OpenMP C/C++ Version 1.0: Syntax-Summary

OpenMP is an industry-standard API for programming shared memory computers. It is based on a fork/join programming model in which a program with a single thread of execution (the master thread) spawns a team of threads to carry out work concurrently.

This note is a brief summary of the OpenMP C/C++ version 1.0 Application Program Interface. To learn more about OpenMP and how to use it, consult the OpenMP web site at www.openmp.org.

Directive format

```
#pragma omp directive-name [clause[ clause]…]
```

The directive applies to at most one succeeding statement which must be a structured block. A structured block is a block of one or more statements with a single point of entry at the top and a single exit at the bottom. Branches into or out of a structure block are not permitted. It is allowed to have an `exit()` statement within a structured block.

Conditional compilation with the OpenMP macro

Conditional compilation is specified with the standard C/C++ preprocessor and the `_OPENMP` macro:

```
#ifdef _OPENMP
      any legal C/C++ constructs
#endif
```

The macro must not be the subject of a `#define` or an `#undef`.

Parallel region construct

```
#pragma omp parallel [clause[ clause] …]
      structured-block
```

Where *clause* is:	`if (scalar-expression)`
	`private(list)`
	`firstprivate(list)`
	`default (shared | none)`
	`shared(list)`
	`copyin(list)`
	`reduction(operator: list)`

Usage Note:

When the `if-clause` is present, the code within the parallel region is only executed with multiple threads if the `scalar-expression` evaluates to a non-zero value. If it evaluates to `zero` the code within the parallel region is serialized; i.e. it is executed by a team of size one.

Restrictions

- At most one `if clause` can appear on the directive.
- It is unspecified whether any side-effects inside the `if expression` occur.
- A throw in C++ from a parallel region must not cross a structured block, and it must be caught by the same thread that threw the exception.
- There is an implied barrier at the end of a parallel region.

Work-sharing Constructs

Work-sharing constructs and the `barrier` directive must be encountered in the same order by all threads in a team. They must be encountered by all threads in a team or none at all.

There is an implied barrier and the end of a work-sharing construct unless a `nowait` clause is specified in which case the threads immediately continue execution.

for construct

```
#pragma omp for [clause[ clause] …]
      for-loop
```

Where *clause* is:	`private(list)`
	`firstprivate(list)`
	`reduction(operator: list)`
	`ordered`
	`schedule(kind[, chunk_size])`
	`nowait`
	`lastprivate(list)`

Usage Note:

The `for` loop must have the standard form

```
      for(init-expr; var logical-op b; incr-expr)
```

where

- `init-expr` is a simple assignment to an integer type.
- `var` is a signed integer variable. It is implicitly made private if it has not been explicitly made private by the programmer. This variable must not be modified within the loop.
- `Logical-op` must be one of `<, <=, >,` or `>=`
- `Incr-expr` must a decrement or increment op applied to `var`, or a simple increment/decrement assignment operation with a loop invariant integer expression.
- `lb`, `b` and `incr` are loop invariant integer expressions. No synchronization takes place during the evaluation of these expressions and any side effects produce indeterminate results.

The `ordered` clause tells the compiler to expect an `ordered` directive in the body of the `for` loop. The `schedule` clause defines how the iterations are mapped onto the team of threads:

- `schedule(static[, chunk_size]):` iterations are divided into chunks of size `chunk_size` and assigned to the members of the team in a round robin fashion. If `chunk_size` isn't given, approximately equal sized chunks are assigned one to each thread.
- `schedule(dynamic[, chunk_size]):` iterations are divided into chunks of size `chunk_size` and assigned one-by-one to the threads as they finish the previous chunk of iterations. When no `chunk_size` is given, it defaults to `1.`
- `schedule(guided[, chunk_size]):` iterations are assigned to threads as with the dynamic schedule, but the chunks are of decreasing sizes. The number of iterations in a chunk start at some large value and decrease down to `chunk_size.` If `chunk_size` equals `1`, the size of each chunk is approximately the number of unassigned iterations divided by the number of threads in the team. If `chunk_size` isn't specified, it defaults to one.
- `schedule(runtime):` The schedule and chunk size are determined at runtime by setting the runtime variable `OMP_SCHEDULE`. If this variable is not set, the behavior is implementation dependent.

Restrictions:

- The `for` loop must be a structured block. Its execution can not be terminated by a `break` statement.
- The values of the loop control expressions in the `for` loop associated with a `for` directive must be the same for all the threads in the team.
- The `for` loop iteration variable must have a signed integer type.
- Only a single `schedule` clause can appear on a `for` directive.
- Only a single `ordered` clause can appear on a `for` directive.
- It is unspecified if or how often any side effects within the `chunk_size`, `lb`, `b`, or `incr` expressions occur.
- The value of the `chunk_size` expression must be the same for all threads in the team.

sections/section construct

```
#pragma omp sections [clause[ clause] …]
#pragma omp section
```

Where *clause* is:

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

Usage Note:

The sections construct is used as follows with a sequence of structured blocks

```
        #pragma omp sections [clause[ clause] …]
        {
        [#pragma omp section]
                structured-block
        [#pragma omp section
                .
                .
        ]
        }
```

Restrictions:

- A `section` directive must be inside the lexical extent of a `sections` directive.

single construct

```
#pragma omp single [clause[ clause] …]
```

Where *clause* is:

```
private(list)
firstprivate(list)
nowait
```

Combined Parallel Work-sharing Constructs

parallel for construct

```
#pragma omp parallel for [clause[ clause] …]
      for-loop
```

Where *clause* is:

```
if (scalar-expression)
private(list)
firstprivate(list)
default (shared | none)
shared(list)
copyin(list)
schedule(type[, chunk_size])
ordered
nowait
lastprivate(list)
reduction(operator: list)
```

Usage Note:

The construct is the same as a `parallel` construct immediately followed by a `for` work sharing construct.

Restrictions:

This construct shares restrictions with the `parallel` and `for` constructs.

parallel sections construct

```
#pragma omp parallel sections [clause[ clause] …]
#pragma omp section
```

Where *clause* is:

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

Usage Note:

This construct is the same as a `parallel` construct followed by a `sections` directive.

Restrictions:

This construct shares restrictions with the `parallel` and `sections` constructs.

Master and Synchronization Constructs

master construct

```
#pragma omp master
```

atomic construct

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

Usage Note:

The `atomic` construct is semantically equivalent to `critical` statement. The single statement `expression-stmt` must use one of the following forms:

```
#pragma omp atomic
      x binop = expr or x++ or ++x or x-- or --x
```

Where `x` is an lvalue expression of scalar type and no side effects.

`expr` is an lvalue expression with no side effects. It must not reference `x`.

`binop` is not overloaded and is one of `+, *, -, /, &, ^, <<, >>, |`

Restriction

All atomic references to the storage location x throughout the program are required to have a compatible type.

barrier construct

```
#pragma omp barrier
```

Restrictions:

The smallest statement that contains a `barrier` directive must be a block (or a compound statement).

critical construct

```
#pragma omp critical [(name)]
```

Where *`name`* is: `An identifier`

flush construct

```
#pragma omp flush [(list)]
```

Where *`list`* is a comma-separated list of variables that need to be flushed

A `flush` is implied by the following constructs:

- `barrier`
- At entry to and exit from `critical`
- At entry to and exit from `ordered`
- At the exit from `parallel`.
- At exit from `for`
- At exit from `sections`
- At exit from `single`

Restriction

A variable specified in a `flush` must not have a reference type.

ordered construct

```
#pragma omp ordered
```

Restrictions:

- An `ordered` directive can only appear in the dynamic extent of a `for` directive that has the `ordered` clause specified.
- An iteration of a loop with a `for` construct must not execute the same `ordered` directive more than once, and it must not execute more than one `ordered` directive.

Data Environment Constructs and Clauses

threadprivate construct

```
#pragma omp threadprivate(list)
```

Where `list` is a comma separated list of variables that do not have an incomplete type.

Restrictions

- A `threadprivate` variable must not appear in any clause other than the `copyin`, schedule, or the if clause. A default clause does not effect a `threadprivate` variable.
- The address of a `threadprivate` variable is not an address constant.
- A `threadprivate` variable must not have a reference type.
- A `threadprivate` variable with class type must have an accessible, unambiguous default constructor.

copyin clause

```
copyin (list)
```

where `list` contains `threadprivate` variables.

Restrictions:

- A variable that is specified in a `copyin` clause must have an accessible, unambiguous copy assignment operator.

default clause

```
default(shared | none)
```

Restrictions:

- Only a single `default` clause may be specified on a `parallel` directive.

firstprivate clause

```
firstprivate(list)
```

Restrictions:

- All restrictions on `private` apply except for the restriction on `const`-qualified types.
- A variable with a class type that is specified `firstprivate` must have an accessible unambiguous copy constructor.

lastprivate clause

```
lastprivate(list)
```

Restrictions:

- All restrictions on `private` apply.
- A variable that is specified as `lastprivate` must have an accessible, unambiguous copy assignment operator.

private clause

```
private(list)
```

Restrictions:

- A variable with a class type that is specified as `private` must have an accessible, unambiguous default constructor.
- Unless it has a class type with a mutable member, a variable specified as `private` must not have a `const`-qualified type.
- A variable specified as `private` must not have an incomplete type or a reference type.
- Variables that are specified `private` on a `parallel` directive cannot be specified `private` again on an enclosed work-sharing or `parallel` directive. As a result, variables that are specified `private` on a work-sharing or `parallel` directive must be specified `shared` in the enclosing parallel region

reduction clause

reduction (*op*:list)

Where *op* is:

`+`	`Initial value = 0`
`*`	`Initial value = 1`
`-`	`Initial value = 0`
`&`	`Initial value = ~0`
`|`	`Initial value = 0`
`^`	`Initial value = 0`
`&&`	`Initial value = 1`
`||`	`Initial value = 0`

Usage Note:

A reduction is typically used in a statement with one of the following forms:

```
x = x op expr
x <op> = expr
x = expr op x (except for subtraction)
x++ or ++x or x-- or --x
```

where `expr` does not reference `x`.

Restrictions:

- The type of the variables in the `reduction` clause must be valid for the `reduction operator` except that pointer types and reference types are never permitted.
- A variable that is specified in the `reduction` clause must not be `const`-qualified.
- A variable that is specified in the `reduction` clause must be `shared` in the enclosing parallel region.

shared clause

```
shared(list)
```

Directive binding

An OpenMP C/C++ program must adhere to the following rules with respect to directive binding:

- The `for`, `sections`, `single`, `master` and `barrier` directives bind to the dynamically enclosing `parallel`, if one exists. If no parallel region is currently being executed, the directives apply to a team consisting of the master thread.
- The `ordered` directive binds to the dynamically enclosing `for`.
- The `atomic` directive enforces exclusive access with respect to `atomic` directives in all threads, not just the current team.
- The `critical` directive enforces exclusive access with respect to `critical` directives in all threads, not just the current team.
- A directive can never bind to any directive outside the closest enclosing `parallel`.

Directive Nesting

An OpenMP C/C++ program must adhere to the following rules with respect to the dynamic nesting of directives:

- A `parallel` directive dynamically inside another `parallel` logically establishes a new team which is composed of only the current thread, unless nested parallelism is enabled.
- `For`, `sections`, and `single` directives that bind to the same `parallel` are not allowed to be nested inside each other.
- `Critical` directives with the same name are not allowed to be nested inside each other
- `For`, `sections` and `single` directives are not permitted in the dynamic extent of `critical`, `ordered` and `master` regions.
- `Barrier` directives are not permitted in the dynamic extent of `for`, `ordered`, `sections`, `single`, `master` and `critical` regions.
- `Master` directives are not permitted in the dynamic extent of `for`, `sections`, and `single` directives.
- `Ordered` directives are not allowed in the dynamic extent of `critical` regions.
- Any directive that is permitted when executed dynamically inside a parallel region is also permitted when executed outside a parallel region.. When executed dynamically outside a user-specified parallel region, the directive is executed with respect to a team composed of only the master thread.

Runtime Library Functions

These routines use the include file <omp.h>. This file includes function prototypes and defines the type `omp_lock_t`.

Execution environment functions

```
void omp_set_num_threads(int num_threads);
int omp_get_num_threads(void);
int omp_get_max_threads(void);
int omp_get_thread_num(void);
int omp_get_num_procs(void);
int omp_in_parallel(void);
void omp_set_dynamic(int dynamic_threads);
int omp_get_dynamic(void);
void omp_set_nested(int nested);
int omp_get_nested(void);
```

Lock functions

```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);

void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);

void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);

void omp_unset_lock(omp_lock_t *lock);
void omp_nuset_nest_lock(omp_nest_lock_t *lock);

int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_nest_lock_t *lock);
```

Environment Variables

```
OMP_SCHEDULE "schedule[, chunk_size]"
OMP_NUM_THREADS int
OMP_DYNAMIC TRUE || FALSE
OMP_NESTED TRUE || FALSE
```

About this document

This note was written by Tim Mattson of Intel Corporation. It is based on the OpenMP specification "OpenMP C/C++ Applications Program Interface" version 1.0 dated October 1998.

Please send any comments or corrections to Tim Mattson at timothy.g.mattson@intel.com.