msdn magazine

OPENMP AND C++

Reap the Benefits of Multithreading without All the Work

Kang Su Gatlin
and Pete Isensee

This article is based on a prerelease version of Visual C++ 2005. All information contained herein is subject to change.

This article discusses:

- OpenMP features for multithreaded applications
- OpenMP pragmas
- OpenMP runtime routines

This article uses the following technologies:
Visual C++ 2005, OpenMP

Among those in the parallel computation field, a common joke is "Parallel computing is the wave of the future...and always will be." This little joke has held true for decades. A similar sentiment was felt in the computer architecture community, where an end of processor clock speed growth seemed always on the horizon, but the clocks kept getting faster. The multicore revolution is the collision of this parallel community optimism and architecture community pessimism.

The major CPU vendors have shifted gears away from ramping up clock speeds to adding parallelism support on-chip with multicore processors. The idea is simple: put more than one CPU core on a single chip. This effectively makes a system with a processor with two cores operate just like a dual-processor computer, and a system with a processor with four cores operate like a quad-processor computer. This practice avoids many of the technological hurdles CPU vendors are encountering in boosting speeds, while still offering a better-performing processor.

So far this seems like pretty good news, but if your application does not take advantage of these multiple cores, it may not be able to operate any faster. This is where OpenMP comes into the picture. OpenMP helps C++ developers create multithreaded applications more quickly.

Tackling the topic of OpenMP in a single article is a daunting task, as it is a large and powerful API. Therefore, this article will simply serve as an initial introduction, demonstrating how you can use the various features of OpenMP to write multithreaded applications quickly. If you'd like to take a look at further information on this topic, we recommend the surprisingly readable specification available from the OpenMP Web site.

Enabling OpenMP in Visual C++

The OpenMP standard was formulated in 1997 as an API for writing portable, multithreaded applications. It started as a Fortran-based standard, but later grew to include C and C++. The current version is OpenMP 2.0, and Visual C++® 2005 supports the full standard. OpenMP is also supported by the Xbox 360™ platform.

Before we get into the fun of playing with code, you'll need to know how to turn on the OpenMP capabilities of the compiler. Visual C++ 2005 provides a new /openmp compiler switch that enables the compiler to understand OpenMP directives. (You can also enable OpenMP directives in the property pages for your project. Click Configuration Properties, then C/C++, then Language, and modify the OpenMP Support property.) When the /openmp switch is invoked, the compiler defines the symbol _OPENMP, which may be used to detect when OpenMP is enabled using #ifndef _OPENMP.

OpenMP is linked to applications through the import lib vcomp.lib. The corresponding runtime library is vcomp.dll. The debug versions of the import lib and runtime (vcompd.lib and vcompd.dll, respectively) have additional error messages that are emitted during certain illegal operations. Note that Visual C++ does not support static linking of the OpenMP runtime, although static linking is supported in the version for Xbox 360.

Parallelism in OpenMP

An OpenMP application begins with a single thread, the master thread. As the program executes, the application may encounter parallel regions in which the master thread creates thread teams (which include the master thread). At the end of a parallel region, the thread teams are parked and the master thread continues execution. From within a parallel region there can be nested parallel regions where each thread of the original parallel region becomes the master of its own thread team. Nested parallelism can continue to further nest other parallel regions.

Figure 1 illustrates how OpenMP parallelism works. The left-most, yellow line is the master thread. This thread runs as a single-threaded application until it hits its first parallel region at point 1. At the parallel region the master thread creates a thread team (consisting of the yellow and orange lines), and these threads now all run concurrently in the parallel region.

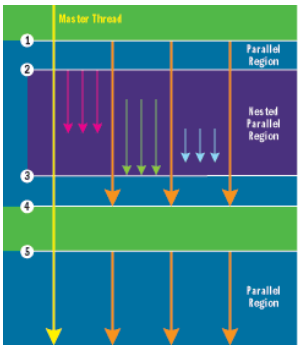

Figure 1 OpenMP Parallel Sections

At point 2, three of the four threads running in the parallel region create new thread teams (pink, green, and blue) in this nested parallel region. The yellow and orange threads that created the teams are the masters of their respective teams. Note that each thread could create a new team at a different point in time, or it may not encounter a nested parallel region at all.

At point 3, the nested parallel regions are finished. Each nested parallel region synchronizes with the other threads in the region, but notice that they do not synchronize across regions. Point 4 ends the first parallel region, and point 5 begins a new parallel region. In the new parallel region at point 5, the thread-local data for each thread is persisted from the previous parallel region.

Now that you have a basic understanding of the execution model, let's examine how you can actually begin making a parallel application.

OpenMP Constructs

OpenMP is easy to use and consists of only two basic constructs: pragmas and runtime routines. The OpenMP pragmas typically direct the compiler to parallelize sections of code. All OpenMP pragmas begin with #pragma omp. As with any pragma, these directives are ignored by compilers that do not support the feature—in this case, OpenMP.

OpenMP runtime routines are used primarily to set and retrieve information about the environment. There are also APIs for certain types of synchronization. In order to use the functions from the OpenMP runtime, the program must include the OpenMP header file omp.h. If the application is only using the pragmas, you can omit omp.h.

You add parallelism to an app with OpenMP by simply adding pragmas and, if needed, using a set of OpenMP function calls from the OpenMP runtime. These pragmas use the following form:

```
#pragma omp <directive> [clause[ [,] clause]...]
```

The directives include the following: parallel, for, parallel for, section, sections, single, master, critical, flush, ordered, and atomic. These directives specify either work-sharing or synchronization constructs. We will cover the majority of these directives in this article.

The clauses are optional modifiers of the directives and affect the behavior of the directives. Each directive has a different set of clauses available to it, and five of the directives (master, critical, flush, ordered, and atomic) do not accept clauses at all.

Specifying Parallelism

Although there are many directives, it's easy to get started knowing just a few. The most common and important directive is parallel. This directive creates a parallel region for the dynamic extent of the structured block that follows the directive. For example:

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

This directive tells the compiler that the structured block of code should be executed in parallel on multiple threads. Each thread will execute the same instruction stream, however not necessarily the same set of instructions. This will depend on control-flow statements such as if-else.

Here is an example using the obligatory "Hello World" program:

```
#pragma omp parallel
{
    printf("Hello World\n");
}
```

On a two-processor system you may expect output like this:

```
Hello World
```

```
Hello World
```

However, you may also get something like this:

```
HellHell oo WorWlodrl
d
```

This second output occurs because the two threads are running in parallel and are both attempting to print at the same time. Whenever more than one thread attempts to read or modify a shared resource (in this case the shared resource is the console window), there is potential for a race condition. These are nondeterministic bugs in your app and are often hard to find. It's the programmer's responsibility to make sure that these don't happen, usually by using locks or avoiding shared resources as much as possible.

Let's look at an example that does something a bit more useful—it computes the average of two values from one array and puts the value into another. Here we introduce a new OpenMP construct: #pragma omp for. This is a work-sharing directive. Work-sharing directives do not create parallelism, but rather distribute the thread team in a logical way to implement the following control-flow construct. The #pragma omp for work-sharing directive tells OpenMP that the for loop that is shown in the following code, when called from a parallel region, should have its iterations divided among the thread team:

```
#pragma omp parallel
{
#pragma omp for
for(int i = 1; i < size; ++i)
        x[i] = (y[i-1] + y[i+1])/2;
}
```

In this case, if size had the value 100 and this were run on a computer with four processors, the iterations of the loop might get allocated such that processor 1 gets iterations 1-25, processor 2 gets iterations 26-50, processor 3 gets iterations 51-75, and processor 4 gets iterations 76-99. This assumes a scheduling policy called static scheduling. We'll discuss scheduling policies in more depth later in this article.

One thing that is not explicit in this program is a barrier synchronization at the end of the parallel region. All threads will block there until the last thread completes.

If the code shown previously did not use #pragma omp for, then each thread would execute the complete for loop, and as a result each thread would do redundant work:

```
#pragma omp parallel // probably not what was intended
{
for(int i = 1; i < size; ++i)
        x[i] = (y[i-1] + y[i+1])/2;
}
```

Since parallel loops are the most common parallelizable work-sharing construct, OpenMP provides a shorthand way to write #pragma omp parallel followed by #pragma omp for:

```
#pragma omp parallel for
for(int i = 1; i < size; ++i)
    x[i] = (y[i-1] + y[i+1])/2;
```

Notice that there are no loop-carried dependencies. This means one iteration of the loop does not depend upon the results of another iteration of the loop. For example, the following two loops have two different loop-carried dependent properties:

```
for(int i = 1; i <= n; ++i)    // Loop (1)
    a[i] = a[i-1] + b[i];

for(int i = 0; i < n; ++i)     // Loop (2)
    x[i] = x[i+1] + b[i];
```

Parallelizing loop 1 is problematic because, in order to execute iteration i of the loop, you need to have the result of iteration i-1. There is a dependency from iteration i to i-1. Parallelizing loop 2 is also problematic, although for a different reason. In this loop you can compute the value of x[i] before computing the value of x[i-1], but in doing so you can no longer compute the value of x[i-1]. There is a dependency from iteration i-1 to i.

When parallelizing loops, you must make sure that your loop iterations do not have dependencies. When there are no loop-carried dependencies, the compiler can execute the loop in any order, even in parallel. This is an important requirement that the compiler does not check. You are effectively asserting to the compiler that the loop being parallelized does not contain loop-carried dependencies. If it turns out that the loop does have dependencies and you told the compiler to parallelize it, the compiler will do as it was told and the end result will be a bug.

Additionally, OpenMP does place restrictions on the form of for loops that are allowed inside of a #pragma omp for or #pragma omp parallel for block. The loops must have the following form:

```
for([integer type] i = loop invariant value;
    i {<,>,=,<=,>=} loop invariant value;
    i {+,-}= loop invariant value)
```

This is required so that OpenMP can determine, at entry to the loop, how many iterations the loop will perform.

Comparing OpenMP and Win32 Threads

It's instructive to compare the #pragma omp parallel for example shown earlier with what would be necessary when using the Windows® API to do threading. As you can see in **Figure 2**, there is a lot more code required to accomplish the same result, and there's even some magic taking place behind the scenes here. For example, the constructor for ThreadData is figuring out what start and stop should be for each thread invocation. OpenMP automatically handles all of these details, and additionally gives the programmer some more knobs to turn to configure their parallel regions and code.

Shared Data and Private Data

In writing parallel programs, understanding which data is shared and which is private becomes very important, not only to performance, but also for correct operation. OpenMP makes this distinction apparent to the programmer, and it is something that you can tweak manually.

Shared variables are shared by all the threads in the thread team. Thus a change of the shared variable in one thread may become visible to another thread in the parallel region. Private variables, on the other hand, have private copies made for each thread in the thread team, so changes made in one thread are not visible in the private variables in the other threads.

By default, all the variables in a parallel region are shared, with three exceptions. First, in parallel for loops, the loop index is private. In the example in **Figure 3**, the i variable is private. The j variable is not private by default, but it's made private explicitly with the use of the firstprivate clause.

Second, variables that are local to the block of the parallel region are private. In **Figure 3**, the variable doubleI is private because it is declared in the parallel region. Any non-static and non-member variables declared in myMatrix::GetElement will be private.

And third, any variables listed in the private, firstprivate, lastprivate, or reduction clauses are private. In **Figure 3**, the variables i, j, and sum are made private to each thread in the thread team. This is done by making a distinct copy of each of these variables for each thread.

Each of the four clauses takes a list of variables, but their semantics are all different. The private clause says that each variable in the list should have a private copy made for each thread. This private copy will be initialized with its default value (using its default constructor where appropriate). For example, the default value for variables of type int is 0.

The firstprivate clause has the same semantics as private, except it copies the value of the private variable before the parallel region into each thread, using the copy constructor where appropriate.

The lastprivate clause has the same semantics as private, except the last iteration or section of the work-sharing construct causes the values of the variables listed in the lastprivate clause to be assigned to the variable of the master thread. When appropriate, the copy assignment operator is used to copy objects.

The reduction clause has similar semantics to private, but it takes both a variable and an operator. The set of operators are limited to the operators listed in **Figure 4**, and the reduction variable must be a scalar variable (such as float, int, or long, but not std::vector, int [], and so on). The reduction variable is initialized to the value listed in the table for each thread. At the end of the code block, the reduction operator is applied to each of the private copies of the variable, as well as to the original value of the variable.

In the example in **Figure 3**, the sum value is implicitly initialized in each thread with the value 0.0f. (Note that the canonical value in the table is 0, which becomes 0.0f since the type of sum is float.) After the #pragma omp for block is completed, the threads apply the + operation to all the private sum values and the original value (the original value of sum in this example is 10.0f). The result is assigned to the original shared sum variable.

Non-Loop Parallelism

OpenMP is typically used for loop-level parallelism, but it also supports function-level parallelism. This mechanism is called OpenMP sections. The structure of sections is straightforward and can be useful in many instances.

Consider one of the most important algorithms in computer science, the quicksort. The example that is used here is a simple recursive quicksort algorithm for a list of integers. For simplicity, we have not used a generic templatized version, but the same OpenMP ideas would apply. The code in **Figure 5** shows the main quicksort function using sections (we've omitted the Partition function for the purpose of simplicity).

In this example, the first #pragma creates a parallel region of sections. Each section is preceded by the directive #pragma omp section. Each section in the parallel region is given to a single thread in the thread team, and all of the sections can be handled concurrently. Each parallel section calls QuickSort recursively.

As in the #pragma omp parallel for construct, you are responsible for ensuring that each section is independent of the other sections so they can be executed in parallel. If the sections update shared resources without synchronizing access to those resources, the results are undefined.

Note that this example uses the shorthand #pragma omp parallel sections in a way analogous to #pragma omp parallel for. You can also use #pragma omp sections within a parallel region as a standalone directive, just as you can with #pragma omp for.

There are a few things to note about the implementation in **Figure 5**. First, the parallel sections are called recursively. Recursive calls are supported with parallel regions and, in this case, specifically with parallel sections. Also, with nesting enabled, new threads will be spawned as the program recursively calls QuickSort. This may or may not be what the application programmer desires, as it can result in quite a lot of threads. The program can disable nesting to bound the number of threads. With nesting disabled, this application will recursively call QuickSort on two threads, and will not spawn more than two threads, even recursively.

In addition, compiling this application without the /openmp switch will generate a perfectly correct serial implementation. One of the benefits of OpenMP is that it coexists with compilers that don't understand OpenMP.

Synchronization Pragmas

With multiple threads running concurrently, there are often times when it's necessary to have one thread synchronize with another thread. OpenMP supplies multiple types of synchronization to help in many different situations.

One type is implied barrier synchronization. At the end of each parallel region is an implied barrier synchronization of all threads within the parallel region. A barrier synchronization requires all threads to reach that point before any can proceed beyond it.

There is also an implied barrier synchronization at the end of each #pragma omp for, #pragma omp single, and #pragma omp sections block. To remove the implied barrier synchronization from these three types of work-sharing blocks, add the nowait clause:

```
#pragma omp parallel
{
    #pragma omp for nowait
    for(int i = 1; i < size; ++i)
        x[i] = (y[i-1] + y[i+1])/2;
}
```

As you can see, the nowait clause on the work-sharing directive indicates that the threads do not have to synchronize at the end of the for loop, although the threads will have to synchronize at the end of the parallel region.

Another type is explicit barrier synchronization. In some situations you'd like to place a barrier synchronization in a place besides the exit of a parallel region. This is placed in the code with #pragma omp barrier.

Critical sections can be used as barriers. In the Win32® API, a critical section is entered and exited through EnterCriticalSection and LeaveCriticalSection. OpenMP gives the same capability with #pragma omp critical [name]. This has the same semantics as the Win32 critical section, and EnterCriticalSection is used under the covers. You can use a named critical section, in which case the block of code is only mutually exclusive with respect to other critical sections with the same name (this applies across the whole process). If no name is specified, then the implementation maps to some user-unspecified name. These unnamed critical sections are all mutually exclusive regions with respect to each other.

In a parallel region, there are often sections of code where limiting access to a single thread is desired, such as when writing to a file in the middle of a parallel region. In many of these cases it does not matter which thread executes this code, as long as it is just one thread. OpenMP has #pragma omp single to do this.

Sometimes specifying single-threaded execution for a section of code in a parallel region with the single directive is not sufficient. On occasion you may want to ensure that the master thread is the thread that executes the section of code—for example, if the master thread is the GUI thread and you need the GUI thread to do some work. #pragma omp master does this. Unlike single, there is no implied barrier at entry or exit of a master block.

Memory fences are implemented with #pragma omp flush. This directive creates a memory fence in the program, which is effectively equivalent to the _ReadWriteBarrier intrinsic.

Note that the OpenMP pragmas must be encountered by all threads in the thread team in the same order (or none at all). Thus, the following code snippet is illegal and has undefined runtime behavior (crashing or hanging would not be out of the ordinary in this particular situation):

```
#pragma omp parallel
{
    if(omp_get_thread_num() > 3)
    {
        #pragma omp single    // May not be accessed by all threads
        x++;
    }
}
```

Execution Environment Routines

Along with the pragmas discussed earlier, OpenMP also has a set of runtime routines that are useful for writing OpenMP applications. There are three broad classes of routines available: execution environment routines, lock/synchronization routines, and timing routines. (The timing routines are not discussed in this article.) All of the OpenMP runtime routines begin with omp_ and are defined in the header file omp.h.

Functionality provided by the execution environment routines allows you to query or set various aspects of the operating environment for OpenMP. Functions that begin with omp_set_ may only be called outside of parallel regions. All other functions can be used in both parallel and non-parallel regions.

In order to get or set the number of threads in the thread team, use the routines omp_get_num_threads and omp_set_num_threads. omp_get_num_threads will return the number of threads in the current thread team. If the calling thread is not in a parallel region, it will return 1. omp_set_num_thread will set the number of threads that should be used for the next parallel region that the thread will encounter.

But that's not the whole story when it comes to setting the number of threads. The number of threads used for parallel regions also depends on two other aspects of the OpenMP environment: dynamic threads and nesting.

Dynamic threading is a Boolean property that is disabled by default. If this property is disabled when a thread encounters a parallel region, the OpenMP runtime creates the thread team with the number of threads as specified by omp_get_max_threads. omp_get_max_threads by default is set to the number of hardware threads on the computer or the value of the OMP_NUM_THREADS environment variable. If dynamic threading is enabled, the OpenMP runtime will create the thread team with a variable number of threads that does not exceed omp_get_max_ threads.

Nesting is another Boolean property that is disabled by default. Nesting of parallel regions occurs when a thread that is already in a parallel region encounters another parallel region. If nesting is enabled, then a new thread team will be formed with the rules specified in the previous dynamic threads section. If nesting is not enabled, then the thread team is formed with just that single thread.

The status of dynamic threads and nesting is set and queried with the runtime routines omp_set_dynamic, omp_get_dynamic, omp_set_nested, and omp_get_nested. Each thread can also query its environment. A thread can find out what its thread number is from within the thread team with the omp_get_thread_num call. This is not the Windows thread ID, but rather a value from 0 to 1 less than omp_get_num_threads.

A thread can find out if it is currently executing in a parallel region with omp_in_parallel. And with omp_get_num_procs, a thread can find out how many processors are in the computer.

To help make some of the interactions between the various environment routines more clear, you should take a look at the code in **Figure 6**. In this example, there are four distinct parallel regions, with two nested parallel regions.

Running on a typical dual-processor computer, a program compiled using Visual Studio 2005 prints the following from this code:

```
Num threads in dynamic region is = 2

Num threads in non-dynamic region is = 10

Num threads in nesting disabled region is = 1
Num threads in nesting disabled region is = 1

Num threads in nested region is = 2
Num threads in nested region is = 2
```

The first parallel region enabled dynamic threads and set the number of threads to 10. Looking at the output of the program, you can see that with dynamic threads enabled the OpenMP runtime decided to allocate only two threads to the thread team, since the computer has two processors. In the second parallel region, the OpenMP runtime allocated 10 threads to the thread team because dynamic threads were not enabled.

In the third and fourth parallel regions, you see the effect of having nesting enabled or disabled. In parallel region three, we disabled nesting, and in this case no new threads were allocated for the nested parallel regions. Thus there was a total of two threads for both the outer and nested parallel regions. In parallel region four, where we enabled nesting, the nested parallel region created a thread team with

two threads (a total of four threads in the nested parallel region). This process of doubling threads for each nested parallel region can continue until you run out of stack space. In practice, you can generate several hundred threads, though the overhead can easily outweigh any performance benefits.

As you may have noticed, for parallel regions three and four, dynamic threading was enabled. What if the code was executed with dynamic threads disabled, as shown here:

```
omp_set_dynamic(0);
omp_set_nested(1);
omp_set_num_threads(10);
#pragma omp parallel
{
    #pragma omp parallel
    {
        #pragma omp single
        printf("Num threads in nested region is = %d\n",
        omp_get_num_threads());
    }
}
```

Here you can see the expected behavior. The first parallel region started with a thread team of 10 threads. The subsequent nested parallel region started with 10 threads for each of the 10 threads in the enclosing parallel region. Thus a total of 100 threads executed within the nested parallel region:

```
Num threads in nested region is = 10
Num threads in nested region is = 10
Num threads in nested region is = 10
Num threads in nested region is = 10
Num threads in nested region is = 10
Num threads in nested region is = 10
Num threads in nested region is = 10
Num threads in nested region is = 10
Num threads in nested region is = 10
Num threads in nested region is = 10
```

Synchronization/Lock Routines

OpenMP has runtime routines to assist in synchronization of code. There are two lock types, simple and nestable, each of which can exist in one of three states: uninitialized, locked, and unlocked.

Simple locks (omp_lock_t) can not be locked more than once, even by the same thread. Nestable locks (omp_nest_lock_t) are identical to simple locks, except when a thread attempts to set a lock that it already holds, it will not block. In addition, nestable locks are reference counted and keep track of how many times they have been set.

There are synchronization routines that act on these locks. For each routine there is a simple lock and a nestable lock variant. There are five actions that can be performed on the lock: initialize, set (acquire lock), unset (release lock), test, and destroy. These all map very closely to Win32 critical section routines—in fact OpenMP is implemented as a thin wrapper on top of these routines. **Figure 7** shows the mapping from the OpenMP routines to the Win32 routines.

Developers can choose to use either the synchronization runtime routines or the synchronization pragmas. The advantage of the pragmas is that they are extremely structured. This can make them easier to understand, and it makes it easier to look at the program to determine the location of the entrance or exit of your synchronized regions.

The advantage of the runtime routines is their flexibility. You can pass a lock to a different function and inside of this other function the lock can be set or unset. This is not possible with the pragmas. In general it makes sense to use synchronization pragmas unless you need a level of flexibility that is only available with the runtime routines.

Parallel Execution of Data Structure Traversal

The code in **Figure 8** shows two examples of executing a parallel for loop where the runtime does not know the number of iterations it will perform when it first encounters the loop. The first example is the traversal of an STL std::vector container, and the second is a standard linked list.

For the STL container portion of the example, every thread in the thread team executes the for loop and each thread has its own copy of the iterator. But only one thread per iteration enters the single block (those are the semantics of single). The OpenMP runtime handles all of the magic that ensures that the single block is executed once and only once for each iteration. There is considerable runtime overhead to this method of iteration, so it's only useful when there's a lot of work happening in the process function. The linked list example follows the same logic.

It's worth pointing out that for the STL vector example we can rewrite the code to use std::vector.size to determine the number of iterations needed prior to entering the loop, which then puts the loop in the canonical OpenMP for loops form. This is demonstrated in the following code:

```
#pragma omp parallel for
   for(int i = 0; i < xVect.size(); ++i)
      process(xVect[i]);
```

Because this method has much lower OpenMP runtime overhead, we recommend it for arrays, vectors, and any other container that can be walked in canonical OpenMP for loop form.

Advanced Scheduling Algorithms

By default, OpenMP parallelized for loops are scheduled with an algorithm called static scheduling. This means that each thread in the thread team is given an equal number of iterations. If there are n iterations and T threads in the thread team, each thread will get n/T iterations (and yes, OpenMP correctly handles the case when n is not evenly divisible by T). But OpenMP also offers a few other scheduling mechanisms which are appropriate for many situations: dynamic, runtime, and guided.

To specify one of these other scheduling mechanisms use the schedule clause with the #pragma omp for or #pragma omp parallel for directives. The schedule clause has the form:

```
schedule(schedule-algorithm[, chunk-size])
```

Here are some examples of the pragma in use:

```
#pragma omp parallel for schedule(dynamic, 15)
for(int i = 0; i < 100; ++i)
{ ...

#pragma omp parallel
   #pragma omp for schedule(guided)
```

Dynamic scheduling schedules each thread to execute the number of threads specified by chunk-size (if chunk-size is not given it defaults to 1). After a thread has finished executing the iterations given to it, it requests another set of chunk-size iterations. This continues until all of the iterations are completed. The last set of iterations may be less than chunk-size.

Guided scheduling schedules each thread to execute the number of threads proportional to this formula:

```
iterations_to_do = max(iterations_not_assigned/omp_get_num_threads(),
                       chunk-size)
```

After a thread has finished executing the iterations given to it, the thread requests another set of iterations based on the iterations_to_do formula. Thus the number of iterations assigned to each thread decreases over time. The last set of iterations scheduled may be less than the value defined by the iterations_to_do function.

Here you can see the process by which four threads might get scheduled for a for loop with 100 iterations using #pragma omp for schedule(dynamic, 15):

```
Thread 0 gets iterations 1-15
Thread 1 gets iterations 16-30
Thread 2 gets iterations 31-45
Thread 3 gets iterations 46-60
Thread 2 finishes
```

```
Thread 2 gets iterations 61-75
Thread 3 finishes
Thread 3 gets iterations 76-90
Thread 0 finishes
Thread 0 gets iterations 91-100
```

Here is how four threads might get scheduled for a for loop with 100 iterations using #pragma omp for schedule(guided, 15):

```
Thread 0 gets iterations 1-25
Thread 1 gets iterations 26-44
Thread 2 gets iterations 45-59
Thread 3 gets iterations 60-64
Thread 2 finishes
Thread 2 gets iterations 65-79
Thread 3 finishes
Thread 3 gets iterations 80-94
Thread 2 finishes
Thread 2 gets iterations 95-100
```

Dynamic scheduling and guided scheduling are both perfect mechanisms for those situations where each iteration has variable amounts of work or where some processors are faster than others. With static scheduling there is no way to load-balance the iterations. Dynamic and guided scheduling automatically load-balance the iterations, by the very nature of how they work. Typically, guided scheduling performs better than dynamic scheduling due to less overhead associated with scheduling.

The last scheduling algorithm, runtime scheduling, really isn't a scheduling algorithm at all, but rather a way to dynamically select among the three previously mentioned scheduling algorithms. When runtime is specified in the schedule clause the OpenMP runtime uses the scheduling algorithm specified in the OMP_SCHEDULE environment variable for this particular for loop. The format for the OMP_SCHEDULE environment variable is type[,chunk size]. For example:

```
set OMP_SCHEDULE=dynamic,8
```

Using runtime scheduling gives the end-user some flexibility in determining the type of scheduling used, with the default being static scheduling.

When to Use OpenMP

Knowing when to use OpenMP is just as important as knowing how to use it. In general we find the following guidelines quite useful in making the decision.

Target platform is multicore or multiprocessor In these types of cases, if the application is saturating a core or processor, making it into a multithreaded application with OpenMP will almost certainly increase the application's performance.

Application is cross-platform OpenMP is a cross-platform and widely supported API. And since the API is implemented through pragmas, the application can still compile even on a compiler that has no knowledge of the OpenMP standard.

Parallelizable loops OpenMP is at its best parallelizing loops. If the application has loops which have no loop-carried dependencies, using OpenMP is an ideal choice.

Last-minute optimizations needed Because OpenMP does not require re-architecting the application, it is the perfect tool for making small surgical changes to get incremental performance improvements.

With all that said, OpenMP isn't designed to handle every multithreading problem. The roots of OpenMP show some of its biases. It was developed for use by the high-performance computing community and it works best in programming styles that have loop-heavy code working on shared arrays of data.

Just as creating a native thread has a cost, creating OpenMP parallel regions has a cost. In order for OpenMP to give a performance gain, the speedup given by the parallelized region must outweigh the cost of thread team startup. The Visual C++ implementation creates the thread team when encountering the first parallel region. After that, the thread team is parked until it's needed again. Under the covers OpenMP uses the Windows thread pool. **Figure 9** shows the speedup using OpenMP on a two processor box for various numbers of iterations, using the simple example program that is shown at the beginning of this article. Performance peaked at about 1.7x, which is typical on a two-processor system. (To better indicate the performance difference, the y axis shows a ratio scale of serial performance to parallel performance.) Notice that it takes about 5,000 iterations before the parallel version is running as fast as the sequential version, but this is just about the worst-case scenario. Most parallelized loops will be faster than the sequential version in many fewer iterations. It simply depends on the amount of work that each iteration does. However, this graph shows why it's important to measure performance. Using OpenMP doesn't guarantee that performance will improve.

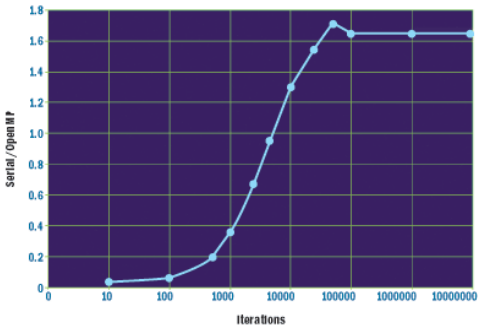

Figure 9 Serial vs. Parallelized Performance on Two Processors

The OpenMP pragmas, while easy to use, don't provide great feedback when an error occurs. If you're writing a mission-critical application that needs to be able to detect faults and gracefully recover from them, then OpenMP is probably not the right tool for you (at least not the current incarnation of OpenMP). For example, if OpenMP cannot create threads for parallel regions or cannot create a critical section then the behavior is undefined. With Visual C++ 2005 the OpenMP runtime continues trying for a period of time before eventually bailing out. One of the things we plan to pursue for future versions of OpenMP is a standard error-reporting mechanism.

Another place where one should tread carefully is when using Windows threads along with OpenMP threads. OpenMP threads are built on top of Windows threads, so they execute just fine in the same process. The problem is that OpenMP has no knowledge of Windows threads that it does not create. This creates two problems: the OpenMP runtime does not keep the other Windows threads as part of its "count" and the OpenMP synchronization routines do not synchronize the Windows threads because they're not part of thread teams.

Common Pitfalls When Using OpenMP in Apps

While OpenMP is about as easy as it gets for adding parallelism to your application, there are still a few things to be aware of. The index variable in the outermost parallel for loop is privatized, but index variables for nested for loops are shared by default. When you have nested loops, you usually want the inner-loop index variables to be private. Use the private clause to specify these variables.

OpenMP applications should use care when throwing C++ exceptions. Specifically, when an application throws an exception in a parallel region, it must be handled in the same parallel region by the same thread. In other words, the exception should not escape the region. As a general rule, if there can be an exception in a parallel region, there should be a catch for it. If the exception isn't caught in the parallel region where it was thrown, the application will typically crash.

The #pragma omp <directive> [clause] statement must end with a new-line, not a brace, to open a structured block. Directives that end with a brace will result in a compiler error:

```
// Bad Bracket
#pragma omp parallel {
   // won't compile Code
}

// Good Bracket
#pragma omp parallel
{
   // Code
}
```

Debugging OpenMP apps with Visual Studio 2005 can sometimes be complicated. In particular, stepping into and/or out of a parallel region with F10/F11 is a less than stellar experience. This is because the compiler creates some additional code to call into the runtime and to invoke the thread teams. The debugger has no special knowledge about this, so the programmer sees what looks like odd behavior. We recommend putting a breakpoint in the parallel region and using F5 to reach it. To exit the parallel region, set a breakpoint outside of the parallel region, and then hit F5.

From within a parallel region the debugger "Threads Window" will show the various threads that are running in the thread team. The thread IDs will not correlate with the OpenMP thread IDs; however, they will reflect the Windows thread ID that OpenMP is built on top of.

OpenMP is currently not supported with Profile Guided Optimization. Fortunately, since OpenMP is pragma-based, you can try compiling your application with /openmp or with PGO and decide which method gives you the best performance improvement.

OpenMP and .NET

High-performance computing and .NET may not be two words that immediately come to mind together, but Visual C++ 2005 makes some strides in this area. One of the things we did was to make OpenMP work with managed C++ code. To do this we made the /openmp switch compatible with both /clr and /clr:OldSyntax. This means you can use OpenMP to get parallelism from methods in .NET types, which are garbage collected. Please note that /openmp is not currently compatible with /clr:safe nor /clr:pure; however, we plan to make them compatible in the future.

There is one noteworthy restriction associated with using OpenMP and managed code. An application that uses OpenMP must only be used in a single application domain program. If another application domain is loaded into a process with the OpenMP runtime already loaded, the application may abort.

OpenMP is a simple yet powerful technology for parallelizing applications. It provides ways to parallelize data processing loops as well as functional blocks of code. It can be integrated easily into existing applications and turned on or off simply by throwing a compiler switch. OpenMP is an easy way to tap into the processing power of multicore CPUs. Again, we strongly suggest reading the OpenMP spec for more details. Have fun multithreading.

Kang Su Gatlin is a Program Manager on the Visual C++ team at Microsoft where he spends most of his day trying to figure out systematic ways to make programs run faster. Prior to his life at Microsoft, he worked on high-performance and grid computing.

Pete Isensee is the Development Manager for the Xbox Advanced Technology Group at Microsoft. He has worked in the game industry for 12 years and speaks regularly at conferences about optimization and performance.

From the October 2005 issue of MSDN Magazine.

Figure 2 Multithreading in Win32

```
class ThreadData {
public:
   ThreadData(int threadNum); // initializes start and stop
   int start;
   int stop;
};

DWORD ThreadFn(void* passedInData)
{
   ThreadData *threadData = (ThreadData *)passedInData;
   for(int i = threadData->start; i < threadData->stop; ++i )
      x[i] = (y[i-1] + y[i+1]) / 2;
   return 0;
}

void ParallelFor()
{
   // Start thread teams
   for(int i=0; i < nTeams; ++i)
      ResumeThread(hTeams[i]);

   // ThreadFn implicitly called here on each thread

   // Wait for completion
   WaitForMultipleObjects(nTeams, hTeams, TRUE, INFINITE);
}

int main(int argc, char* argv[])
{
   // Create thread teams
   for(int i=0; i < nTeams; ++i)
   {
      ThreadData *threadData = new ThreadData(i);
      hTeams[i] = CreateThread(NULL, 0, ThreadFn, threadData,
          CREATE_SUSPENDED, NULL);
   }

   ParallelFor(); // simulate OpenMP parallel for

   // Clean up
   for(int i=0; i < nTeams; ++i)
      CloseHandle(hTeams[i]);
}
```

Figure 3 OpenMP Clauses and a Nested For Loop

```
float sum = 10.0f;
MatrixClass myMatrix;
int j = myMatrix.RowStart();
int i;
#pragma omp parallel
{
   #pragma omp for firstprivate(j) lastprivate(i) reduction(+: sum)
   for(i = 0; i < count; ++i)
   {
      int doubleI = 2 * i;
      for(; j < doubleI; ++j)
      {
         sum += myMatrix.GetElement(i, j);
      }
   }
}
```

Figure 4 Reduction Operators

Reduction operator	Initialized value (canonical value)
+	0
*	1
-	0
&	~0(every bit set)
|	0
^	0
&&	1

||	0

Figure 5 Quicksort with Parallel Sections

```
void QuickSort (int numList[],  int nLower, int nUpper)
   {
   if (nLower < nUpper)
      {
      // create partitions
      int nSplit = Partition (numList, nLower, nUpper);
      #pragma omp parallel sections
      {
         #pragma omp section
         QuickSort (numList, nLower, nSplit - 1);

         #pragma omp section
         QuickSort (numList, nSplit + 1, nUpper);
      }
   }
}
```

Figure 6 Using OpenMP Environment Routines

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

int main()
   {
   omp_set_dynamic(1);
   omp_set_num_threads(10);
   #pragma omp parallel        // parallel region 1
   {
      #pragma omp single
      printf("Num threads in dynamic region is = %d\n",
      omp_get_num_threads());
   }
   printf("\n");
   omp_set_dynamic(0);
   omp_set_num_threads(10);
   #pragma omp parallel        // parallel region 2
   {
      #pragma omp single
      printf("Num threads in non-dynamic region is = %d\n",
      omp_get_num_threads());
   }
   printf("\n");
   omp_set_dynamic(1);
   omp_set_num_threads(10);
   #pragma omp parallel        // parallel region 3
   {
      #pragma omp parallel
      {
         #pragma omp single
         printf("Num threads in nesting disabled region is = %d\n",
          omp_get_num_threads());
      }
   }
   printf("\n");
   omp_set_nested(1);
   #pragma omp parallel        // parallel region 4
   {
      #pragma omp parallel
      {
         #pragma omp single
         printf("Num threads in nested region is = %d\n",
          omp_get_num_threads());
      }
   }
}
```

Figure 7 OpenMP Lock Routines and Win32

OpenMP Simple Lock	OpenMP Nested Lock	Win32 Routine
omp_lock_t	omp_nest_lock_t	CRITICAL_SECTION
omp_init_lock	omp_init_nest_lock	InitializeCriticalSection
omp_destroy_lock	omp_destroy_nest_lock	DeleteCriticalSection
omp_set_lock	omp_set_nest_lock	EnterCriticalSection
omp_unset_lock	omp_unset_nest_lock	LeaveCriticalSection
omp_test_lock	omp_test_nest_lock	TryEnterCriticalSection

Figure 8 Handling Variable Iterations at Run Time

```
#pragma omp parallel
{
   // Traversing an STL vector in parallel
   std::vector<int>::iterator iter;
   for(iter = xVect.begin(); iter != xVect.end(); ++iter)
   {
      #pragma omp single nowait
      {
         process1(*iter);
      }
   }

   // Walking a standard linked-list in parallel
   for(LList *listWalk = listHead; listWalk != NULL;
         listWalk = listWalk->next)
   {
      #pragma omp single nowait
      {
         process2(listWalk);
      }
   }
```

```
}
```


Related Articles from MSDN Magazine:

- **WinUnit:** Simplified Unit Testing for Native C++ Applications by Maria Blees
- **C++:** An Inside Look At The Next Generation Of Visual C++ by Tarek Madkour
- **Windows with C++:** Windows Template Library 8.0 by Kenny Kerr
- **C++ Q&A:** Call Unmanaged DLLs from C#, Killing Processes Cleanly by Paul DiLascia
- **C++ -> C#:** What You Need to Know to Move from C++ to C# by Jesse Liberty
- **Sharp New Language:** C# Offers the Power of C++ and Simplicity of Visual Basic by Joshua Trupin
- **XmlLite:** A Small And Fast XML Parser For Native C++ by Kenny Kerr
- **Windows with C++:** Windows Vista Control Enhancements by Kenny Kerr