Learning the OpenMP framework with GCC

Arpan Sen

September 07, 2012

The Open Multiprocessing (OpenMP) framework is an extremely powerful specification that helps you harness the benefits of multiple cores from your C, C++, and Fortran applications. This article explains how to use the OpenMP features in your C++ code and provides examples that can help you get started with OpenMP.

The OpenMP framework is a powerful way of doing concurrent programming in C, C++, and Fortran. The GNU Compiler Collection (GCC) version 4.2 supports the OpenMP 2.5 standard, while GCC 4.4 supports the latest OpenMP 3 standard. Other compilers, including Microsoft® Visual Studio, support OpenMP as well. In this article, you can learn to use OpenMP compiler pragmas, find support for some of the application programming interfaces (APIs) that OpenMP provides, and test OpenMP with some parallel algorithms. This article uses GCC 4.2 as the preferred compiler.

Getting started

A great feature of OpenMP is that you do not need anything other than your standard GCC installation. Programs with OpenMP must be compiled with the `-fopenmp` option.

Your first OpenMP program

Let us start with a simple **Hello, World!** printing application that includes an additional pragma. Listing 1 shows the code.

Listing 1. Hello World with OpenMP

```
#include <iostream>
int main()
{
  #pragma omp parallel
  {
    std::cout << "Hello World!\n";
  }
}
```

When compiling and running the code in Listing 1 with g++, a single **Hello, World!** should be displayed in the console. Now, recompile the code with the `-fopenmp` option. Listing 2 shows the output.

Listing 2. Compiling and running the code with the `-fopenmp` command

```
tintin$ g++ test1.cpp -fopenmp
tintin$ ./a.out
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

So, what just happened? The magic of the `#pragma omp parallel` works only when you specify the `-fopenmp` compiler option. Internally, during compilation, GCC generates the code to create as many threads as it optimally can at run time based on the hardware and operating system configuration, with the start routine of each created thread being the code in the block that follows the pragma. This behavior is *implicit parallelization,* and OpenMP in its core consists of a set of powerful pragmas that relieve you of having to perform a lot of boilerplate coding. (For the sake of comparison, check out what a Portable Operating System Interface (POSIX) threads [pthreads] implementation of what you just did would be like.) Because I am using a computer running an Intel® Core i7 processor with four physical cores and two logical cores per physical core, the output from Listing 2 seems quite reasonable (8 threads = 8 logical cores).

Now, let us get in to more details about parallel pragmas.

Fun with OpenMP parallel

It is easy enough to control the number of threads using the `num_threads` argument to the pragma. Here is the code from Listing 1 again with the number of available threads specified at 5 (as shown in Listing 3).

Listing 3. Controlling the number of threads with `num_threads`

```
#include <iostream>
int main()
{
  #pragma omp parallel num_threads(5)
  {
    std::cout << "Hello World!\n";
  }
}
```

Instead of the `num_threads` approach, here is an alternative to change the number of threads running the code. That also brings us to the first OpenMP API you will be using: `omp_set_num_threads`. You define this function in the omp.h header file. No additional libraries need to be linked to get the code in Listing 4 to work—just `-fopenmp`.

Listing 4. Using `omp_set_num_threads` to fine tune thread creation

```
#include <omp.h>
#include <iostream>
int main()
{
  omp_set_num_threads(5);
  #pragma omp parallel
  {
    std::cout << "Hello World!\n";
  }
}
```

Finally, OpenMP also uses external environment variables to control its behavior. You can tweak the code in Listing 2 to simply print **Hello World!** six times by setting the `OMP_NUM_THREADS` variable to `6`. Listing 5 shows the execution.

Listing 5. Using environment variables to tweak OpenMP behavior

```
tintin$ export OMP_NUM_THREADS=6
tintin$ ./a.out
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

You have discovered all the three facets of OpenMP: compiler pragmas, runtime APIs, and environment variables. What happens if you use both the environment variable and the runtime API? The runtime API gets higher precedence.

A practical example

OpenMP uses implicit parallelization techniques, and you can use pragmas, explicit functions, and environment variables to instruct the compiler. Let us look at an example in which OpenMP can be of real help. Consider the code in Listing 6.

Listing 6. Sequential processing in a `for` loop

```
int main( )
{
int a[1000000], b[1000000];
// ... some initialization code for populating arrays a and b;
int c[1000000];
for (int i = 0; i < 1000000; ++i)
  c[i] = a[i] * b[i] + a[i-1] * b[i+1];
// ... now do some processing with array c
 }
```

Clearly, you can potentially split the `for` loop and run into multiple cores, calculating any `c[k]` is dependency free from other elements of the `c` array. Listing 7 shows how OpenMP helps you do so.

Listing 7. Parallel processing in a `for` loop with the `parallel for` pragma

```
int main( )
{
int a[1000000], b[1000000];
// ... some initialization code for populating arrays a and b;
int c[1000000];
#pragma omp parallel for
for (int i = 0; i < 1000000; ++i)
  c[i] = a[i] * b[i] + a[i-1] * b[i+1];
// ... now do some processing with array c
 }
```

The `parallel for` pragma helps to split the `for` loop workload across multiple threads, with each thread potentially running on a different core, thus reducing the total computation time significantly. Listing 8 proves the point.

Listing 8. Understanding omp_get_wtime

```
#include <omp.h>
#include <math.h>
#include <time.h>
#include <iostream>

int main(int argc, char *argv[]) {
    int i, nthreads;
    clock_t clock_timer;
    double wall_timer;
    double c[1000000];
    for (nthreads = 1; nthreads <=8; ++nthreads) {
        clock_timer = clock();
        wall_timer = omp_get_wtime();
        #pragma omp parallel for private(i) num_threads(nthreads)
        for (i = 0; i < 1000000; i++)
          c[i] = sqrt(i * 4 + i * 2 + i);
        std::cout << "threads: " << nthreads <<  " time on clock(): " <<
            (double) (clock() - clock_timer) / CLOCKS_PER_SEC
           << " time on wall: " <<  omp_get_wtime() - wall_timer << "\n";
    }
}
```

In Listing 8, you benchmark the time it takes to run the inner `for` loop by continuously increasing the number of threads. The `omp_get_wtime` API returns the elapsed wall time in seconds from some arbitrary but consistent point. So, `omp_get_wtime() - wall_timer` returns the observed wall time to run the `for` loop. The `clock()` system call is used to estimate the processor usage time for the whole program—that is, individual thread-specific processor usage time is summed up before reporting the final figures. On my Intel Core i7 computer, Listing 9 shows what gets reported.

Listing 9. Statistics for running the inner `for` loop

```
threads: 1 time on clock(): 0.015229 time on wall: 0.0152249
threads: 2 time on clock(): 0.014221 time on wall: 0.00618792
threads: 3 time on clock(): 0.014541 time on wall: 0.00444412
threads: 4 time on clock(): 0.014666 time on wall: 0.00440478
threads: 5 time on clock(): 0.01594 time on wall: 0.00359988
threads: 6 time on clock(): 0.015069 time on wall: 0.00303698
threads: 7 time on clock(): 0.016365 time on wall: 0.00258303
threads: 8 time on clock(): 0.01678 time on wall: 0.00237703
```

Although the processor time is almost the same across executions (they should be, except for some additional time to create the threads and the context switch), it is the wall time that is interesting and is progressively reduced as the number of threads increases, implying that data is being crunched by cores in parallel. A final note on the pragma syntax: `#pragma parallel for private(i)` means that the loop variable `i` is to be treated as a thread local storage, with each thread having a copy of the variable. The thread local variable is not initialized.

Critical sections with OpenMP

You were not exactly thinking of having OpenMP figure out how to deal with critical sections automatically, were you? Sure, you do not have to explicitly create a mutual exclusion (mutex), but you still need to specify the critical section. Here is the syntax:

```
#pragma omp critical (optional section name)
{
// no 2 threads can execute this code block concurrently
}
```

The code that follows `pragma omp critical` can only be ran by a single thread at a given time. Also, `optional section name` is a global identifier, and no two threads can run critical sections with the same global identifier name at the same time. Consider the code in Listing 10.

Listing 10. Multiple critical sections with the same name

```
#pragma omp critical (section1)
{
myhashtable.insert("key1", "value1");
}
// ... other code follows
#pragma omp critical (section1)
{
myhashtable.insert("key2", "value2");
}
```

Based on this code, you can safely assume that the two hash table insertions will never happen concurrently, because the critical section names are the same. This is slightly different from the way you are accustomed to dealing with critical sections while using pthreads, which are by and large characterized by the use (or abuse) of locks.

Locking and mutexes with OpenMP

Interestingly enough, OpenMP comes with its own versions of mutexes (so it is not all pragmas after all): Welcome to `omp_lock_t`, defined as part of the omp.h header file. The usual pthread-style mutex operations hold true—even the API names are similar. There are five APIs that you need to know:

- **`omp_init_lock`**: This API must be the first API to access `omp_lock_t`, and it is used for initialization. Note that right after initialization, the lock is deemed to be in an unset state.
- **`omp_destroy_lock`**: This API destroys the lock. The lock must be in an unset state when this API is called, which means that you cannot call `omp_set_lock`, and then make a call to destroy the lock.

- **`omp_set_lock`**: This API sets `omp_lock_t`—that is, the mutex is acquired. If a thread cannot set the lock, then it continues to wait until it is able to lock.
- **`omp_test_lock`**: This API tries to lock if the lock is available, and returns `1` if successful and `0` otherwise. This is a *non-blocking API*—that is, this function does not make the thread wait to set the lock.
- **`omp_unset_lock`**: This API releases the lock.

Listing 11 shows a trivial implementation of a legacy single-threaded queue extended to handle multithreading using OpenMP locks. Note that this might not be the right thing to do in every situation, and the example as such is primarily meant as a quick illustration.

Listing 11. Using OpenMP to extend a single-threaded queue

```
#include <openmp.h>
#include "myqueue.h"

class omp_q : public myqueue<int> {
public:
   typedef myqueue<int> base;
   omp_q( ) {
      omp_init_lock(&lock);
   }
   ~omp_q() {
       omp_destroy_lock(&lock);
   }
   bool push(const int& value) {
      omp_set_lock(&lock);
      bool result = this->base::push(value);
      omp_unset_lock(&lock);
      return result;
   }
   bool trypush(const int& value)
   {
       bool result = omp_test_lock(&lock);
       if (result) {
          result = result && this->base::push(value);
          omp_unset_lock(&lock);
      }
      return result;
   }
   // likewise for pop
private:
   omp_lock_t lock;
};
```

Nested locks

Other types of locks that OpenMP provides are the `omp_nest_lock_t` lock variants. These are similar to `omp_lock_t`, with the additional benefit that these locks can be locked multiple times by the thread that is already holding the lock. Each time the nested lock is reacquired by the holding thread using `omp_set_nest_lock`, an internal counter is incremented. The lock is freed from the holding thread when one or more calls to `omp_unset_nest_lock` finally reset the internal lock counter to `0`. Here are the APIs used for `omp_nest_lock_t`:

- **`omp_init_nest_lock(omp_nest_lock_t* )`**: This API initializes the internal nesting count to `0`.
- **`omp_destroy_nest_lock(omp_nest_lock_t* )`**: This API destroys the lock. Calling this API on a lock with a non-zero internal nesting count results in an undefined behavior.

- **`omp_set_nest_lock(omp_nest_lock_t* )`**: This API is similar to `omp_set_lock`, except that the thread can call this function multiple times while the thread is holding the lock.
- **`omp_test_nest_lock(omp_nest_lock_t* )`**: This API is a non-blocking version of `omp_set_nest_lock`.
- **`omp_unset_nest_lock(omp_nest_lock_t* )`**: This API releases the lock when the internal counter is `0`. Otherwise, the counter is decremented with each call to this method.

Fine-grained control over task execution

You have already seen that all the threads run the code block that follows `pragma omp parallel` in parallel. It is possible to further categorize the code inside this block for execution by select threads. Consider the code in Listing 12.

Listing 12. Learning to use the parallel sections pragma

```
int main( )
{
  #pragma omp parallel
  {
    cout << "All threads run this\n";
    #pragma omp sections
    {
      #pragma omp section
      {
        cout << "This executes in parallel\n";
      }
      #pragma omp section
      {
        cout << "Sequential statement 1\n";
        cout << "This always executes after statement 1\n";
      }
      #pragma omp section
      {
        cout << "This also executes in parallel\n";
      }
    }
  }
}
```

The code that precedes `pragma omp sections`, but just after `pragma omp parallel`, is ran by all the threads in parallel. The block that succeeds `pragma omp sections` is further classified into individual subsections using `pragma omp section`. Each `pragma omp section` block is available for execution by an individual thread. However, the individual statements inside the section block are always run sequentially. Listing 13 shows the output of the code from Listing 12.

Listing 13. Output from running the code in Listing 12

```
tintin$ ./a.out
All threads run this
All threads run this
All threads run this
All threads run this
All threads run this
All threads run this
All threads run this
All threads run this
This executes in parallel
Sequential statement 1
This also executes in parallel
This always executes after statement 1
```

In Listing 13, you again have eight threads being created initially. Of these eight threads, there is sufficient work for only three threads in the `pragma omp sections` block. Within the second section, you specify the order in which the print statements are ran. That is the whole point behind having the `sections` pragma. If there is a need, you will be able to specify ordering of code blocks.

Understanding the `firstprivate` and `lastprivate` directives in conjunction with parallel loops

Earlier, you saw the use of `private` to declare thread local storage. So, how should you initialize the thread local variables? Perhaps synchronize them with the value of the variable in the main thread before ensuing operations? This is where the `firstprivate` directive comes in handy.

The firstprivate directive

Using `firstprivate(variable)`, you can initialize the variable in a thread to whatever value it had in the main. Consider the code in Listing 14.

Listing 14. Using the thread local variable that is not synchronized with the main thread

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

int main()
{
  int idx = 100;
  #pragma omp parallel private(idx)
  {
    printf("In thread %d idx = %d\n", omp_get_thread_num(), idx);
  }
}
```

Here is the output I get. Your results might vary.

```
In thread 1 idx = 1
In thread 5 idx = 1
In thread 6 idx = 1
In thread 0 idx = 0
In thread 4 idx = 1
In thread 7 idx = 1
In thread 2 idx = 1
In thread 3 idx = 1
```

Listing 15 shows the code with the `firstprivate` directive. The output, as expected, prints`idx` initialized to `100` in all the threads.

Listing 15. Using the `firstprivate` directive to initialize thread local variables

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

int main()
{
  int idx = 100;
  #pragma omp parallel firstprivate(idx)
  {
    printf("In thread %d idx = %d\n", omp_get_thread_num(), idx);
  }
}
```

Also, note that you have used the `omp_get_thread_num( )` method to access a thread's ID. This is different from the thread ID that the Linux®`top` command shows, and this scheme is just a way for OpenMP to keep track of the thread counts. Another note on the `firstprivate` directive if you are planning to use it with your `C++` code: The variable that the `firstprivate` directive uses is a copy constructor to initialize itself from the master thread's variable, so having a copy constructor that is private to your class will invariably result in bad things. Let us move on to the `lastprivate` directive now, which in many ways is the other side of the coin.

The `lastprivate` directive

Instead of initializing a thread local variable with the main thread's data, you now intend to synchronize the main thread's data with whatever data the last ran loop count generated. The code in Listing 16 runs a parallel `for` loop.

Listing 16. Using a parallel `for` loop with no data synchronization with the main thread

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

int main()
{
  int idx = 100;
  int main_var = 2120;

  #pragma omp parallel for private(idx)
  for (idx = 0; idx < 12; ++idx)
  {
    main_var = idx * idx;
    printf("In thread %d idx = %d main_var = %d\n",
      omp_get_thread_num(), idx, main_var);
  }
  printf("Back in main thread with main_var = %d\n", main_var);
}
```

On my development computer with eight cores, OpenMP ends up creating six threads for the `parallel for` block. Each thread in turn accounts for two iterations of the loop. The final value of `main_var` depends on the last thread that ran, and therefore, the value of `idx` in that thread. In other words, the value of `main_var` does not depend on the last value of `idx` but the value of `idx` in whichever thread that ran last. The code in Listing 17 illustrates the point.

Listing 17. The value of main_var depends on the last thread run

```
In thread 4 idx = 8 main_var = 64
In thread 2 idx = 4 main_var = 16
In thread 5 idx = 10 main_var = 100
In thread 3 idx = 6 main_var = 36
In thread 0 idx = 0 main_var = 0
In thread 1 idx = 2 main_var = 4
In thread 4 idx = 9 main_var = 81
In thread 2 idx = 5 main_var = 25
In thread 5 idx = 11 main_var = 121
In thread 3 idx = 7 main_var = 49
In thread 0 idx = 1 main_var = 1
In thread 1 idx = 3 main_var = 9
Back in main thread with main_var = 9
```

Run the code in Listing 17 a couple of times to be convinced that the value of `main_var` in the main thread is always dependent on the value of `idx` in the last-run thread. Now, what if you want to synchronize the value of the main thread with the final value of `idx` in the loop? This is where the `lastprivate` directive comes in, which is illustrated in Listing 18. Similar to the code in Listing 17, run the code in Listing 18 a few times to convince yourself that the final value of `main_var` in the main thread is `121` (`idx` being the final loop counter value).

Listing 18. Using the `lastprivate` directive for synchronization

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

int main()
{
  int idx = 100;
  int main_var = 2120;

  #pragma omp parallel for private(idx) lastprivate(main_var)
  for (idx = 0; idx < 12; ++idx)
  {
    main_var = idx * idx;
    printf("In thread %d idx = %d main_var = %d\n",
      omp_get_thread_num(), idx, main_var);
  }
  printf("Back in main thread with main_var = %d\n", main_var);
}
```

Listing 19 shows the output of Listing 18.

Listing 19. Output from the code in Listing 18 (note that the `main_var always` value equals 121 in the main thread)

```
In thread 3 idx = 6 main_var = 36
In thread 2 idx = 4 main_var = 16
In thread 1 idx = 2 main_var = 4
In thread 4 idx = 8 main_var = 64
In thread 5 idx = 10 main_var = 100
In thread 3 idx = 7 main_var = 49
In thread 0 idx = 0 main_var = 0
In thread 2 idx = 5 main_var = 25
In thread 1 idx = 3 main_var = 9
In thread 4 idx = 9 main_var = 81
In thread 5 idx = 11 main_var = 121
In thread 0 idx = 1 main_var = 1
Back in main thread with main_var = 121
```

A final note: Supporting the `lastprivate` operator for a `C++` object requires that the corresponding class have the `operator=` method publicly available.

Merge sort with OpenMP

Let us look at a real-world example in which knowing OpenMP will help you save on the run time. This is not a heavily optimized version of `merge sort`, but it is enough to show the benefits of using OpenMP in your code. Listing 20 shows the example's code.

Listing 20. Merge sort using OpenMP

```
#include <omp.h>
#include <vector>
#include <iostream>
using namespace std;

vector<long> merge(const vector<long>& left, const vector<long>& right)
{
    vector<long> result;
    unsigned left_it = 0, right_it = 0;

    while(left_it < left.size() && right_it < right.size())
    {
        if(left[left_it] < right[right_it])
        {
            result.push_back(left[left_it]);
            left_it++;
        }
        else
        {
            result.push_back(right[right_it]);
            right_it++;
        }
    }

    // Push the remaining data from both vectors onto the resultant
    while(left_it < left.size())
    {
        result.push_back(left[left_it]);
        left_it++;
    }

    while(right_it < right.size())
    {
        result.push_back(right[right_it]);
        right_it++;
    }

    return result;
}

vector<long> mergesort(vector<long>& vec, int threads)
{
    // Termination condition: List is completely sorted if it
    // only contains a single element.
    if(vec.size() == 1)
    {
        return vec;
    }

    // Determine the location of the middle element in the vector
    std::vector<long>::iterator middle = vec.begin() + (vec.size() / 2);

    vector<long> left(vec.begin(), middle);
    vector<long> right(middle, vec.end());

    // Perform a merge sort on the two smaller vectors

    if (threads > 1)
    {
      #pragma omp parallel sections
      {
        #pragma omp section
        {
          left = mergesort(left, threads/2);
        }
        #pragma omp section
        {
          right = mergesort(right, threads - threads/2);
        }
      }
    }
    else
    {
      left = mergesort(left, 1);
      right = mergesort(right, 1);
    }

    return merge(left, right);
}

int main()
{
  vector<long> v(1000000);
  for (long i=0; i<1000000; ++i)
    v[i] = (i * i) % 1000000;
  v = mergesort(v, 1);
  for (long i=0; i<1000000; ++i)
    cout << v[i] << "\n";
}
```

Using eight threads to run this `merge sort` gave me a run time execution time of 2.1 seconds, while with one thread it was 3.7 seconds. The only thing worth remembering here is that you need to be careful with the number of threads. I started off with eight threads: The mileage can vary according to your system configuration. However, without the explicit thread count, you would end up creating hundreds if not thousands of threads, and the probability that the system performance would degrade is pretty high. Also, the `sections` pragma discussed earlier has been put to good use with the `merge sort` code.

Conclusion

That is it for this article. We covered a fair bit of ground here: You were introduced to OpenMP parallel pragmas; learned different ways to create threads; convinced yourself of the better time performance, synchronization, and fine-grained control that OpenMP provides; and wrapped it all up with a practical application of OpenMP with `merge sort`. There is a lot more to study, though, and the best place to check it out is the OpenMP project site. Be sure to check the Related topics section for additional details.

Related topics

- Be sure to check out the OpenMP project site.
- For more information on improving merge-sort performance, read Shared Memory, Message Passing, and Hybrid Merge Sorts for Standalone and Clustered SMPs by Atanas Radenski.