



Choosing between OpenMP* and Explicit Threading Methods

by Andrew Binstock

The information contained in this document is provided for informational purposes only and represents the current view of Intel Corporation Intel on the date of publication. Intel makes no commitment to update the information contained in this document, and Intel reserves the right to make changes at any time, without notice.

DISCLAIMER. THIS DOCUMENT AND ALL INFORMATION CONTAINED HEREIN IS PROVIDED AS IS. INTEL MAKES NO REPRESENTATIONS OF ANY KIND WITH RESPECT TO PRODUCTS REFERENCED HEREIN, WHETHER SUCH PRODUCTS ARE THOSE OF INTEL OR THIRD PARTIES. INTEL EXPRESSLY DISCLAIMS ANY AND ALL WARRANTIES, IMPLIED OR EXPRESS, INCLUDING WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, AND ANY WARRANTY ARISING OUT OF THE INFORMATION CONTAINED HEREIN, INCLUDING WITHOUT LIMITATION, ANY PRODUCTS, SPECIFICATIONS, OR OTHER MATERIALS REFERENCED HEREIN. INTEL DOES NOT WARRANT THAT THIS DOCUMENT OR THE INFORMATION CONTAINED HEREIN IS FREE FROM ERRORS, OR THAT ANY PRODUCTS OR OTHER TECHNOLOGY DEVELOPED IN CONFORMANCE WITH THIS DOCUMENT WILL PERFORM IN THE INTENDED MANNER, OR WILL BE FREE FROM INFRINGEMENT OF THIRD PARTY PROPRIETARY RIGHTS, AND INTEL DISCLAIMS ALL LIABILITY THEREFOR.

INTEL DOES NOT WARRANT THAT ANY PRODUCT REFERENCED HEREIN OR ANY PRODUCT OR TECHNOLOGY DEVELOPED IN RELIANCE UPON THIS DOCUMENT, IN WHOLE OR IN PART, WILL BE SUFFICIENT, ACCURATE, RELIABLE, COMPLETE, FREE FROM DEFECTS OR SAFE FOR ITS INTENDED PURPOSE, AND HEREBY DISCLAIMS ALL LIABILITIES THEREFOR. ANY PERSON MAKING, USING OR SELLING SUCH PRODUCT OR TECHNOLOGY DOES SO AT HIS OR HER OWN RISK.

Licenses may be required. Intel and others may have patents or pending patent applications, trademarks, copyrights or other intellectual proprietary rights covering subject matter contained or described in this document. No license, express, implied, by estoppels or otherwise, to any intellectual property rights of Intel or any other party is granted herein. It is your responsibility to seek licenses for such intellectual property rights from Intel and others where appropriate.

Limited License Grant. Intel hereby grants you a limited copyright license to copy this document for your use and internal distribution only. You may not distribute this document externally, in whole or in part, to any other person or entity.

LIMITED LIABILITY. IN NO EVENT SHALL INTEL HAVE ANY LIABILITY TO YOU OR TO ANY OTHER THIRD PARTY, FOR ANY LOST PROFITS, LOST DATA, LOSS OF USE OR COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, OR FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF YOUR USE OF THIS DOCUMENT OR RELIANCE UPON THE INFORMATION CONTAINED HEREIN, UNDER ANY CAUSE OF ACTION OR THEORY OF LIABILITY, AND IRRESPECTIVE OF WHETHER INTEL HAS ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING THE FAILURE OF THE ESSENTIAL PURPOSE OF ANY LIMITED REMEDY.

Intel, the Intel logo, Pentium, Intel Xeon, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2005 Intel Corporation

Introduction

Other articles on this Web site extol the virtues of programming with OpenMP, a vendor-neutral interface for threading portions of programs in a simple, portable fashion. It consists of a set of pragmas, APIs, and environment variables, and it is supported by compilers on a wide range of platforms. OpenMP's greatest attributes are this portability and the simplicity it brings to parallel programming. Let's look at a quick example, written in C/C++:

```
int j;

#pragma omp parallel for
for ( j = 0; j < ARRAY_SIZE; j++ );
    array[j] += j;
```

Figure 1. Using OpenMP to parallelize a simple for-loop.

The `pragma` statement that appears just before the for-loop tells the compiler to generate threaded code that will do the following: start up the appropriate number of threads for the run-time environment, break up the for-loop work across these threads, and wait for the threads to complete, suspend the running threads, and return to the original thread of execution. That's rather a lot of work for a single statement, and it is all done without the developer having to do anything to create or manage threads.

Strengths of OpenMP

If the compiler does not recognize the pragma statement, it is skipped (per the ANSI standards for C and C++). Thus, code containing these pragmas compiles as single-threaded code if the compiler does not support OpenMP, and as multithreaded code if the compiler does support OpenMP. Notice that OpenMP does not require that single-threaded code be changed for threading. OpenMP only adds compiler directives in the form of pragmas. By disabling OpenMP, the codebase will compile and work exactly as it did previously. (OpenMP supports the same functionality in Fortran via the use of directives, rather than pragmas. More on OpenMP syntax and operation can be found at openmp.org.)

Developers who have studied program performance know that hot spots tend to occur inside loops, and that one of the simplest ways to resolve these hotspots is to use data decomposition to distribute the loop's work across multiple threads. This simple, effective solution suffers from one drawback in explicit threading APIs, such as Win32* or UNIX/Linux* Pthreads. Specifically, how does one know how many threads will be available at run-time? Unless your code will only ever be run on a designated system, the answer is simply that you don't know. There are ways of extracting this information from the system at run time and dynamically creating the appropriate number of threads, but this process can be messy and, with Hyper-Threading Technology, error-prone.

A simpler solution is to let OpenMP figure out the correct number of threads and automate the distribution of work. In certain rare cases, the developer might need to specify a predetermined number of threads to use. This can still be done in OpenMP by use of an environment variable or an API call. OpenMP cognoscenti discourage the use of environment variables and APIs, however,

because it reduces the portability of the original program and places a key factor outside the program's control. As much as possible, users of OpenMP are encouraged to use only pragmas.

The for-loop in Figure 1 is a canonical example of what OpenMP can do. OpenMP's specific operation cannot be seen in this simple statement. The opening curly brace of the for-loop begins what OpenMP calls a parallel region: one that will rely on multiple threads under OpenMP control. All parallel regions end in a barrier. At such a barrier, the program pauses until all OpenMP threads have finished their work. This pause is important. In the case of Figure 1, you probably don't want to proceed until the entire array has been initialized.

Any transition from parallel to serial code has an implicit barrier in it. Sometimes, however, you have multiple loops working and you don't want there to be a barrier between them. You want the threads of one loop to immediately be used as threads for a second parallelized loop right after it. This can be done using the `nowait` keyword, as in the following pragma, which is used on the first loop.

```
#pragma omp for nowait
```

The keyword `nowait` means the first thread to finish will continue on to the second loop without waiting for the any of the other threads to finish the first loop.

Not all parallelizable work, of course, appears in the context of a loop. Often a program will contain independent tasks that can be executed concurrently by assigning separate tasks to different threads. This design is known as functional decomposition, and it is supported in OpenMP via the `sections` pragma:

```
#pragma omp sections
{
    #pragma omp section
    {
        TaskA();
    }

    #pragma omp section
    {
        TaskB();
    }

    #pragma omp section
    {
        TaskC();
    }
}
```

Figure 2. How to parallelize tasks in OpenMP.

The curly braces after the pragma are not needed when only one statement is executed in the section. With more than one statement, the braces are required.

When OpenMP encounters this code, each task is assigned to a thread that ultimately executes it. As with native threading APIs, OpenMP makes no guarantee whatsoever as to how these tasks will be scheduled. `TaskC()` might very well execute first.

Developers who have worked with threads know that as soon as two or more threads are running in parallel, safeguards must be put in place to prevent one of the headaches of parallel programming: keeping two threads from updating a shared data item at the same time (a situation known as a *data race*). Predictably, OpenMP provides for this need.

The pragma shown below identifies a section of code that can be executed only by one thread at a time:

```
#pragma omp critical
{
    ...some code here...
}
```

Figure 3. Locking sections for code in OpenMP.

This keyword `critical` is an allusion to the idea of critical regions as they appear in native APIs such as Pthreads and Win32. If the code is being run by one thread, any other thread that wants to execute it must wait until the first thread reaches the closing curly brace. (Notice how the curly braces play a key role here, as they did in Figures 1 and 2. The braces tell OpenMP exactly what portions of the code the pragma covers, which is why OpenMP pragmas are immediately followed either by a single statement – as in Figure 1 – or by an opening curly brace, as in Figures 2 and 3.)

As one can see from these explanations, OpenMP offers a significant subset of the functionality provided by explicit threading APIs. Its high-level implementation, however, requires OpenMP to work on code that fits within specific expectations. If code does not fit within the guidelines, OpenMP is no longer the solution of choice.

Limitations of OpenMP

Not all loops can be threaded. For example, loops whose results are used by other iterations of the same loop – a situation called flow dependency – will not work correctly. Moreover, the compiler and OpenMP code will not be able to detect this situation, and so the threaded code will generate the wrong result. Figure 4 provides an example.

```
#pragma omp parallel for
for ( i = 2; i < 10; i++ )
{
    factorial[i] = i * factorial[i-1];
}
```

Figure 4. Code that will not work in OpenMP due to flow dependency.

OpenMP does not analyze code correctness, and so it cannot detect this dependency. As a result, it will generate code that generates an incorrect result. In many cases, flow dependency will be less obvious, but the results will remain equally undesirable. Likewise, data races and other threading problems can lead to generation of code that does not work correctly. In summary, OpenMP requires that developers have made their code thread-safe.

OpenMP works at a fairly coarse-grained level. It is masterly in its ability to perform data decomposition on loops, assigning tasks to individual threads, and other high-level operations. If

your code needs to perform intricate threading operations, however, OpenMP is less suitable than native API sets.

Consider, for example, a queue into which data is being deposited by some threads and removed by others. Such a queue might be used to deposit data read from a source while waiting for threads to fetch this data and parse it. If parsing is complex and time consuming, while reading the data is fast, it might be desirable to use such a queue and assign many threads to parsing and only a few to reading.

Such a queue has complex mechanisms. Lots of locking takes place as individual threads put or remove data from the queue. When the queue is full, the input threads must wait; when the queue is empty, the parsing threads must wait. This kind of fine-grained control of threads based upon their function and the status of specific variables is nearly impossible to attain using OpenMP.

Another area where OpenMP cannot be used is changing the priority of thread execution. All native thread APIs enable a developer to specify that certain threads obtain more of the system's resources, especially execution time, by being accorded a higher priority. Individual thread priorities cannot be modified in OpenMP, and so this level of control is not available.

Several specific capabilities that are unique to different native threading APIs are also missing from OpenMP. Two of them are discussed here.

In the POSIX* support for threads, there is a locking construct called a semaphore that does more than just lock or unlock code. It enables a lock to be locked (or unlocked) by multiple threads. Specific rules apply to how access to protected code is given to threads waiting on the semaphore, and certain applications can effectively leverage this counting scheme. OpenMP has no equivalent construct.

In the Win32 threading API, there is a threading option called fibers that enables users to write their own thread scheduler and so exert fine-grained control over threading operations. This too is not possible in OpenMP.

This last point highlights a tradeoff that you must accept with OpenMP: it will do a lot of threading work behind the scenes for you. In return, you must accept that you will not know all that it is actually doing. In fact, OpenMP provides very little information on what it is doing behind the scenes. As a result, if you need to tweak one of these activities (such as raising thread priority), you cannot use OpenMP.

By the same token, if the program appears to work incorrectly under OpenMP, there is little you can do to find out what is happening. Intel's threading tools help provide some insight, and a handful of OpenMP APIs give you additional information and a limited ability to test the code under different scenarios. Beyond this, however, not much is available. In counterpoint, OpenMP implementations have a history of very reliable performance, so if code does not work correctly under OpenMP, chances are very good that your code, rather than OpenMP, is to blame.

Conclusion: How to Choose?

As we have seen, OpenMP is a powerful, portable, and simple means of threading programs. For many applications, it is entirely sufficient. Such applications are characterized by the following:

- Easy data decomposition

- Clean functional decomposition
- Simple needs for locking and mutual exclusion

Programs that manage complex interactions between threads or that rely on intimate manipulation of threading functions will need to use native threading APIs.

It is important to recognize that the choice is not exclusive. Many programs use both OpenMP and native threading APIs. Portions of programs that have aspects consistent with OpenMP threads use it, while other portions rely on native libraries. This hybrid approach permits easy threading of individual modules and rewards them with greater portability.

About the Author

Andrew Binstock is the principal analyst at Pacific Data Works LLC. Previously he was the director of PricewaterhouseCoopers's Global Technology Forecasts. He writes the business integration column for *SD Times*. His latest book, "Programming with Hyper-Threading Technology: How to Write Multithreaded Software for Intel IA-32 Processors," is now available from Intel Press.

Additional Resources

The following resources are useful adjuncts to this article:

- [Intel® Threading Developer Center](#): Comprehensive information about threading for Intel® architecture.
- [Intel® Threading Knowledgebase](#): Concise solutions to practical developer challenges related to threading.
- [Intel® Threading Forum](#): A discussion board where you can post questions and insights for industry peers and Intel experts.
- [Intel® Threading Tools](#): Intel® Thread Checker locates threading errors, and Thread Profiler simplifies performance tuning of threaded code.
- [OpenMP.org*](#): The OpenMP application programming interface home page

