OpenMP C and C++ Application Program Interface

DRAFT

Version 2.0 November 2001 DRAFT 11.05

Document Number xxx-xxxx-xxx

Contents

CHAPTER 1

Introduction

This document specifies a collection of compiler directives, library functions, and environment variables that can be used to specify shared-memory parallelism in C and C++ programs. The functionality described in this document is collectively known as the *OpenMP C/C++ Application Program Interface (API)*. The goal of this specification is to provide a model for parallel programming that allows a program to be portable across shared-memory architectures from different vendors. The OpenMP C/C++ API will be supported by compilers from numerous vendors. More information about OpenMP, including the *OpenMP Fortran Application Program Interface*, can be found at the following web site:

`http://www.openmp.org`

The directives, library functions, and environment variables defined in this document will allow users to create and manage parallel programs while permitting portability. The directives extend the C and C++ sequential programming model with single program multiple data (SPMD) constructs, work-sharing constructs, and synchronization constructs, and they provide support for the sharing and privatization of data. Compilers that support the OpenMP C and C++ API will include a command-line option to the compiler that activates and allows interpretation of all OpenMP compiler directives.

1.1 Scope

This specification covers only user-directed parallelization, wherein the user explicitly specifies the actions to be taken by the compiler and run-time system in order to execute the program in parallel. OpenMP C and C++ implementations are not required to check for dependencies, conflicts, deadlocks, race conditions, or other problems that result in incorrect program execution. The user is responsible for ensuring that the application using the OpenMP C and C++ API constructs executes correctly. Compiler-generated automatic parallelization and directives to the compiler to assist such parallelization are not covered in this document.

1.2 Definition of Terms

The following terms are used in this document:

barrier	A synchronization point that must be reached by all threads in a team. Each thread waits until all threads in the team arrive at this point. There are explicit barriers identified by directives and implicit barriers created by the implementation.
construct	A construct is a statement. It consists of a directive and the subsequent structured block. Note that some directives are not part of a construct. (See *openmp-directive* in Appendix C).
directive	A C or C++ **`#pragma`** followed by the **`omp`** identifier, other text, and a new line. The directive specifies program behavior.
dynamic extent	All statements in the *lexical extent*, plus any statement inside a function that is executed as a result of the execution of statements within the lexical extent. A dynamic extent is also referred to as a *region*.
lexical extent	Statements lexically contained within a *structured block*.
master thread	The thread that creates a team when a *parallel region* is entered.
parallel region	Statements that bind to an OpenMP parallel construct and may be executed by multiple threads.
private	A private variable names a block of storage that is unique to the thread making the reference. Note that there are several ways to specify that a variable is private: a definition within the parallel region, a **`threadprivate`** directive, a **`private`**, **`firstprivate`**, **`lastprivate`**, or **`reduction`** clause, or use of the variable as a **`for`** loop control variable in a **`for`** loop immediately following a **`for`** or **`parallel for`** directive.
region	A dynamic extent.
serial region	Statements executed only by the *master thread* outside of the dynamic extent of any *parallel region*.
serialize	To execute a parallel construct with a team of threads consisting of only a single thread (which is the master thread for that parallel construct), with serial order of execution for the statements within the structured block (the same order as if the block were not part of a parallel construct), and with no effect on the value returned by **`omp_in_parallel()`** (apart from the effects of any nested parallel constructs).

shared	A shared variable names a single block of storage. All threads in a team that access this variable will access this single block of storage.
structured block	A structured block is a statement (single or compound) that has a single entry and a single exit. No statement is a structured block if there is a jump into or out of that statement (including a call to **`longjmp`**(3C) or the use of **`throw`**, but a call to **`exit`** is permitted). A compound statement is a structured block if its execution always begins at the opening **{** and always ends at the closing **}**. An expression statement, selection statement, iteration statement, or **`try`** block is a structured block if the corresponding compound statement obtained by enclosing it in **{** and **}** would be a structured block (this is an "as if" rule). A jump statement, labeled statement, or declaration statement is not a structured block.
team	One or more threads cooperating in the execution of a construct.
thread	An execution entity having a serial flow of control, a set of private variables, and access to shared variables.
variable	An identifier, optionally qualified by namespace names, that names an object.

1.3 Execution Model

OpenMP uses the fork-join model of parallel execution. Although this fork-join model can be useful for solving a variety of problems, it is somewhat tailored for large array-based applications. OpenMP 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 does not behave correctly when executed sequentially. Furthermore, different degrees of parallelism 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.

A program written with the OpenMP C/C++ API begins execution as a single thread of execution called the *master thread*. The master thread executes in a serial region until the first parallel construct is encountered. In the OpenMP C/C++ API, the **`parallel`** directive constitutes a parallel construct. When a parallel construct is encountered, the master thread creates a team of threads, and the master becomes master of the team. Each thread in the team executes the statements in the dynamic extent of a parallel region, except for the work-sharing constructs. Work-sharing constructs must be encountered by all threads in the team, and the statements within the associated structured block are executed by one or more of the threads. The barrier implied at the end of a work-sharing construct without a **nowait** clause is executed by all threads in the team.

If a thread modifies a shared object, it affects not only its own execution environment, but also those of the other threads in the program. The modification is guaranteed to be complete, from the point of view of one of the other threads, at the next sequence point (as defined in the base language) only if the object is declared to be volatile. Otherwise, the modification is guaranteed to be complete after first the modifying thread, and then (or concurrently) the other threads, encounter a **flush** directive that specifies the object (either implicitly or explicitly). Note that when the **flush** directives that are implied by other OpenMP directives are not sufficient to ensure the desired ordering of side effects, it is the programmer's responsibility to supply additional, explicit **flush** directives.

Upon completion of the parallel construct, the threads in the team synchronize at an implicit barrier, and only the master thread continues execution. Any number of parallel constructs can be specified in a single program. As a result, a program may fork and join many times during execution.

The OpenMP C/C++ API allows programmers to use directives in functions called from within parallel constructs. Directives that do not appear in the lexical extent of a parallel construct but may lie in the dynamic extent are called *orphaned* directives. Orphaned directives give programmers the ability to execute major portions of their program in parallel with only minimal changes to the sequential program. With this functionality, users can code parallel constructs at the top levels of the program call tree and use directives to control execution in any of the called functions.

Unsynchronized calls to C and C++ output functions that write to the same file may result in output in which data written by different threads appears in non-deterministic order. Similarly, unsynchronized calls to input functions that read from the same file may read data in non-deterministic order. Unsynchronized use of I/O, such that each thread accesses a different file, produces the same results as serial execution of the I/O functions.

1.4 Compliance

An implementation of the OpenMP C/C++ API is *OpenMP-compliant* if it recognizes and preserves the semantics of all the elements of this specification, as laid out in Chapters 1, 2, 3, 4, and Appendix C., Appendix A, B, D, E, and F are for information purposes only and are not part of the specification. Implementations that include only a subset of the API are not OpenMP-compliant.

The OpenMP C and C++ API is an extension to the base language that is supported by an implementation. If the base language does not support a language construct or extension that appears in this document, the OpenMP implementation is not required to support it.

All standard C and C++ library functions and built-in functions (that is, functions of which the compiler has specific knowledge) must be thread-safe. Unsynchronized use of thread–safe functions by different threads inside a parallel region does not produce undefined behavior. However, the behavior might not be the same as in a serial region. (A random number generation function is an example.)

The OpenMP C/C++ API specifies that certain behavior is *implementation-defined.* A conforming OpenMP implementation is required to define and document its behavior in these cases. See Appendix E, page 97, for a list of implementation-defined behaviors.

1.5 Normative References

- ISO/IEC 9899:1999, *Information Technology - Programming Languages - C*. This OpenMP API specification refers to ISO/IEC 9899:1999 as C99.
- ISO/IEC 9899:1990, *Information Technology - Programming Languages - C*. This OpenMP API specification refers to ISO/IEC 9899:1990 as C90
- ISO/IEC 14882:1998, *Information Technology - Programming Languages - C++*. This OpenMP API specification refers to ISO/IEC 14882:1998 as C++

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

1.6 Organization

- Directives (see Chapter 2).
- Run-time library functions (see Chapter 3).
- Environment variables (see Chapter 4).
- Examples (see Appendix A).
- Stubs for the run-time library (see Appendix B).
- OpenMP Grammar for C and C++ (see Appendix C).
- Using the **`schedule`** clause (see Appendix D).
- Implementation-defined behaviors in OpenMP C/C++ (see Appendix E).
- New features in OpenMP C/C++ Version 2.0 (see Appendix F).

CHAPTER **2**

Directives

Directives are based on **#pragma** directives defined in the C and C++ standards. Compilers that support the OpenMP C and C++ API will include a command line option that activates and allows interpretation of all OpenMP compiler directives.

2.1 Directive Format

The syntax of an OpenMP directive is formally specified by the grammar in Appendix C, and informally as follows:

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

Each directive starts with **#pragma omp**, to reduce the potential for conflict with other (non-OpenMP or vendor extensions to OpenMP) pragma directives with the same names. 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 the **#pragma omp** are subject to macro replacement.

Directives are case-sensitive. The order in which clauses appear in directives is not significant. Clauses on directives may be repeated as needed, subject to the restrictions listed in the description of each clause. If *variable-list* appears in a clause, it must specify only variables. Only one directive name can be specified per directive. For example, the following directive is not allowed:

```
/* ERROR - multiple directive names not allowed */
#pragma omp parallel barrier
```

An OpenMP directive applies to at most one succeeding statement, which must be a structured block.

2.2 Conditional Compilation

The **_OPENMP** macro name is defined by OpenMP-compliant implementations as the decimal constant *yyyymm*, which will be the year and month of the approved specification. This macro must not be the subject of a **#define** or a **#undef** preprocessing directive.

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

If vendors define extensions to OpenMP, they may specify additional predefined macros.

2.3 parallel Construct

The following directive defines a parallel region, which is a region of the program that is to be executed by multiple threads in parallel. This is the fundamental construct that starts parallel execution.

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

The *clause* is one of the following:

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

When a thread encounters a parallel construct, a team of threads is created if one of the following cases is true:

- No **`if`** clause is present.
- The **`if`** expression evaluates to a non-zero value.

This thread becomes the master thread of the team, with a thread number of 0, and all threads, including the master thread, execute the region in parallel. If the value of the **`if`** expression is zero, the region is serialized.

To determine the number of threads that are requested, the following rules will be considered in order. The first rule whose condition is met will be applied:

1. If the **`num_threads`** clause is present, then the value of the integer expression is the number of threads requested.
2. If the **`omp_set_num_threads`** library function has been called, then the value of the argument in the most recently executed call is the number of threads requested.
3. If the environment variable **`OMP_NUM_THREADS`** is defined, then the value of this environment variable is the number of threads requested.
4. If none of the methods above were used, then the number of threads requested is implementation-defined.

If the **`num_threads`** clause is present then it supersedes the number of threads requested by the **`omp_set_num_threads`** library function or the **`OMP_NUM_THREADS`** environment variable only for the parallel region it is applied to. Subsequent parallel regions are not affected by it.

The number of threads that execute the parallel region also depends upon whether or not dynamic adjustment of the number of threads is enabled. If dynamic adjustment is disabled, then the requested number of threads will execute the parallel region. If dynamic adjustment is enabled then the requested number of threads is the maximum number of threads that may execute the parallel region.

If a parallel region is encountered while dynamic adjustment of the number of threads is disabled, and the number of threads requested for the parallel region exceeds the number that the run-time system can supply, the behavior of the program is implementation-defined. An implementation may, for example, interrupt the execution of the program, or it may serialize the parallel region.

The **`omp_set_dynamic`** library function and the **`OMP_DYNAMIC`** environment variable can be used to enable and disable automatic adjustment of the number of threads.