Directives (continued)

The `sections` construct contains a set of structured blocks to be distributed among and executed by the executing team of threads.

```c
#pragma omp sections [clause[1], clause[2], ...]
{...
  #pragma omp section
  structured-block
  ...
  #pragma omp section
  structured-block
{...}
```

```c
#pragma omp end sections [nowait] clause:
private(ot)
firstprivate(ot)
lastprivate(ot)
reduction([operator][intrinsic_procedure_name]: ot)
```

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.

```c
#pragma omp single [clause[1], clause[2], ...]
{...
  #pragma omp single
  structured-block
  ...
  #pragma omp single
  end_clause[1], end_clause[2], ...
{...}
```

The `workshare` construct divides the execution of the enclosed structured block into separate units of work.

```c
#pragma omp workshare [structured-block]
{...
  #pragma omp workshare
  structured-block
{...}
```

The combined parallel worksharing constructs are a shortcut for specifying a parallel construct containing one work-sharing construct and no other statements.

```c
#pragma omp do [clause[1], clause[2], ...]
do-loop
{...
  #pragma omp do
  structured-block
{...}
```

The barrier construct specifies an explicit barrier at the point at which the corresponding construct appears.

```c
#pragma omp barrier
```

The taskwait construct specifies a wait on the completion of child tasks generated since the beginning of the current task.

```c
#pragma omp taskwait
{...
  #pragma omp taskwait
{...}
```

The `atomic` construct ensures that a specific storage location is updated atomically, rather than exposing it to the possibility of multiple simultaneous writing threads.

```c
#pragma omp atomic
{...
  #pragma omp atomic
{...}
```

Clauses

Not all of the clauses are valid on all directives. The set of clauses that is valid on a particular directive is described with the directive. Most of the clauses accept a comma-separated list of list items. All list items appearing in a clause must be visible.

Data Sharing Attribute Clauses

Data-sharing attribute clauses apply only to variables whose names are visible in the construct on which the clause appears. Controls the default data-sharing attributes of variables that are referenced in a parallel or task construct.

```c
default[private][firstprivate][shared][none]
```

Declared or more list items to be shared by tasks generated by a parallel or task construct.

```c
private[0]
```

Declared or more list items to be private to a task.

```c
firstprivate[ot]
```

Declared 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.
Clauses (continued)

lastprivate(list) Declares one or more list items to be private to an implicit task, and causes the corresponding original item to be updated after the end of the region.

reduction(operator intrinsic procedure name list) Declares accumulation into the list items using the indicated associative operator. Accumulation occurs into a private copy for each list item which is then combined with the original item.

Data Copying Clauses

These clauses support the copying of data values from private or thread-private variables on one implicit task or thread to the corresponding variables on other implicit tasks or threads in the team.

copyin(list) Copies the value of the master thread's threadprivate variable to the threadprivate variable of each other member of the team executing the parallel region.

copyprivate(list) Broadcasts a value from the data environment of one implicit task to the data environments of the other implicit tasks belonging to the parallel region.

Runtime Library Routines

Execution environment routines affect and monitor threads, processors, and the parallel environment. Lock routines support synchronization with OpenMP locks. Timing routines support a portable wall clock timer. Prototypes for the runtime library routines appear in the include file "omp.h" and the Fortran module "omp.f".

Execution Environment Routines

subroutine omp_get_max_threads() integer num_threads" Affects the number of threads that could be used to form a new team using a "parallel" construct without a "num_threads" clause.

integer omp_get_thread_num() Returns the ID of the encountering thread where ID ranges from zero to the size of the team minus 1.

integer omp_get_num_procs() Returns the number of processors available to the program.

Runtime Library Routines (continued)

logical function omp_in_parallel() Returns true if the call to the routine is enclosed by an active parallel region; otherwise, it returns false.

subroutine omp_set_dynamic(dynamic_threads) Enables/denies dynamic adjustment of number of threads available.

logical function omp_get_dynamic() Returns value of dyn-var internal control variable (ICV), determining if dynamic adjustment of number of threads is enabled or disabled.

subroutine omp_set_nested(nestlevel) Allows or disables nested parallelism, by setting the nest-var ICV.

logical function omp_get_nested() Returns the value of the nest-var ICV, which determines if parallelism is enabled or disabled.

subroutine omp_set_schedule(kind modifier) Limits the number of nested active parallel regions, by setting the max-active-levels var ICV.

integer function omp_get_max_threads() Returns the maximum number of nested active parallel regions.

integer function omp_get_thread_limit() Returns the number of threads in the current team.

integer function omp_get_max_threads() Returns the maximum number of threads that could be used to form a new team using a "parallel" construct without a "num_threads" clause.

integer function omp_get_thread_num() Returns the ID of the encountering thread where ID ranges from zero to the size of the team minus 1.

integer function omp_get_num_procs() Returns the number of processors available to the program.

Routine Library Routines (continued)

Lock Routines

subroutine omp_init destroying set(unset) [nest_] lock integer(kind omp_lock_kind var) These routines initialize, initialize, set, or unset a (nested) OpenMP lock.

logical function omp_lock [nest_] lock integer(kind omp_lock_kind var) These routines attempt to set an OpenMP lock but do not suspend execution of the task executing the routine.

Timing Routines

omp_get_wtime() Returns the elapsed wall clock time in seconds.

omp_get_wtick() Returns the precision of the timer used by omp_get_wtime.

Environment Variables

Environment variable names are upper case, and the values assigned to them are case insensitive and may have leading and trailing white space.

OMP_SCHEDULE [type ] [chunk] Sets the run-sched ICV for the runtime schedule type and chunk size. Valid OpenMP schedule types are static, dynamic, guided, or auto. Chunk is a positive integer.

OMP_NUM_THREADS Sets the number of ICV for number of threads for parallel regions.

OMP_MAX_DYNAMIC Sets the max dynamic ICV for the dynamic adjustment of threads to use for parallel regions. Valid values for dynamic are true or false.

OMP_NESTED Sets ICV to enable or to disable nested parallelism. Valid values for nested are true or false.

OMP_SCHEDULE size Sets block-size ICV that specifies size of skip for threads created by the OpenMP implementation. Valid values for size (a positive integer) are size, stride, size, size, size. If units B, M, G are not specified, size is measured in kilobytes (K).

OMP_WAIT_POLICY policy Sets the wait-policy ICV that controls the desired behavior of waiting threads. Valid values for policy are active (waiting threads consume processor cycles while waiting) and passive.

OMP_MAX_ACTIVE_LEVELS levels Limits the maximum number of ICV that controls the maximum number of nested active parallel regions.

OMP_THREAD_LIMIT sets specifies the ICV that controls the maximum number of threads participating in the OpenMP program.