Week 6: OpenMP

Ed Davis

08/03/2021

Introduction

OpenMP is an interface which allows for running programs over multiple cores.

Creating an OpenMP programs

Consider the simple OpenMP C++ program, which is a program to print out `Hello OpemMP!`.

```cpp
#include <iostream>

int main(int argc, const char **argv)
{
    #pragma omp parallel // This tells the program to run this code block for each thread
    {
        std::cout << "Hello OpenMP!\n";
    }

    return 0;
}
```

This can be compiled by g++ in the terminal through the use of the following command,

```
g++ -fopenmp hello_openmp.cpp -o hello_openmp
```

which produces the `hello_openmp` executable. This can then be run by simply entering `hello_openmp` into the terminal. The program will then output the message, `Hello OpenMP!` the same number of times as the number of threads on your computer.

The number of threads can be set by editing the `OMP_NUM_THREADS` variable.

```
set OMP_NUM_THREADS=8  # selects to use 8 threads (windows)
export OMP_NUM_THREADS=8  # selects to use 8 threads (linux / terminals with export command)
```

Directives (Pragmas)

In a standard program, each line is executed one at a a time, starting from some `main` function. This single thread execution is known as the `main` thread, and all programs have a single `main` thread. Our `hello_openmp` program also has a single `main` thread of execution, however this `main` thread is split into a team of threads by OpenMP in parallel. This is why `Hello OpenMP!` is printed for each thread. If you use the compile command above without `-fopenmp`, the program will run as a single thread program.

OpenMP directives are given below.

- `parallel` : Used to create a parallel block of code which will be executed by a team of threads
- `sections` : Used to specify different sections of the code that can be run in parallel by different threads.
- `for` (C/C++), `do` (Fortran) : Used to specify loops where different iterations of the loop are performed by different threads.
- `critical` : Used to specify a block of code that can only be run by one thread at a time.
- `reduction` : Used to combine (reduce) the results of several local calculations into a single result

These are added to code with the syntax, `#pragma omp name_of_directive`.

Sections

Code can be sectioned between threads through the use of the sections pragma. Consider a section of `C++` below where `times_table()`, `countdown()` and `long_loop` are functions.

```
int main(int argc, const char **argv)
{
    std::cout << "This is the main thread.\n";

    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                times_table(12);
            }
            #pragma omp section
            {
                countdown();
            }
            #pragma omp section
            {
                long_loop();
            }
        }
    }
```

In this script, we have used the `parallel` and `sections` pragmas. The `sections` pragma specifies sections of the code to be run in parallel by different threads. In this case, only three threads will be doing anything as we have designed then to calculating `times_table(12)`, `countdown()` and `long_loop()`. If this program was to be run with `OMP_NUM_THREADS=8`, then three threads will be calculating in parallel and the other five threads will be left idle.

Note that for some terminals (e.g. Windows), the output from these three threads will be interupting each other, an example of which is shown below.

```
Thread 2 says 3...
Thread 1 says 8 times 12 equals 96
Thread 1Thread 2 says 2...
 says 9 times 12 equals 108
Thread Thread 2 says 1...
1 says 10 times 12 equals 120
Thread 2 says "Lift off!"
```

But other terminals such as Mac and Linux make sure that the threads do not interrupt.

Loops

Running iterations of a loop in parallel is difficult to scale as you can only run as many threads as there are functions to be run and if there are more threads than functions, surplus threads will be idle. In addition, if functions take different amounts of time to run, then it could lead to iterations completing in different orders.

The following is a simple program that calculates a loop in parallel.

```
#include <iostream>

#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
#endif

int main(int argc, const char **argv)
{
    #pragma omp parallel
    {
        int nloops = 0;

        #pragma omp for
        for (int i=0; i<1000; ++i)
        {
            ++nloops;
        }

        int thread_id = omp_get_thread_num();

        std::cout << "Thread " << thread_id << " performed "
                  << nloops << " iterations of the loop.\n";
    }

    return 0;
}
```

The output for this program will be

```
Thread Thread 2 performed 125 iterations of the loop.
Thread 7 performed 125 iterations of the loop.
Thread 5 performed 125 iterations of the loop.
Thread 6 performed 125 iterations of the loop.
Thread 3 performed 125 iterations of the loop.
0 performed 125 iterations of the loop.
Thread 4 performed 125 iterations of the loop.
Thread 1 performed 125 iterations of the loop.
```

for eight threads. We can see that using the `for` pragma, each of the threads calculated the same amount of iterations in the loop. It is important to note that this will only work if each iteration of the loop is independent, meaning that the order in which the iterations are calculated does not affect the result of the program and each iteration does not affect another. For this program the number of iterations is 1000; if 1500 threads were used, 500 would be left idle.

Critical Code

Up to now, threads running in parallel have not been working together, e.g. each thread is aware of how many iterations of a loop it has done but no idea about the number of total iterations done by them all.

A `critical` section of the code (with the `critical` pragma) is one which is performed by all threads in the team, but is only performed by a single thread at a time. This allows for each thread to update a global variable with the local result calculated by the thread, without worrying about other threads updating the variable at the same time. These `critical` sections are important for guaranteeing reproducibility and reliability.

Reduction

Reduction is the process of combining or reducing the results of many sub-calculations into a single, reduced, result.