February 22, 2009

INTRODUCTION TO OPENMP

John Dexter
d000hg

DURATION
17 min

CATEGORIES
Development
Miscellaneous

TAGS
OpenMP C++

SHARE

INTRODUCTION

As most people involved in programming probably know, multi-threading is going to be "the next big thing" in terms of using modern hardware to its full extent. The rapid rise in CPU clock speeds of the last 10 years has, for the time being, slowed to a crawl. Instead, hardware giants such as AMD and Intel are switching to multi-core systems - in effect, squeezing several CPUs onto one chip. Motherboards that support 2 or more separate CPUs have been around for years, but these were generally expensive and used for specialist tasks, such as servers. Now, though, standard desktop PCs are beginning to ship with dual-core CPUs, and this is just the beginning - quad core chips are already in production, 8-core chips are just around the corner, and they're even further advanced in the research labs. If clock speed remains stalled and Moore's law remains even approximately true, in 5 years a normal desktop PC could have 16 or maybe 32 cores crammed into one chip, each one equivalent to a top-line P4.

The problem is, that program you just wrote and compiled may not run any faster on such a PC than it does on current technology!

A normal program that you write is likely to be single-threaded - it consists of a long stream of instructions that are executed one after another. Such a program can only make use of a single core - which, until now, is what most PCs had. But with multi-core machines becoming standard, you'll want your program to use all those cores. In other words, you want it to do several tasks at once, in parallel. That's what multi-threading lets you do.

Every program has a single main thread, but you need to create extra threads, each running code simultaneously on a different core/CPU. While performance doesn't scale linearly with the number of cores (four cores doesn't translate to four times the performance) it doesn't take a genius to see that doing multiple things at the same time is a good thing.

The issue, of course, is how to create and manage these multiple threads in a way that yields good performance, scales well as systems have more and more cores, and is not a total nightmare to understand. One option is to specifically create threads in your code, tell them what tasks to perform, and destroy them when they finish these tasks. But then you also have to think of things like synchronisation - if starting one task requires another to be finished, then you must make sure that that's what actually happens - and dealing with issues where multiple threads try to update the same variable at the same time. This can all start to become a bit complicated – often you just want to perform several tasks simultaneously, or process a pool of objects in parallel. This is exactly the kind of thing that OpenMP was designed to accomplish.

WHAT IS OPENMP?

OpenMP is an API that is designed to facilitate the use of multithreaded programming, by creating and synchronising threads automatically at your command. It is an unofficial standard for C++ and Fortran (though this article discusses C++ only), backed by several heavyweights of the microprocessor world,

including IBM & Intel. While not yet an ANSI standard, both Microsoft and Intel's latest flagship compilers provide excellent support - and there is even a GCC-OpenMP project.

In OpenMP, you never 'see' a thread in your code. Instead, you inform the compiler that a section of code may be parallelised through use of #pragma directives. With a few clarifications to control how data is accessed and modified, the compiler is able to generate an application that consists of a single thread, which forks into several threads for a parallel region; these threads are then synchronised and all but one terminate at the end of the parallel region as the program execution reverts to a single master thread. This fork/join principle is perhaps better explained through a simple diagram:

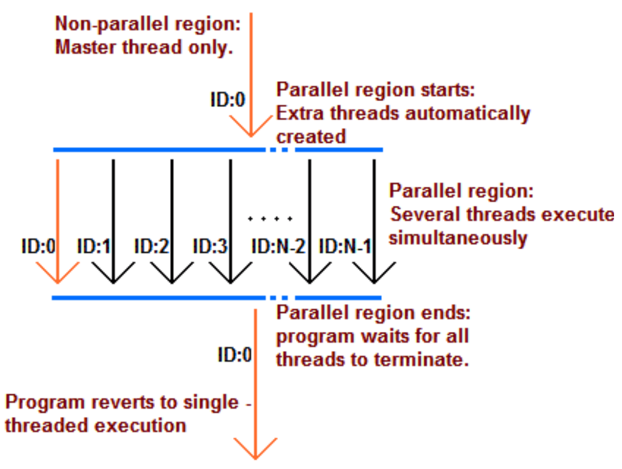
Non-parallel region:
Master thread only.
ID:0
Parallel region starts:
Extra threads automatically created
ID:0 ID:1 ID:2 ID:3 .... ID:N-2 ID:N-1
Parallel region:
Several threads execute simultaneously
Parallel region ends:
program waits for all threads to terminate.
ID:0
Program reverts to single - threaded execution

Figure 1.

USING OPENMP

OpenMP Compilers
Because OpenMP is controlled by pragmas, OpenMP C++ code will normally compile on any C++ compiler, as unrecognised pragmas should be ignored. However, several OpenMP API functions exist and to use these

you must include the OMP.h header file. The easiest way to check if your compiler has OpenMP support is to try and include this file:

```
#include <omp.h>
```

If you get errors about the file not existing, try another compiler.

Compilers that support OpenMP 2.0 include Intel's ICC (v9.0 and above), and Visual C++ 2005. There is also a project named GOMP, which will add OpenMP functionality to GCC/GNU, but at the time of writing you can get ICC for free on Linux.

Assuming you have an OpenMP-compatible compiler, you also have to enable OpenMP. In both Intel and Microsoft's compilers simply use the /OpenMP compiler option (Visual C++ allows you to do this through the project settings as a C++ language option).

Setting the thread count
Before we dive into some code we need to make sure OpenMP is set to use multiple threads in parallel sections - if you don't know this then you could spend a while wondering why your application refuses to use more than one thread! There are actually three ways that this is determined - in order of decreasing priority, they are:

1. omp_set_num_threads(int num_threads) - declared in <omp.h> - can be used to set the maximum thread count at run-time, OUTSIDE of a parallel section.
2. Set an environment variable OMP_NUM_THREADS to the maximum thread count.
3. If neither of these is used, the default is implementation specific - it could be 1, or it could be the number of cores, or it could, in theory, be any other number.
   In examples here, I'll use omp_set_num_threads. Now, let's look at our first OpenMP code… since it's the first one, I've given you a complete program that should compile as-is:

Our First OpenMP Program

Example 1:

```
 #include <iostream>

using namespace std;#include <omp.h>    //required for omp_... API functions

void main() {
  //tell OpenMP to use 3 threads in parallel regions
  omp_set_num_threads(3);

  //inform OpenMP that the following code block is a parallel region
  #pragma omp parallel {
    //create a local variable for each thread
    int thread = omp_get_thread_num();
    //prove it's multi-threaded!
    cout << "Hello world from thread " << thread << endl;
  } //end of parallel region
}
```

This code should compile without problems on any OpenMP compiler. If it does not compile, check that your compiler supports OpenMP.

When you run this, you should see the following output:

Hello world from thread 0

```
Hello world from thread 1
Hello world from thread 2
```

NOTE: If you see only a single line of output, check you have enabled OpenMP support in your compiler (typically with the /OpenMP compiler switch). Don't worry if you see these three outputs in a different order, or even partially mixed up - because the threads run simultaneously the 3 outputs can also happen simultaneously, giving something like this:

```
Hello world from thread Hello world from thread 13

Hello world from thread 2
```

We'll see how such issues can be addressed a little later.

So what's going on with this code? Figure 1 shows the program flow:

1. Initially a single master thread is running.
2. At the start of the block preceded by #pragma omp parallel, the master thread creates two child threads.
3. All three threads now execute the body of the parallel section simultaneously. Each thread creates its

   own variable thread, containing the ID of that thread. That's right - there are three different variables, each named thread.
4. At the end of the parallel section OpenMP places what is known as a barrier. This simply means that program execution waits here until all threads have completed. In normal multithreaded programming you have to explicitly create a barrier otherwise the master thread would quite happily continue on with the rest of the program - in all likelihood immediately trying to read some data which has not yet been written by an incomplete thread.
5. When all threads have reached the barrier, all but the master thread are destroyed or deactivated.
6. Program execution resumes with the master thread only - in this case, only as far as the end of the program.
   If all has gone well, you have just written your first parallel program. Now that we've got our feet wet, let's start looking at the capabilities of OpenMP in a slightly more structured fashion.

OPENMP DIRECTIVES

#pragma omp parallel
This is the main part of any OpenMP program. As we've seen, it causes a team of threads to be created that run the following code block in parallel.

The code in Example 1 is quite legal, but it does not illustrate a very important part of using this fundamental construct - data scoping modifiers, which control how variables should be treated in a parallel region.

As demonstrated, a variable declared inside the parallel region actually causes one copy of the variable to be created for each thread. Such a variable is termed to be private (note: this is nothing to do with C++ access modifiers), because each thread has a private copy. The opposite of a private variable is a shared variable. Only one copy of a shared variable will exist - the same copy is seen by all the threads. Just as variables declared inside a parallel region are automatically private, variables declared outside a parallel region are treated as shared within the parallel region by default.

Let's look at another example…

Example 2:

```
void example2() {
  //tell OpenMP to use 6 threads in parallel regions
  omp_set_num_threads(6);
  //this variable will be of shared scope in the parallel region
  int i = 0;
  //inform OpenMP that the following code block is a parallel region
  #pragma omp parallel {
    cout << "++i = " << ++i << endl;
  } //end of parallel region
}
```

Apart from the fact that our output is still likely to be messed up, you should clearly be able to see that the values 1,2,3,4,5,6 have been displayed. Clearly, the changes made by one thread are visible to the other threads - the variable is shared between the threads.

NOTE: This is just an example - in real code you really don't want to be writing to a shared variable from multiple threads at once without any safeguards. Consider the case where i=3 & two threads both want to increment it at the exact same time - both might read it as 3 and then both update it to 4! And if the update takes more than a single CPU instruction, the results can be totally undefined… don't worry though; OpenMP provides mechanisms for dealing with this situation and we'll come on to those later.

OpenMP allows us to set the scope that should be used for variables. Let's look at the syntax of the parallel construct:

```
#pragma omp parallel
[
    if (scalar_expression)] / [private(list)] / [shared(list)] / [
    default (private * | shared | none)] / [firstprivate(list)] / [reduction(o
    ...
  }
```

All the modifiers to the construct are optional, and any that you use must be on the same line. We're not going to cover what all these modifiers do (I suggest you consult a reference once you're comfortable with the basics). For now, we'll examine the most basic modifiers, followed by a look at reduction:

Private: Applying the private modifier to a list of variables causes a local copy of the variable to exist for each thread, similar to variables declared within the parallel region. Each copy will be initialised to the value of the original at the start of the parallel region.

Shared: Specifies that the listed variables should be shared by all threads. This is what normally happens, but it's often useful to explicitly specify such things for your own understanding.

Default: This is used to override the default scope assigned to any variables declared outside the parallel region – but used within it – that are not explicitly scoped using private or shared. Setting the default to none means that no assumptions are made by the compiler, and that you must explicitly specify the scope of all variables used inside the parallel section.

- The OpenMP C++ specification does not include private as an option for the default modifier (although it does for Fortran). However several implementations allow it - you can test your compiler to see.

Personally, I recommend that while you're learning OpenMP you always use the default(none) clause with

Personally, I recommend that while you're learning OpenMP you always use the default(none) clause with your parallel constructs - it forces you to think about every variable's use, which can reduce the number of bizarre compiler errors you'll get.

Let's see these modifiers in action…

Example 3:

```
void example3() {
 //tell OpenMP to use 6 threads in parallel regions
 omp_set_num_threads(6);

 int i = 0;
 //the default clause means we're forced to specify the scope for i.
 //here it's private. Note that as an object, even cout must be scoped!
 #pragma omp parallel
 default (none) private(i) shared(cout) {
  cout << "++i = " << ++i << endl;
 } //end of parallel region

 cout << "After parallel region, i =" << i << endl;
}
```

There are two important things to notice in this example. Although each copy of i is correctly incremented from 0 to 1, after the parallel region finishes the original remains unaltered; every thread makes its own copy of the variable. Secondly, since cout is actually an object declared outside the parallel region, using it inside the parallel region requires a scope to be set.

Let's look at one more clause this section:

If: From the syntax for the parallel construct, we see that this clause takes not a list of variable names, but a 'scalar expression'. A scalar expression, in this context, is anything in which you could use a normal C++ if as a test. This clause allows you to set a condition deciding whether or not to actually enable parallelisation in the following block of code. Basically, if the expression evaluates to be zero then the code will run with no parallelisation just as if the #pragma omp parallel were not there - meaning you have a way at run-time to disable the parallel construct based on whatever conditions you deem important.

You might use this to easily compare perfomance with parallelisation enabled/disabled, for example.

Example 4:

```
void example4() {
 omp_set_num_threads(2);

 for (int i = 0; i < 3; ++i) {
  //the default clause means we're forced to specify the scope for i.
  //here we've made it private. Note that even cout must be scoped!
  cout << endl << "Thread listing for i=" << i << ':' << endl;
  #pragma omp parallel
  default (none) shared(i) shared(cout) if (i) {
   cout << "Thread " << omp_get_thread_num() << endl;
  } //end of parallel region
 }
}
```

Results from Example 4:
Thread listing for i=0:
Thread 0

Thread listing for i=1:
Thread 0
Thread 1

Thread listing for i=2:
Thread 0
Thread 1

Synchronisation Directives
In the previous section, I warned you that updating a shared variable within a parallel region could be dangerous. You can read from shared variables simultaneously from multiple threads without risk, but writing to them leaves you open to subtle bugs that are hard to reproduce and can be even harder to debug.

The problem is that most operations, even simple ones, are not atomic. An operation has the property of

atomicity if it can be performed in a single CPU instruction. For instance, consider the expression:
`X+=5;`
The way the compiler translates this into machine codes to be run by the CPU is likely to produce some process like this:

1. Load the value from X’s memory location into a CPU register.
2. Add 5 to the contents of the register.
3. Store the contents of the register back to X’s memory location.
   Because this is not done as a single instruction, two threads which are unlucky enough to process this expression at the same time may cause the following situation (bold for thread 1, italic for thread 2). Let’s take X=2 initially; the expression is being run twice, so the expected result is that X=12:
4. Thread 1 loads X (2) into a CPU register.
5. Thread 1 adds 5 to the contents of the register (2). So the register holds 7, X is still 2.
6. Thread 2 loads X (2) into a CPU register.
7. Thread 1 stores the contents of the register (7) into X. So X=7.
8. Thread 2 adds 5 to the contents of the register (2). So the register holds 7.
9. Thread 2 stores the contents of the register (7) into X. So X=7.
   If you happen to get a bug like this in your code, you may not be able to reproduce it - the threads must both be at just the right point, which is unlikely. This means that diagnosing the problem can be difficult And because the timing of the threads is so critical, running it through a debugger or putting in some calls to printf() around the buggy part of the code can often stop the bug from happening!

Synchronisation bugs like this are notoriously difficult to diagnose. For this reason, you need to be doubly careful not to introduce them in the first place. Luckily, OpenMP has some pretty simple tools to aid usÉ

The solution to synchronisation problems is to prevent multiple threads from simultaneously executing code to write to the same memory location. There are two ways to accomplish this: either make the operation atomic, so that there are no local copies of data made, or modify a section of code so that multiple threads are banned from running it at the same time. The latter technique is termed a critical section.

#pragma omp atomic
This directive applies to the immediately following line of code, not to a code block. For example in the potentially buggy code we looked at, we can make it safe like this:

```
#pragma omp atomic
X += 5;
```

The atomic directive is understandably strict in what expressions may be used - only those for which suitable atomic machine code instructions exist. From the OpenMP specification, the only valid expressions are:

- ++X & X++
- –X & X–
- X binary-op=expr, where expr is independent of X. e.g. X+=5, X&=Y
  For an expression to be valid, all values must be of scalar type - you can use int, float, bool, int *, etc., but you can't use objects implementing operator+(), for example. Basically, very simple operations on single values are allowed, and everything else is forbidden.

#pragma omp critical
This directive instructs OpenMP to treat the following code block as a critical section - a block which only one thread may enter at a time. If one thread is executing code from a critical section and another thread reaches the start of that section, it is blocked – i.e. it is kept waiting – until the first thread exits the section. Only then is the second thread allowed to continue into the critical section.

Using a critical section for our previous example we would see something like this:

```
#pragma omp critical CRIT_1 {
  X += 5;
}
```

The critical directive allows an optional name field to be given - CRIT_1 in our example. This is important because OpenMP treats all critical sections with the same name as being part of the same critical section - if we have 10 critical sections named CRIT_1 and a thread enters one of them, any thread trying to enter any of the sections will be blocked until the first thread exits the critical section. Note that all un-named critical sections are treated as shared, too.

There are times when you might want to share a section, and others when you definitely do not. In our example all expressions modifying the shared variable X should be in critical sections of the same name. But if another variable Y is also being modified in our parallel region, we shouldn't prevent threads from modifying X & Y at the same time since that is needless, and slow.

A very common use for a nameless critical section would be around output operations to a file. For instance, when two threads try to use cout simultaneously, the character output often get jumbled up. Placing such code within a critical section ensures that the whole string is output without interruption from other threads.

```
#pragma omp critical {
  cout << "Hello from thread" << omp_get_thread_num() << endl;
}
```

CONSIDERATIONS WITH SYNCHRONISATION

There are a few simple but important performance issues to bear in mind when using synchronisation techniques:

- Any synchronisation technique is slower than no synchronisation - be sure that you really need it.
- An atomic operation is preferable to wrapping the operation within a critical section, since the compiler can optimise the operation to a single instruction.
- If you have several operations that need to be 'safe,' you may need to experiment to find which technique is faster. For instance, which is better.

```
 #pragma omp atomic
X += 5;
#pragma omp atomic
Y *= X;
#pragma omp atomic
Y %= 4;
```

Or:

```
 #pragma omp critical {
  X += 5;
  Y = (Y * X) % 4;
}
```

- When using critical sections, make sure to name them so that only sections modifying the same variables share the same name.
  #pragma omp for
  OpenMP's for construct is almost certainly going to be the main reason you'll want to use OpenMP. It is one of OpenMP's work-sharing constructs, which allow the team of threads to do something more interesting than simply run the exact same code more than once in parallel. What this construct does is to parallelise a C++ for loop, running several iterations of the loop simultaneously, one per thread.

Let's see a trivial example:

Example 5:

```
void example5() {
  omp_set_num_threads(4);

  int val[8];
  #pragma omp parallel
  default (none) shared(val) {
    //we'll use our 4 threads to calculate & store the cubes of 1 - 9
    #pragma omp
    for
    for (int i = 1; i < 9; ++i) {
      val[i - 1] = i * i * i;
      int thread = omp_get_thread_num();
      //use a critical section for i/o
      #pragma omp critical CRIT_1 {
        cout << i << "^3=" << val[i - 1] << ", using thread " << thread << en
      }
    }
  }
```

```
    }

}
```

Results from Example 5:

```
1^3=1, using thread 0
2^3=8, using thread 0
3^3=27, using thread 1
4^3=64, using thread 1
5^3=125, using thread 2
6^3=216, using thread 2
7^3=343, using thread 3
8^3=512, using thread 3
```

As the example shows, we first declare our parallel region as normal, then inside this region we declare that the for loop should be parallelised. When the code is run, the iterations of the loop are shared among the team of threads. Pretty simple, no? For just a few extra lines of code we've accomplished something that would require substantial effort using normal threading methods.

You'll note I've used a critical section around the output to keep it from getting garbled. You may also have noticed that despite the shared variable val being modified, no synchronisation is used there. That's because synchronisation is only needed when the same memory is being modified, and in Example 5 it should be pretty clear that each iteration can only write one element of the array - the fact that it's an array is of no consequence. Of course, if the same elements could be written to by more than one iteration of the loop, then multiple threads could modify the same memory and then synchronisation would become important.

The main issue when trying to parallelise a loop is to determine if the iterations are independent. If any iteration of the loop depends on the result of an earlier iteration then it's probably a non-starter. For instance, consider the following:

```
                for (int i = 1; i < 9; ++i) {
                  val[i - 1] = i * i * i;
                  for (int i = 1; i < 8; ++i) {
                    val[i] = val[i - 1] * 2;
```

In this new case, val[i-1] must be calculated before val[i]. The worst-case scenario is that the application blindly goes ahead and does what you told it to - and your values are total rubbish.

When I actually tried to get this to happen, though, I did get the expected values. I can only assume that the particular example I used was simple enough that the particular compiler I built it with could figure out what was going on, and made the threads run in the right order. As you might imagine, you DO NOT want to trust that this will always be the case. If changing the order in which the iterations of your loop are run could mess things up, it's not a candidate for parallelisation – unless you can find a way to remove the dependencies between iterations.

At a high level it's hard to find much more to say on this construct - for something so powerful it really is amazingly simple to use. If you've done some multithreaded programming previously I'm sure you'll agree – after messing around with thread objects in the past I simply fell in love with OpenMP!

0 **0**

ABOUT
About Topcoder

EXPERTISE
Data Science
Design
Development
QA

CUSTOMER
Why Topcoder
Challenge Model
Full Service
Self Service
TopCrowd
Success Stories
Partners
Security
FAQ

INDUSTRIES
BFSI
Communications
Energy / Utilities
Health / Pharma
Public Sector
Retail
Technology

MEDIA
Blog
Newsletter
Press Room
Videos
Whitepapers
Releases

TALENT
Learn
Earn
Compete
Connect
Benefits
Statistics
Getting Paid
Topcoder Open
FAQ

CONTACT
Talk to Sales
Book a demo
Support
Report a Bug

CAREERS
Work at Topcoder

LEGAL
Privacy Policy