




OPENMP - HIGH-LEVEL PARALLELISM



Emily Bodenhamer
CS 495 Dr. Donald Davendra

Table of Contents

Introduction.....	2
Scalability & Portability	2
Benefits & Limitations.....	2
Architecture.....	3
Execution Model	3
Memory Model	4
Pragmas.....	4
Parallel Construct	5
Library Functions.....	6
Environment variables	7
Example	8
Conclusion	9
References.....	9

Table of Figures

Figure 1. The Fork and Join Model..	3
Figure 2. A simple Hello World application in C using the OpenMP API.....	8
Figure 3. The result obtained from the C code.	8

Introduction

OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism. OpenMP is the most widely used standard for SMP systems, it supports Fortran and C/C++, and has been implemented by many vendors. OpenMP is a small and simple specification and it supports incremental parallelism. Research is constantly done using OpenMP and it keeps up to date with the latest hardware developments.

Scalability & Portability

Scalability is achieved by using shared-memory parallel programming. OpenMP provides constructs that simplify the specification of the parallelism and can be implemented with low overhead by compiler vendors. The OpenMP model allows for portability across shared memory architectures from different vendors. The directives, library routines, and environment variables in the OpenMP API allows users to create and manage parallel programs while remaining portable.

Benefits & Limitations

OpenMP is simple to implement onto already existing code. It's easier to program and debug than MPI, which is focused on mobile devices and not multi-core processors in general. Parallelization can be implemented gradually. By using directives, a programmer can specify which sections of code to parallelize. A program can still be run as serial code.

OpenMP can only be run in shared memory computers, no codes can run on distributed memory computers. OpenMP requires a compiler that supports its API; however, most compilers do support OpenMP. OpenMP is mostly used for loop parallelization.

Architecture

Execution Model

OpenMP uses the fork-join model of parallel execution. This model uses all the available processing power to enhance the performance of an application. This model is tailored more towards large array-based applications. OpenMP is suited for programs that will execute as both parallel programs and as sequential programs. Programs that do not execute correctly sequentially can also be used too.

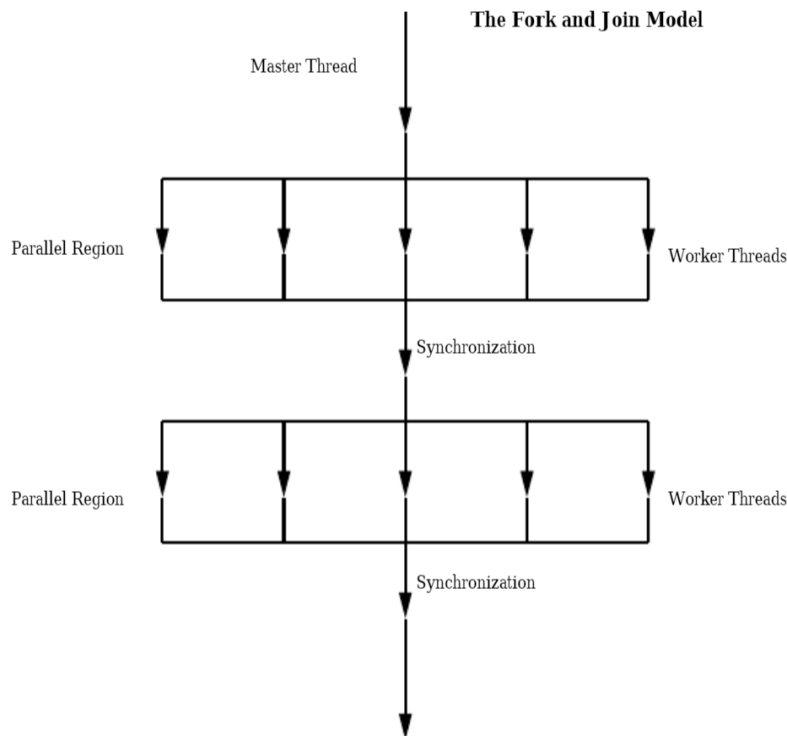


Figure 1. The Fork and Join Model.

A program using OpenMP will first begin with a master thread, this is running sequentially until threads reach a parallel section where they fork into multiple threads. The threads re-join into the master thread once at the end of the parallel section. The parallel directive constitutes a parallel construct. If a thread modifies a shared object, it affects its own execution environment and those of the other threads in the program. There is no limitation to the parallel constructs that can be specified in a program. This program would then fork and join many times during execution.

Memory Model

OpenMP uses a shared-memory model. Threads have access to memory which stores and retrieves variables. Each thread can have its own temporary view of the memory. This temporary view allows the thread to cache variables and avoid going to memory for every reference to a variable. Each thread also has access to threadprivate memory which is another type of memory not accessible by other threads.

The **share** and **private** clauses can be used to accept data-sharing attribute clauses to determine access to variables. Each reference to a shared variable in the structured block becomes a reference to the original variable located outside the block. Creating the new version does not change the value of the original variable. Each thread has its own copy of a private variable that can not be accessed by other threads. The scope of the variables is up to the programmer to make sure the memory accessed is correct.

Pragmas

OpenMP uses directives in functions that are called from within parallel constructs. These directives lie within the dynamic extent and are called orphaned directives. These

directives execute major portions of the program in parallel with only minimal changes to the sequential program. Parallel constructs can be coded at the top levels of the program call tree and directives can be used to control execution in any of the called functions.

Unsynchronized calls to output functions that are written to the same file could produce results of data written by different threads appearing in nondeterministic order. The same nondeterministic order occurs in input functions that read from the same file.

The OpenMP directives are based on **#pragma** directives defined in the C and C++ standards. Each directive starts with **#pragma omp directive-name [clause[[,] clause]...] new-line**. This is done to reduce the potential for conflict with other pragma directives with the same name. Directives are case-sensitive. Only one directive name can be specified per directive.

There are numerous pragmas for OpenMP, one that will be expanded upon is the parallel construct.

Parallel Construct

To define a parallel region, where multiple threads are executed in parallel, this directive will be created **#pragma omp parallel [clause[[,] clause] ...] new-line structured-block**. When a thread finds a parallel construct, a team of threads are created if there are no **if** clause or the **if** expression evaluates to a nonzero value. The thread becomes the master thread of the team, having a thread number of 0. All threads execute the region in parallel. If the value of the **if** expression is zero, then the region is serialized.

The number of threads is determined by different rulings.

1. If the **num_threads** clause is called, then the value of the integer expression is the number of threads requested.

2. If the **omp_set_num_threads** library function is 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 this value is the number of threads requested.
4. If none of the 3 previous rules were used, then the number of threads requested is implementation-defined.

The **num_threads** clause supersedes the number of threads requested by rules 2 and 3 only for the parallel region it is applied to. The number of threads executed also depends on if the dynamic adjustment of the number of threads is enabled. If disabled, the number of threads will execute the parallel region. If enabled, the number of threads is the maximum number of threads that may execute the parallel region. The **omp_set_dynamic** library function and the **OMP_DYNAMIC** environment variable can be used to enable and disable dynamic adjustment of the number of threads.

Once at the end of a parallel region, the master thread of the team is the only thread to continue execution. If a thread in a team executing a parallel region finds another parallel construct, that thread creates a new team and it becomes the master thread of that new team. Nested parallel regions are serialized. This serialization can be changed by using the runtime library function **omp_set_nested** or the environment variable **OMP_NESTED**. However, the number of threads in that team is implementation-defined.

Library Functions

The **<omp.h>** header declares several functions that can be used to control and query the parallel execution environment and lock functions that can be used to synchronize access to data.

Omp_lock_t is an object type that means a lock is available or a thread owns a lock. Locks such as these are simple locks. **Omp_nest_lock_t** is an object type that means a lock is available or both the identity of the thread that owns the lock and a nesting count.

There are functions that can affect and monitor threads, processors, and parallel environment. One being **omp_set_num_threads**, written as, **#include void omp_set_num_threads(int num_threads);** which as discussed before sets the default number of threads to use for subsequent parallel regions that do not specify a **num_threads** clause.

Lock functions can be used for synchronization. A lock variable can be of type **omp_lock_t** or **omp_nest_lock_t**. There are different functions that can be only used for each variable type and the functions have a pointer to that specified type. Some functions can initialize a lock, removes a lock, wait for an available lock, release a lock, or test a lock. The OpenMP lock functions access the lock variable so that they are always read and update the most current value of the lock variable. Lock variables are consistent among different threads.

Environment variables

Environment variables control the execution of parallel code. The names are always uppercase. The values assigned to them are case insensitive and may have white spaces. Modifications to the values after the program starting is ignored. There are four variables, **OMP_SCHEDULE**, **OMP_NUM_THREADS**, **OMP_DYNAMIC**, and **OMP_NESTED**.

OMP_SCHEDULE, is used only to **for** and **parallel for** directives that have the schedule type **runtime**. This variable sets the run-time schedule type and chunk size. **OMP_NUM_THREADS** will set the number of threads to use during execution. Its effect depends upon whether dynamic adjustment of the number of threads is enabled.

OMP_DYNAMIC enables or disables dynamic adjustment of the number of threads.

This is set to **TRUE** or **FALSE**. **OMP_NESTED** enables or disables nested parallelism. This is set to **TRUE** or **FALSE**.

Example

The following figure example shows a simple Hello World application. The code will print to the console, “Hello from thread _, nthreads _” using OpenMP directives and functions.

```
#include <omp.h>
#include <stdio.h>
int main()
{
    #pragma omp parallel
    printf("Hello from thread %d, nthreads %d\n", omp_get_thread_num(), omp_get_num_threads());
}
```

Figure 2. A simple Hello World application in C using the OpenMP API.

The first thing to observe in this example is including **<omp.h>**, this will allow for the ability to use the libraries for OpenMP. The first directive that is used in this code is **#pragma omp parallel**. This directive allows for the definition of a parallel region. In the print statement, **omp_get_thread_num()** is called to return the thread number within its team, of the thread executing the function. The master thread is always 0. Next, **omp_get_num_threads()** is called to return the number of threads currently in the team executing in the parallel region.

```
Hello from thread 0, nthreads 4
Hello from thread 1, nthreads 4
Hello from thread 2, nthreads 4
Hello from thread 3, nthreads 4

C:\Users\emyli\Desktop\CS473\HelloWorld\Debug\HelloWorld.exe (process 14604) exited with code 0.
```

Figure 3. The result obtained from the C code.

The output for this program shows that **omp_get_num_threads()** returns 4, since there was no number of threads set, the default of 4 was used since my system has 4 processors. The

`omp_get_thread_num()` returned each thread in the team. Since there were 4 threads in the team because of my 4 processors, the print statement executed 4 times for each thread.

Conclusion

This document outlined some basics of OpenMP and illustrated the potential that OpenMP offers. There is a plethora of different directives that OpenMP offers as well as more details that was not expanded upon in this documentation. OpenMP offers a simple API that can be used on different compilers to improve code performance. OpenMP is a great platform to use in order to try to obtain better results for research purposes. With the amount of literature available for OpenMP, there will be no difficulty finding specification explanation.

References

Dartmouth. “Pros and Cons of OpenMP/MPI.” *Dartmouth.edu*, 14 Feb. 2011,
https://www.dartmouth.edu/~rc/classes/intro_mpi/parallel_prog_compare.html

Kiessling, Alina. “An Introduction to Parallel Programming with OpenMP.” *ResearchGate*, April. 2009,
https://www.researchgate.net/figure/The-Fork-and-Join-Model-OpenMP-programs-start-with-a-master-thread-running_fig3_265109968

OpenMP ARB Corporation. “OpenMP FAQ.” *Openmp.org*, Version 3.0. 6 June 2018,
<https://www.openmp.org/about/openmp-faq/#WhatIs>

OpenMP Architecture Review Board. “OPENMP API Specification: Version 5.0.” *Openmp.org*, Nov. 2018, <https://www.openmp.org/spec-html/5.0/openmpsu9.html>

OpenMP Architecture Review Board. “OpenMP C and C++ Application Program Interface.”

Version 2.0. Mar. 2002, <https://www.openmp.org/wp-content/uploads/cspec20.pdf>