Summary of OpenMP 3.0 C/C++ Syntax

Download the full OpenMP API Specification at www.openmp.org.

Directives

An OpenMP executable directive applies to the succeeding structured block or an OpenMP Construct. A “structured block” is a single statement or a compound statement with a single entry at the top and a single exit at the bottom.

The `parallel` construct forms a team of threads and starts parallel execution.

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

```
clause:  if(scalar-expression)
         num_threads(integer-expression)
         default(shared | none)
         private(list)
         firstprivate(list)
         shared(list)
         copyin(list)
         reduction(operator: list)
```

The loop construct specifies that the iterations of loops will be distributed among and executed by the encountering team of threads.

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

```
clause:  private(list)
         firstprivate(list)
         lastprivate(list)
         reduction(operator: list)
         schedule(kind[, chunk_size])
         collapse(n)
         ordered
         nowait
```

The most common form of the for loop is shown below.

```
for(var = lb;
var relational-op b;
var += incr)
```

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

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

(See applicable clauses on next page.)

Directives (continued)

```
clause:  private(list)
         firstprivate(list)
         lastprivate(list)
         reduction(operator: list)
         nowait
```

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.

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

```
clause:  private(list)
         firstprivate(list)
         copyprivate(list)
         nowait
```

The combined parallel worksharing constructs are a shortcut for specifying a parallel construct containing one worksharing construct and no other statements. Permitted clauses are the union of the clauses allowed for the `parallel` and worksharing contructs.

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

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

The `task` construct defines an explicit task. The data environment of the task is created according to the data-sharing attribute clauses on the task construct and any defaults that apply.

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

```
clause:  if(scalar-expression)
         untied
         default(shared | none)
         private(list)
         firstprivate(list)
         shared(list)
```

The `master` construct specifies a structured block that is executed by the master thread of the team. There is no implied barrier either on entry to, or exit from, the master construct.

```
#pragma omp master new-line
        structured-block
```


Directives (continued)

The `critical` construct restricts execution of the associated structured block to a single thread at a time.

```
#pragma omp critical [(name)] new-line
        structured-block
```

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

```
#pragma omp barrier new-line
```

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

```
#pragma omp taskwait newline
```

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

```
#pragma omp atomic new-line
        expression-stmt
```

expression-stmt: one of the following forms:

```
x binop = expr
x++
++x
x--
--x
```

The `flush` construct executes the OpenMP flush operation, which makes a thread’s temporary view of memory consistent with memory, and enforces an order on the memory operations of the variables.

```
#pragma omp flush [(list)] new-line
```

The `ordered` construct specifies a structured block in a loop region that will be executed in the order of the loop iterations. This sequentializes and orders the code within an ordered region while allowing code outside the region to run in parallel.

```
#pragma omp ordered new-line
        structured-block
```

The `threadprivate` directive specifies that variables are replicated, with each thread having its own copy.

```
#pragma omp threadprivate(list) new-line
```


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.

`default(shared|none);`
Controls the default data-sharing attributes of variables that are referenced in a `parallel` or `task` construct.

`shared(`*list*`);`
Declares one or more list items to be shared by tasks generated by a `parallel` or `task` construct.

`private(`*list*`);`
Declares one or more list items to be private to a task.

`firstprivate(`*list*`);`
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.

`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*`:`*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 are defined in the file “omp.h”.

Execution Environment Routines

`void omp_set_num_threads(int` *num_threads*`);`
Affects the number of threads used for subsequent **`parallel`** regions that do not specify a **`num_threads`** clause.

`int omp_get_num_threads(void);`
Returns the number of threads in the current team.

`int omp_get_max_threads(void);`
Returns maximum number of threads that could be used to form a new team using a “parallel” construct without a “num_threads” clause.

`int omp_get_thread_num(void);`
Returns the ID of the encountering thread where ID ranges from zero to the size of the team minus 1.

`int omp_get_num_procs(void);`
Returns the number of processors available to the program.

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

`void omp_set_dynamic(int` *dynamic_threads*`);`
Enables or disables dynamic adjustment of the number of threads available.

`int omp_get_dynamic(void);`
Returns the value of the *dyn-var* internal control variable (ICV), determining whether dynamic adjustment of the number of threads is enabled or disabled.

`void omp_set_nested(int` *nested*`);`
Enables or disables nested parallelism, by setting the *nest-var* ICV.

`int omp_get_nested(void);`
Returns the value of the *nest-var* ICV, which determines if nested parallelism is enabled or disabled.

`void omp_set_schedule(omp_sched_t` *kind*`, int` *modifier*`);`
Affects the schedule that is applied when **`runtime`** is used as schedule kind, by setting the value of the *run-sched-var* ICV.

```
void omp_get_schedule(omp_sched_t *kind,
        int *modifier);
```
Returns the schedule applied when **`runtime`** schedule is used.

Runtime Library Routines (continued)

`int omp_get_thread_limit(void)`
Returns the maximum number of OpenMP threads available to the program.

`void omp_set_max_active_levels(int` *max_levels*`);`
Limits the number of nested active **`parallel`** regions, by setting the *max-active-levels-var* ICV.

`int omp_get_max_active_levels(void);`
Returns the value of the *max-activelevels-var ICV*, which determines the maximum number of nested active **`parallel`** regions.

`int omp_get_level(void);`
Returns the number of nested **`parallel`** regions enclosing the task that contains the call.

`int omp_get_ancestor_thread_num(int` *level*`);`
Returns, for a given nested level of the current thread, the thread number of the ancestor or the current thread.

`int omp_get_team_size(int` *level*`);`
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.

`int omp_get_active_level(void);`
Returns the number of nested, active **`parallel`** regions enclosing the task that contains the call.

Lock Routines

`void omp_init_lock(omp_lock_t *`*lock*`);`
`void omp_init_nest_lock(omp_nest_lock_t *`*lock*`);`
These routines initialize an OpenMP lock.

`void omp_destroy_lock(omp_lock_t *`*lock*`);`
`void omp_destroy_nest_lock(omp_nest_lock_t *`*lock*`);`
These routines ensure that the OpenMP lock is uninitialized.

`void omp_set_lock(omp_lock_t *`*lock*`);`
`void omp_set_nest_lock(omp_nest_lock_t *`*lock*`);`
These routines provide a means of setting an OpenMP lock.

`void omp_unset_lock(omp_lock_t *`*lock*`);`
`void omp_unset_nest_lock(omp_nest_lock_t *`*lock*`);`
These routines provide a means of setting an OpenMP lock.

`int omp_test_lock(omp_lock_t *`*lock*`);`
`int omp_test_nest_lock(omp_nest_lock_t *`*lock*`);`
These routines attempt to set an OpenMP lock but do not suspend execution of the task executing the routine.

Runtime Library Routines (continued)

Timing Routines

`double omp_get_wtime(void);`
Returns elapsed wall clock time in seconds.

`double omp_get_wtick(void);`
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-var* 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` *num*
Sets the *nthreads-var* ICV for the number of threads to use for **`parallel`** regions.

`OMP_DYNAMIC` *dynamic*
Sets the *dyn-var* ICV *for* the dynamic adjustment of threads to use for **`parallel`** regions. Valid values for *dynamic* are **`true`** or **`false`**.

`OMP_NESTED` *nested*
Sets the *nest-var* ICV to enable or to disable nested parallelism. Valid values for *nested* are true or false.

`OMP_STACKSIZE` *size*
Sets the *stacksize-var* ICV that specifies the size of the stack for threads created by the OpenMP implementation. Valid values for *size* (a positive integer) are *size*, *size*`B`, *size*`K`, *size*`M`, *size*`G`. If units `B`, `K`, `M` or `G` are not specified, size is measured in kilobytes (`K`).

`OMP_WAIT_POLICY` *policy*
Sets the *wait-policy-var* 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*
Sets the *max-active-levels-var* ICV that controls the maximum number of nested active **`parallel`** regions.

`OMP_THREAD_LIMIT` *limit*
Sets the *thread-limit-var* ICV that controls the maximum number of threads participating in the OpenMP program.

Details

Operators legally allowed in a reduction

Operator	Initialization value
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

Schedule types for the loop construct

static Iterations are divided into chunks of size *chunk_size*, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number.

dynamic Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed.

guided Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned. The chunk sizes start large and shrink to the indicated *chunk_size* as chunks are scheduled.

auto The decision regarding scheduling is delegated to the compiler and/or runtime system.

runtime The schedule and chunk size are taken from the run-sched-var ICV.

Rev 1108-001