a total of two inner **parallel** regions, the print statement will be executed
31 twice – once per inner **parallel** region.

▼ C / C++ ▼

32 *Example icv.1.c*

```

S-1  #include <stdio.h>
S-2  #include <omp.h>
S-3
S-4  int main (void)
S-5  {
S-6      omp_set_nested(1);
S-7      omp_set_max_active_levels(8);
S-8      omp_set_dynamic(0);
S-9      omp_set_num_threads(2);
S-10     #pragma omp parallel
S-11     {
S-12         omp_set_num_threads(3);
S-13
S-14         #pragma omp parallel
S-15         {
S-16             omp_set_num_threads(4);
S-17             #pragma omp single
S-18             {
S-19                 /*
S-20                 * The following should print:
S-21                 * Inner: max_act_lev=8, num_thds=3, max_thds=4
S-22                 * Inner: max_act_lev=8, num_thds=3, max_thds=4
S-23                 */
S-24                 printf ("Inner: max_act_lev=%d, num_thds=%d, max_thds=%d\n",
S-25                     omp_get_max_active_levels(), omp_get_num_threads(),
S-26                     omp_get_max_threads());
S-27             }
S-28         }
S-29
S-30         #pragma omp barrier
S-31         #pragma omp single
S-32         {
S-33             /*
S-34             * The following should print:
S-35             * Outer: max_act_lev=8, num_thds=2, max_thds=3
S-36             */
S-37             printf ("Outer: max_act_lev=%d, num_thds=%d, max_thds=%d\n",
S-38                 omp_get_max_active_levels(), omp_get_num_threads(),
S-39                 omp_get_max_threads());
S-40         }
S-41     }
S-42     return 0;
S-43 }

```

C / C++

1

Example icv.1.f

```

S-1      program icv
S-2      use omp_lib
S-3
S-4      call omp_set_nested(.true.)
S-5      call omp_set_max_active_levels(8)
S-6      call omp_set_dynamic(.false.)
S-7      call omp_set_num_threads(2)
S-8
S-9      !$omp parallel
S-10     call omp_set_num_threads(3)
S-11
S-12     !$omp parallel
S-13     call omp_set_num_threads(4)
S-14     !$omp single
S-15     !      The following should print:
S-16     !      Inner: max_act_lev= 8 , num_thds= 3 , max_thds= 4
S-17     !      Inner: max_act_lev= 8 , num_thds= 3 , max_thds= 4
S-18     print *, "Inner: max_act_lev=", omp_get_max_active_levels(),
S-19     &         ", num_thds=", omp_get_num_threads(),
S-20     &         ", max_thds=", omp_get_max_threads()
S-21     !$omp end single
S-22     !$omp end parallel
S-23
S-24     !$omp barrier
S-25     !$omp single
S-26     !      The following should print:
S-27     !      Outer: max_act_lev= 8 , num_thds= 2 , max_thds= 3
S-28     print *, "Outer: max_act_lev=", omp_get_max_active_levels(),
S-29     &         ", num_thds=", omp_get_num_threads(),
S-30     &         ", max_thds=", omp_get_max_threads()
S-31     !$omp end single
S-32     !$omp end parallel
S-33     end

```

9.3 Placement of flush, barrier, taskwait and taskyield Directives

The following example is non-conforming, because the **flush**, **barrier**, **taskwait**, and **taskyield** directives are stand-alone directives and cannot be the immediate substatement of an **if** statement.

C / C++

Example standalone.1.c

```
S-1
S-2 void standalone_wrong()
S-3 {
S-4     int a = 1;
S-5
S-6         if (a != 0)
S-7             #pragma omp flush(a)
S-8             /* incorrect as flush cannot be immediate substatement
S-9                of if statement */
S-10
S-11             if (a != 0)
S-12                 #pragma omp barrier
S-13                 /* incorrect as barrier cannot be immediate substatement
S-14                    of if statement */
S-15
S-16             if (a!=0)
S-17                 #pragma omp taskyield
S-18                 /* incorrect as taskyield cannot be immediate substatement of if statement
S-19                    */
S-20
S-21             if (a != 0)
S-22                 #pragma omp taskwait
S-23                 /* incorrect as taskwait cannot be immediate substatement
S-24                    of if statement */
S-25
S-26 }
```

C / C++

The following example is non-conforming, because the **flush**, **barrier**, **taskwait**, and **taskyield** directives are stand-alone directives and cannot be the action statement of an **if** statement or a labeled branch target.

1

Example standalone.1.f90

```

S-1  SUBROUTINE STANDALONE_WRONG()
S-2      INTEGER  A
S-3      A = 1
S-4      ! the FLUSH directive must not be the action statement
S-5      ! in an IF statement
S-6      IF (A .NE. 0) !$OMP FLUSH(A)
S-7
S-8      ! the BARRIER directive must not be the action statement
S-9      ! in an IF statement
S-10     IF (A .NE. 0) !$OMP BARRIER
S-11
S-12     ! the TASKWAIT directive must not be the action statement
S-13     ! in an IF statement
S-14     IF (A .NE. 0) !$OMP TASKWAIT
S-15
S-16     ! the TASKYIELD directive must not be the action statement
S-17     ! in an IF statement
S-18     IF (A .NE. 0) !$OMP TASKYIELD
S-19
S-20     GOTO 100
S-21
S-22     ! the FLUSH directive must not be a labeled branch target
S-23     ! statement
S-24     100 !$OMP FLUSH(A)
S-25     GOTO 200
S-26
S-27     ! the BARRIER directive must not be a labeled branch target
S-28     ! statement
S-29     200 !$OMP BARRIER
S-30     GOTO 300
S-31
S-32     ! the TASKWAIT directive must not be a labeled branch target
S-33     ! statement
S-34     300 !$OMP TASKWAIT
S-35     GOTO 400
S-36
S-37     ! the TASKYIELD directive must not be a labeled branch target
S-38     ! statement
S-39     400 !$OMP TASKYIELD
S-40
S-41     END SUBROUTINE

```

Fortran

The following version of the above example is conforming because the **flush**, **barrier**, **taskwait**, and **taskyield** directives are enclosed in a compound statement.

C / C++

Example standalone.2.c

```
S-1 void standalone_ok()
S-2 {
S-3     int a = 1;
S-4
S-5     #pragma omp parallel
S-6     {
S-7         if (a != 0) {
S-8             #pragma omp flush(a)
S-9             }
S-10        if (a != 0) {
S-11            #pragma omp barrier
S-12            }
S-13        if (a != 0) {
S-14            #pragma omp taskwait
S-15            }
S-16            if (a != 0) {
S-17                #pragma omp taskyield
S-18                }
S-19        }
S-20    }
```

C / C++

The following example is conforming because the **flush**, **barrier**, **taskwait**, and **taskyield** directives are enclosed in an **if** construct or follow the labeled branch target.

1

Example standalone.2.f90

```

S-1  SUBROUTINE STANDALONE_OK()
S-2      INTEGER A
S-3      A = 1
S-4      IF (A .NE. 0) THEN
S-5          !$OMP FLUSH(A)
S-6      ENDIF
S-7      IF (A .NE. 0) THEN
S-8          !$OMP BARRIER
S-9      ENDIF
S-10     IF (A .NE. 0) THEN
S-11         !$OMP TASKWAIT
S-12     ENDIF
S-13     IF (A .NE. 0) THEN
S-14         !$OMP TASKYIELD
S-15     ENDIF
S-16     GOTO 100
S-17     100 CONTINUE
S-18     !$OMP FLUSH(A)
S-19     GOTO 200
S-20     200 CONTINUE
S-21     !$OMP BARRIER
S-22     GOTO 300
S-23     300 CONTINUE
S-24     !$OMP TASKWAIT
S-25     GOTO 400
S-26     400 CONTINUE
S-27     !$OMP TASKYIELD
S-28 END SUBROUTINE

```

9.4 Cancellation Constructs

The following example shows how the **cancel** directive can be used to terminate an OpenMP region. Although the **cancel** construct terminates the OpenMP worksharing region, programmers must still track the exception through the pointer `ex` and issue a cancellation for the **parallel** region if an exception has been raised. The master thread checks the exception pointer to make sure that the exception is properly handled in the sequential part. If cancellation of the **parallel** region has been requested, some threads might have executed `phase_1()`. However, it is guaranteed that none of the threads executed `phase_2()`.

C++

Example cancellation.1.cpp

```
S-1  #include <iostream>
S-2  #include <exception>
S-3  #include <cstdint>
S-4
S-5  #define N 10000
S-6
S-7  extern void causes_an_exception();
S-8  extern void phase_1();
S-9  extern void phase_2();
S-10
S-11 void example() {
S-12     std::exception *ex = NULL;
S-13     #pragma omp parallel shared(ex)
S-14     {
S-15         #pragma omp for
S-16         for (int i = 0; i < N; i++) {
S-17             // no 'if' that prevents compiler optimizations
S-18             try {
S-19                 causes_an_exception();
S-20             }
S-21             catch (std::exception *e) {
S-22                 // still must remember exception for later handling
S-23             #pragma omp atomic write
S-24                 ex = e;
S-25             // cancel worksharing construct
S-26             #pragma omp cancel for
S-27             }
S-28         }
S-29         // if an exception has been raised, cancel parallel region
S-30         if (ex) {
S-31             #pragma omp cancel parallel
S-32         }
S-33         phase_1();
```

```

S-34  #pragma omp barrier
S-35      phase_2();
S-36      }
S-37      // continue here if an exception has been thrown in the worksharing loop
S-38      if (ex) {
S-39          // handle exception stored in ex
S-40      }
S-41  }

```

▲ C++ ▲

1 The following example illustrates the use of the **cancel** construct in error handling. If there is an
2 error condition from the **allocate** statement, the cancellation is activated. The encountering
3 thread sets the shared variable **err** and other threads of the binding thread set proceed to the end of
4 the worksharing construct after the cancellation has been activated.

▼ Fortran ▼

5 *Example cancellation.1.f90*

```

S-1  subroutine example(n, dim)
S-2      integer, intent(in) :: n, dim(n)
S-3      integer :: i, s, err
S-4      real, allocatable :: B(:)
S-5      err = 0
S-6      !$omp parallel shared(err)
S-7      ! ...
S-8      !$omp do private(s, B)
S-9          do i=1, n
S-10         !$omp cancellation point do
S-11             allocate(B(dim(i)), stat=s)
S-12             if (s .gt. 0) then
S-13                 !$omp atomic write
S-14                 err = s
S-15                 !$omp cancel do
S-16             endif
S-17             ! ...
S-18         ! deallocate private array B
S-19             if (allocated(B)) then
S-20                 deallocate(B)
S-21             endif
S-22         enddo
S-23     !$omp end parallel
S-24 end subroutine

```

The following example shows how to cancel a parallel search on a binary tree as soon as the search value has been detected. The code creates a task to descend into the child nodes of the current tree node. If the search value has been found, the code remembers the tree node with the found value through an **atomic** write to the result variable and then cancels execution of all search tasks. The function **search_tree_parallel** groups all search tasks into a single task group to control the effect of the **cancel taskgroup** directive. The *level* argument is used to create undeferred tasks after the first ten levels of the tree.

Example cancellation.2.c

```

S-1  #include <stddef.h>
S-2
S-3  typedef struct binary_tree_s {
S-4      int value;
S-5      struct binary_tree_s *left, *right;
S-6  } binary_tree_t;
S-7
S-8  binary_tree_t *search_tree(binary_tree_t *tree, int value, int level) {
S-9      binary_tree_t *found = NULL;
S-10     if (tree) {
S-11         if (tree->value == value) {
S-12             found = tree;
S-13         }
S-14         else {
S-15             #pragma omp task shared(found) if(level < 10)
S-16             {
S-17                 binary_tree_t *found_left = NULL;
S-18                 found_left = search_tree(tree->left, value, level + 1);
S-19                 if (found_left) {
S-20                     #pragma omp atomic write
S-21                     found = found_left;
S-22                     #pragma omp cancel taskgroup
S-23                     }
S-24                 }
S-25             #pragma omp task shared(found) if(level < 10)
S-26             {
S-27                 binary_tree_t *found_right = NULL;
S-28                 found_right = search_tree(tree->right, value, level + 1);
S-29                 if (found_right) {
S-30                     #pragma omp atomic write
S-31                     found = found_right;
S-32                     #pragma omp cancel taskgroup
S-33                     }
S-34             }

```

```

S-35  #pragma omp taskwait
S-36      }
S-37      }
S-38      return found;
S-39  }
S-40  binary_tree_t *search_tree_parallel(binary_tree_t *tree, int value) {
S-41      binary_tree_t *found = NULL;
S-42      #pragma omp parallel shared(found, tree, value)
S-43      {
S-44          #pragma omp master
S-45          {
S-46              #pragma omp taskgroup
S-47              {
S-48                  found = search_tree(tree, value, 0);
S-49              }
S-50          }
S-51      }
S-52      return found;
S-53  }

```

C / C++

1 The following is the equivalent parallel search example in Fortran.

Fortran

2 *Example cancellation.2.f90*

```

S-1  module parallel_search
S-2      type binary_tree
S-3          integer :: value
S-4          type(binary_tree), pointer :: right
S-5          type(binary_tree), pointer :: left
S-6      end type
S-7
S-8  contains
S-9      recursive subroutine search_tree(tree, value, level, found)
S-10         type(binary_tree), intent(in), pointer :: tree
S-11         integer, intent(in) :: value, level
S-12         type(binary_tree), pointer :: found
S-13         type(binary_tree), pointer :: found_left => NULL(), found_right => NULL()
S-14
S-15         if (associated(tree)) then
S-16             if (tree%value .eq. value) then
S-17                 found => tree
S-18             else
S-19                 !$omp task shared(found) if(level<10)
S-20                 call search_tree(tree%left, value, level+1, found_left)
S-21                 if (associated(found_left)) then

```

```

S-22      !$omp critical
S-23          found => found_left
S-24      !$omp end critical
S-25
S-26      !$omp cancel taskgroup
S-27          endif
S-28      !$omp end task
S-29
S-30      !$omp task shared(found) if(level<10)
S-31          call search_tree(tree%right, value, level+1, found_right)
S-32          if (associated(found_right)) then
S-33      !$omp critical
S-34          found => found_right
S-35      !$omp end critical
S-36
S-37      !$omp cancel taskgroup
S-38          endif
S-39      !$omp end task
S-40
S-41      !$omp taskwait
S-42          endif
S-43      endif
S-44      end subroutine
S-45
S-46      subroutine search_tree_parallel(tree, value, found)
S-47          type(binary_tree), intent(in), pointer :: tree
S-48          integer, intent(in) :: value
S-49          type(binary_tree), pointer :: found
S-50
S-51          found => NULL()
S-52      !$omp parallel shared(found, tree, value)
S-53      !$omp master
S-54      !$omp taskgroup
S-55          call search_tree(tree, value, 0, found)
S-56      !$omp end taskgroup
S-57      !$omp end master
S-58      !$omp end parallel
S-59      end subroutine
S-60
S-61      end module parallel_search

```

▲ ————— Fortran ————— ▲

1 9.5 Nested Loop Constructs

2 The following example of loop construct nesting is conforming because the inner and outer loop
3 regions bind to different **parallel** regions:

▼ C / C++ ▼

4 *Example nested_loop.l.c*

```
S-1 void work(int i, int j) {}
S-2
S-3 void good_nesting(int n)
S-4 {
S-5     int i, j;
S-6     #pragma omp parallel default(shared)
S-7     {
S-8         #pragma omp for
S-9         for (i=0; i<n; i++) {
S-10             #pragma omp parallel shared(i, n)
S-11             {
S-12                 #pragma omp for
S-13                 for (j=0; j < n; j++)
S-14                     work(i, j);
S-15             }
S-16         }
S-17     }
S-18 }
```

▲ C / C++ ▲
▼ Fortran ▼

5 *Example nested_loop.l.f*

```
S-1 SUBROUTINE WORK(I, J)
S-2     INTEGER I, J
S-3     END SUBROUTINE WORK
S-4
S-5 SUBROUTINE GOOD_NESTING(N)
S-6     INTEGER N
S-7
S-8     INTEGER I
S-9     !$OMP PARALLEL DEFAULT(SHARED)
S-10     !$OMP DO
S-11         DO I = 1, N
S-12             !$OMP PARALLEL SHARED(I,N)
S-13             !$OMP DO
S-14                 DO J = 1, N
S-15                     CALL WORK(I, J)
S-16                 END DO
```

```

S-17      !$OMP      END PARALLEL
S-18      END DO
S-19      !$OMP      END PARALLEL
S-20      END SUBROUTINE GOOD_NESTING

```

Fortran

1 The following variation of the preceding example is also conforming:

C / C++

2 *Example nested_loop.2.c*

```

S-1      void work(int i, int j) {}
S-2
S-3
S-4      void work1(int i, int n)
S-5      {
S-6          int j;
S-7          #pragma omp parallel default(shared)
S-8          {
S-9              #pragma omp for
S-10             for (j=0; j<n; j++)
S-11                 work(i, j);
S-12         }
S-13     }
S-14
S-15
S-16     void good_nesting2(int n)
S-17     {
S-18         int i;
S-19         #pragma omp parallel default(shared)
S-20         {
S-21             #pragma omp for
S-22             for (i=0; i<n; i++)
S-23                 work1(i, n);
S-24         }
S-25     }

```

C / C++

1

Example nested_loop.2.f

```

S-1      SUBROUTINE WORK(I, J)
S-2      INTEGER I, J
S-3      END SUBROUTINE WORK
S-4
S-5      SUBROUTINE WORK1(I, N)
S-6      INTEGER J
S-7      !$OMP PARALLEL DEFAULT(SHARED)
S-8      !$OMP DO
S-9          DO J = 1, N
S-10         CALL WORK(I, J)
S-11     END DO
S-12     !$OMP END PARALLEL
S-13     END SUBROUTINE WORK1
S-14
S-15     SUBROUTINE GOOD_NESTING2(N)
S-16     INTEGER N
S-17     !$OMP PARALLEL DEFAULT(SHARED)
S-18     !$OMP DO
S-19         DO I = 1, N
S-20             CALL WORK1(I, N)
S-21         END DO
S-22     !$OMP END PARALLEL
S-23     END SUBROUTINE GOOD_NESTING2

```

1 9.6 Restrictions on Nesting of Regions

2 The examples in this section illustrate the region nesting rules.

3 The following example is non-conforming because the inner and outer loop regions are closely
4 nested:

▼ C / C++ ▼

5 *Example nesting_restrict.1.c*

```
S-1 void work(int i, int j) {}  
S-2 void wrong1(int n)  
S-3 {  
S-4     #pragma omp parallel default(shared)  
S-5     {  
S-6         int i, j;  
S-7         #pragma omp for  
S-8         for (i=0; i<n; i++) {  
S-9             /* incorrect nesting of loop regions */  
S-10            #pragma omp for  
S-11                for (j=0; j<n; j++)  
S-12                    work(i, j);  
S-13            }  
S-14        }  
S-15    }
```

▲ C / C++ ▲
▼ Fortran ▼

6 *Example nesting_restrict.1.f*

```
S-1     SUBROUTINE WORK(I, J)  
S-2     INTEGER I, J  
S-3     END SUBROUTINE WORK  
S-4     SUBROUTINE WRONG1(N)  
S-5     INTEGER N  
S-6         INTEGER I, J  
S-7     !$OMP PARALLEL DEFAULT(SHARED)  
S-8     !$OMP DO  
S-9         DO I = 1, N  
S-10    !$OMP DO ! incorrect nesting of loop regions  
S-11        DO J = 1, N  
S-12            CALL WORK(I, J)  
S-13        END DO  
S-14    END DO  
S-15    !$OMP END PARALLEL  
S-16    END SUBROUTINE WRONG1
```

Fortran

1 The following orphaned version of the preceding example is also non-conforming:

C / C++

2 *Example nesting_restrict.2.c*

```
S-1 void work(int i, int j) {}
S-2 void work1(int i, int n)
S-3 {
S-4     int j;
S-5     /* incorrect nesting of loop regions */
S-6     #pragma omp for
S-7         for (j=0; j<n; j++)
S-8         work(i, j);
S-9 }
S-10
S-11 void wrong2(int n)
S-12 {
S-13     #pragma omp parallel default(shared)
S-14     {
S-15         int i;
S-16         #pragma omp for
S-17             for (i=0; i<n; i++)
S-18             work1(i, n);
S-19     }
S-20 }
```

C / C++

Fortran

3 *Example nesting_restrict.2.f*

```
S-1      SUBROUTINE WORK1(I,N)
S-2      INTEGER I, N
S-3      INTEGER J
S-4      !$OMP DO      ! incorrect nesting of loop regions
S-5          DO J = 1, N
S-6              CALL WORK(I,J)
S-7          END DO
S-8      END SUBROUTINE WORK1
S-9      SUBROUTINE WRONG2(N)
S-10     INTEGER N
S-11     INTEGER I
S-12     !$OMP PARALLEL DEFAULT(SHARED)
S-13     !$OMP DO
S-14         DO I = 1, N
S-15             CALL WORK1(I,N)
```

```

S-16         END DO
S-17 !$OMP   END PARALLEL
S-18         END SUBROUTINE WRONG2

```

Fortran

The following example is non-conforming because the loop and **single** regions are closely nested:

C / C++

Example nesting_restrict.3.c

```

S-1 void work(int i, int j) {}
S-2 void wrong3(int n)
S-3 {
S-4     #pragma omp parallel default(shared)
S-5     {
S-6         int i;
S-7         #pragma omp for
S-8         for (i=0; i<n; i++) {
S-9             /* incorrect nesting of regions */
S-10            #pragma omp single
S-11            work(i, 0);
S-12        }
S-13    }
S-14 }

```

C / C++

Fortran

Example nesting_restrict.3.f

```

S-1         SUBROUTINE WRONG3(N)
S-2         INTEGER N
S-3
S-4         INTEGER I
S-5 !$OMP   PARALLEL DEFAULT(SHARED)
S-6 !$OMP   DO
S-7         DO I = 1, N
S-8 !$OMP   SINGLE                                ! incorrect nesting of regions
S-9         CALL WORK(I, 1)
S-10 !$OMP   END SINGLE
S-11         END DO
S-12 !$OMP   END PARALLEL
S-13         END SUBROUTINE WRONG3

```

Fortran

The following example is non-conforming because a **barrier** region cannot be closely nested inside a loop region:

1

Example nesting_restrict.4.c

```

S-1 void work(int i, int j) {}
S-2 void wrong4(int n)
S-3 {
S-4
S-5     #pragma omp parallel default(shared)
S-6     {
S-7         int i;
S-8         #pragma omp for
S-9         for (i=0; i<n; i++) {
S-10             work(i, 0);
S-11 /* incorrect nesting of barrier region in a loop region */
S-12         #pragma omp barrier
S-13         work(i, 1);
S-14     }
S-15 }
S-16 }

```

2

Example nesting_restrict.4.f

```

S-1     SUBROUTINE WRONG4(N)
S-2     INTEGER N
S-3
S-4     INTEGER I
S-5     !$OMP PARALLEL DEFAULT(SHARED)
S-6     !$OMP DO
S-7         DO I = 1, N
S-8             CALL WORK(I, 1)
S-9     ! incorrect nesting of barrier region in a loop region
S-10    !$OMP BARRIER
S-11        CALL WORK(I, 2)
S-12    END DO
S-13    !$OMP END PARALLEL
S-14    END SUBROUTINE WRONG4

```

3

4

5

The following example is non-conforming because the **barrier** region cannot be closely nested inside the **critical** region. If this were permitted, it would result in deadlock due to the fact that only one thread at a time can enter the **critical** region:

Example *nesting_restrict.5.c*

```

S-1 void work(int i, int j) {}
S-2 void wrong5(int n)
S-3 {
S-4     #pragma omp parallel
S-5     {
S-6         #pragma omp critical
S-7         {
S-8             work(n, 0);
S-9         /* incorrect nesting of barrier region in a critical region */
S-10         #pragma omp barrier
S-11         work(n, 1);
S-12     }
S-13 }
S-14 }

```

Example *nesting_restrict.5.f*

```

S-1      SUBROUTINE WRONG5(N)
S-2      INTEGER N
S-3
S-4      !$OMP    PARALLEL DEFAULT(SHARED)
S-5      !$OMP    CRITICAL
S-6          CALL WORK(N,1)
S-7      ! incorrect nesting of barrier region in a critical region
S-8      !$OMP    BARRIER
S-9          CALL WORK(N,2)
S-10     !$OMP    END CRITICAL
S-11     !$OMP    END PARALLEL
S-12     END SUBROUTINE WRONG5

```

The following example is non-conforming because the **barrier** region cannot be closely nested inside the **single** region. If this were permitted, it would result in deadlock due to the fact that only one thread executes the **single** region:

1

Example nesting_restrict.6.c

```

S-1 void work(int i, int j) {}
S-2 void wrong6(int n)
S-3 {
S-4     #pragma omp parallel
S-5     {
S-6         #pragma omp single
S-7         {
S-8             work(n, 0);
S-9         /* incorrect nesting of barrier region in a single region */
S-10         #pragma omp barrier
S-11         work(n, 1);
S-12     }
S-13 }
S-14 }

```

2

Example nesting_restrict.6.f

```

S-1      SUBROUTINE WRONG6(N)
S-2      INTEGER N
S-3
S-4      !$OMP    PARALLEL DEFAULT(SHARED)
S-5      !$OMP    SINGLE
S-6          CALL WORK(N,1)
S-7      ! incorrect nesting of barrier region in a single region
S-8      !$OMP    BARRIER
S-9          CALL WORK(N,2)
S-10     !$OMP    END SINGLE
S-11     !$OMP    END PARALLEL
S-12     END SUBROUTINE WRONG6

```

Document Revision History

A.1 Changes from 4.0.2 to 4.5.0

- Reorganized into chapters of major topics
- Included file extensions in example labels to indicate source type
- Applied the explicit **map(tofrom)** for scalar variables in a number of examples to comply with the change of the default behavior for scalar variables from **map(tofrom)** to **firstprivate** in the 4.5 specification
- Added the following new examples:
 - **linear** clause in loop constructs (Section 1.8 on page 22)
 - task priority (Section 3.2 on page 71)
 - **taskloop** construct (Section 3.6 on page 85)
 - *directive-name* modifier in multiple **if** clauses on a combined construct (Section 4.1.5 on page 93)
 - unstructured data mapping (Section 4.3 on page 108)
 - **link** clause for **declare target** directive (Section 4.5.5 on page 123)
 - asynchronous target execution with **nowait** clause (Section 4.7 on page 135)
 - device memory routines and device pointers (Section 4.9.4 on page 152)
 - doacross loop nest (Section 6.8 on page 194)
 - locks with hints (Section 6.9 on page 200)
 - C/C++ array reduction (Section 7.9 on page 233)
 - C++ reference types in data sharing clauses (Section 7.12 on page 246)

1 **A.2 Changes from 4.0.1 to 4.0.2**

- 2 • Names of examples were changed from numbers to mnemonics
- 3 • Added SIMD examples (Section [5.1](#) on page [155](#))
- 4 • Applied miscellaneous fixes in several source codes
- 5 • Added the revision history

6 **A.3 Changes from 4.0 to 4.0.1**

7 Added the following new examples:

- 8 • the **proc_bind** clause (Section [2.1](#) on page [42](#))
- 9 • the **taskgroup** construct (Section [3.4](#) on page [80](#))

10 **A.4 Changes from 3.1 to 4.0**

11 Beginning with OpenMP 4.0, examples were placed in a separate document from the specification
12 document.

13 Version 4.0 added the following new examples:

- 14 • task dependences (Section [3.3](#) on page [73](#))
- 15 • **target** construct (Section [4.1](#) on page [88](#))
- 16 • **target data** construct (Section [4.2](#) on page [96](#))
- 17 • **target update** construct (Section [4.4](#) on page [111](#))
- 18 • **declare target** construct (Section [4.5](#) on page [115](#))
- 19 • **teams** constructs (Section [4.6](#) on page [126](#))
- 20 • asynchronous execution of a **target** region using tasks (Section [4.7.1](#) on page [135](#))
- 21 • array sections in device constructs (Section [4.8](#) on page [144](#))
- 22 • device runtime routines (Section [4.9](#) on page [148](#))

- 1 • Fortran ASSOCIATE construct (Section [7.13](#) on page [247](#))
- 2 • cancellation constructs (Section [9.4](#) on page [267](#))