OpenMP 01: An Introduction to OpenMP

Draft 2 -- Patrick Garrity -- St. Olaf College

OpenMP is a well-developed threading library for C, C++, and FORTRAN that simplifies the task of parallel programming. Using preprocessor directives, OpenMP allows developers to add concurrency to software without worrying about explicit threading, and low-level problems such as scheduling and thread management. One benefit of this library is that it has a very accessible learning curve, but at the same time scales to be an extremely powerful tool in the right hands. Because of this, and the standard inclusion of OpenMP in various compilers such as GCC and ICC, at least an entry-level knowledge of OpenMP is essential for any modern C++ programmer.

1. About OpenMP

As stated in the introduction, OpenMP is a library that simplifies parallel programming. Almost every aspect of thread management is hidden from the developer, allowing problems to be approached from a higher level. This allows tasks such as parallelizing a loop to remain simple, rather than cluttered by thread management code. OpenMP is capable of threading based on sections, loops, and tasks, making it widely applicable. The syntax of OpenMP for C++ is simple and based on pragmas, which are special preprocessor directives. This style of programming allows OpenMP to be inserted into programs without any need for special objects or variables. Finally, a small function library is offered that allows developers to get and set basic information about the programming environment. Using OpenMP is often best-explained through example, so this lesson will present two example programs that add parallelism to a program in two different ways.

2. Example Program

The following example program introduces OpenMP by running two loops at the same time. A thorough explanation of the source code will follow the program listing.

```
// File: lesson01.cpp
#include <omp.h>
#include <iostream>

using std::cout;
using std::endl;

int main(int argc, char ** argv)
{
    int i = 0;
    int p = 0;
    int n = 10;
    float * data = new float[n];

    cout << "Threads: ";
```

```
    #pragma omp parallel sections default(shared) private(i, p)
    {
        #pragma omp section
        {
            // Loop #1 (i=0,1,2,3,4)
                  for (i = 0; i < (n/2); i++)
                  {
                            data[i] = i * 10;
                  }

                  p = omp_get_thread_num();
                  cout << p;
        }

        #pragma omp section
        {
            // Loop #2 (i=5,6,7,8,9)
                  for (i = (n/2); i < n; i++)
                  {
                            data[i] = i * 10;
                  }

                  p = omp_get_thread_num();
                  cout << p;
        }
    }
    cout << endl;

    if (data != 0)
        delete [] data;
    data = 0;
    return 0;
}
```

Code Breakdown

The program fills an array of size 10 with the values 0, 10, 20, 30, 40, 50, 60, 70, 80, 90. It does so by executing two loops. The first loop fills indices 0, 1, 2, 3, 4, while the second loop fills the indices 5, 6, 7, 8, 9. This program is trivial, but OpenMP has been used to make the two loops run in parallel. There are three important lines of code (which have been highlighted in the source listing).

`#pragma omp parallel sections default(shared) private(i, p)`

This line of code defines a block of code where various *sections* can be executed in parallel. The *`parallel`* keyword causes OpenMP to, "form a team of threads and start parallel execution." [2] This is all done behind the scenes, allowing the developer to focus on other details. The *`sections`* keyword lets OpenMP know that the directive will be followed by a block of code with multiple sections inside. These sections will all be executed in parallel if possible.

The final two clauses of the pragma are related to managing variables. Variables can be either

shared or ***private***. If a variable is shared, then each thread accesses and changes the same variable. If a variable is private, then each thread makes its own copy of that variable. In most cases, it is important to specify which variables are shared and which are private. The clause `default(shared)` informs OpenMP that within this set of sections, any variables that are not explicitly declared private will be treated as shared variables. We specify the variables `i` and `p` to be private with the final clause of the directive.

`#pragma omp section`
This line defines a block of code that is treated as a *section* by OpenMP. All sections of code within a set of parallel sections will run in parallel. In this example, there are two sections, and therefore two threads will be used to run these sections concurrently if possible. Since each section contains one of the loops, these loops will likely run (and modify the `data[]` variable) at the same time.

`p = omp_get_thread_num();`
This line is an example of OpenMP code that does not use pragmas. There are a small number of functions available that allow developers to get and set certain information. In this case, the function returns the number of the current thread. Thread numbers start at 0 for the main thread, and increase as more threads are present.

How to Compile and Link the Example

```
$ g++ -fopenmp -c lesson01.cpp -o lesson01.o
$ g++ -fopenmp -o lesson01 lesson01.o
```

Note: The `-fopenmp` option tells GCC that we want to compile and link with support for OpenMP.

Example Output

```
$ ./lesson01
Threads: 01

$ OMP_NUM_THREADS=4 ./lesson01
Threads: 01

$ OMP_NUM_THREADS=1 ./lesson01
Threads: 00
```

The only output of this program is the number of each thread that is used. Run on a dual-core machine, it created two threads (0 and 1) by default. OpenMP allows developers to specify the number of threads that should be created in various ways. One of these is through the environment variable `OMP_NUM_THREADS`. When this is set to 4, we still only use two threads since there are only two sections. The excess threads go unused. It is important to note that even though a dual-core machine is being used, more than 2 threads can be created. The only limitation is that any number of threads beyond the number of cores will *not* run in parallel. Finally, the `OMP_NUM_THREADS` variable is set to 1 in the final example run which prevents OpenMP from creating any additional threads. The program does in fact react to this, forcing

both sections to run on the default thread (0).

3. Another Example Program: Parallel For

The previous example does introduce parallelism by splitting a loop into two chunks, but performing manual splitting for extremely large loops (or non-hard-coded loops) could become extremely tedious or impractical. For this situation, OpenMP offers the `parallel for` directive which automatically parallelizes the same `for` loop.

```
// File: lesson01-2.cpp
#include <omp.h>
#include <iostream>

using std::cout;
using std::endl;

int main(int argc, char ** argv)
{
    int i = 0;
    int p = 0;
    int n = 10;
    float * data = new float[n];

    cout << "Threads: ";
    #pragma omp parallel for default(shared) private(i, p)
    for (i = 0; i < n; i++)
    {
        data[i] = i * 10;
        p = omp_get_thread_num();
        cout << p;
    }
    cout << endl;

    if (data != 0)
        delete [] data;
    data = 0;
    return 0;
}
```

Code Breakdown

The behavior of this program is almost identical to that of the first example, though it fills the `data[]` array in a different way. In this example, only one `for` loop is created, and no sections are used. Again, the OpenMP-related lines of code have been highlighted.

`#pragma omp parallel for default(shared) private(i, p)`

This pragma tells OpenMP to expect a `for` loop to follow the line - this `for` loop is parallelized according to the rest of the line. The new code in this line is the presence of the **`for`** directive as opposed to `sections` as used in the first example. The `omp parallel for` directive has

various clauses that may be used to tune its performance that can be found in [2].

How to Compile and Link the Example

```
$ g++ -fopenmp -c lesson01-2.cpp -o lesson01-2.o
$ g++ -fopenmp -o lesson01-2 lesson01-2.o
```

Example Output

```
$ ./lesson01-2
Threads: 0000011111

$ ./lesson01-2
Threads: 0111110000

$ OMP_NUM_THREADS=4 ./lesson01-2
Threads: 0001112223

$ OMP_NUM_THREADS=8 ./lesson01-2
Threads: 0022334411
```

Again, the output of this program is the number of each thread that is used. In this case, the thread prints its number every time it executes the body of the loop, which means that we should have 10 numbers of output. The first two example runs demonstrate that the threads are not bound to any particular order of execution - they run in parallel. Notice that the default number of threads used was again two. In the third example the number of threads is set to 4, which is reflected in the execution of the program. We can see that threads 0, 1, and 2 each handled three iterations of the loop, while thread 3 only received one iteration of work. Finally when the thread count is set to 8 notice that some of the threads go unused, as they were deemed unnecessary by OpenMP for such a small problem.

4. Conclusion

Like most introductions, this lesson has only scratched the surface of what is possible to do using OpenMP. However, due to the simplicity and consistent syntax of OpenMP, learning the additional features and clauses is extremely straightforward. The greatest challenge in parallel programming is determining a good way to add parallelism to a piece of software. OpenMP is designed so that once this problem has been solved, the parallelism can be added using a minimal amount of code.

5. References

[1] “Pragmas - The C Preprocessor”. http://gcc.gnu.org/onlinedocs/cpp/Pragmas.html
[2] “The OpenMP API Specification for Parallel Programming”. http://www.openmp.org/mp-documents/OpenMP3.0-SummarySpec.pdf