CS402: Seminar 2
An Introduction to OpenMP

OpenMP (Multiprocessing) is an API and runtime which enables the programming of multiple processing cores with *shared memory*. The API is a collection of functions and pragmas, the former allows the querying of information, such as the number of active threads; and the latter allows the definition of parallel regions. It is in these parallel regions, that threads are spawned by the OpenMP runtime and the code within executed on multiple cores. These threads are then able to communicate with each other using shared memory. This lab will give a basic introduction in how to use the OpenMP API to exploit parallelism in applications.

1 Hello World in OpenMP

As with the other labs, the first example we will look at is a variant of *"Hello, world!"* which has been modified such that each thread prints out its ID and *"Hello, world!"*. You can find the code in Listing 1 and the file named `helloWorldOMP.c`.

```
#include <omp . h>
#include <stdio . h>
#include <s t d l i b . h>

int main( int argc , char *argv [ ] ) {
    /* Define parallel region */
    #pragma omp parallel
    {
        int thread_id ;
        /* Use the OpenMP API to query which thread we are in */
        thread_id = omp_get_thread_num () ;
        p r i n t f (" Hello World from thread: 91cd\n" , thread_id ) ;

        /* Ensure all threads have printed before continuing */
        #pragma omp barrier
        /* Action only from the thread with ID 0 */
        i f ( thread_id == 0){
            /* Print the total the total number of threads */
            p r i n t f ("Total number of threads: 91cd\n" ,
                omp_get_num_threads () ) ;
        }
    }

    return 0;
}
```

Listing 1: OpenMP "Hello World" example

This example can be compiled with the following command: `gcc -fopenmp -o helloWorldOMP.b helloWorldOMP.c`

The only difference from the normal compilation process is the addition of the `-fopenmp` flag. This is a compiler flag rather than just a collection of libraries, because OpenMP requires compiler support for the additional language features it provides. Run the compiled binary just like you would any other application, and you will be greeted with output similar to that below:

```
Hello World from thread: 0
Hello World from thread: 1
Total number of threads: 2
```

Your output may vary from the above for two reasons: 1) there is no guarantee to the order in which each thread will print *"Hello, world!"* and; 2) by default the OpenMP run time will choose the number of threads to spawn in a parallel region based on your hardware. In this case the runtime has selected 2 threads as this is what it thinks is best for the available hardware. It is possible to override this decision by setting the `OMP_NUM_THREADS` environmental variable. For this lab we will leave the OpenMP runtime to decide how many threads should be used. Congratulations, you have compiled and run a basic OpenMP program, now we will look at each part of the program in more detail.

- `#include<omp.h>` – Necessary for accessing the functions provided by the OpenMP runtime, such as `omp_get_thread_num()`.

- `#pragma omp parallel` – Define the following region (the area scoped by { and }) as to be executed in parallel. If this pragma is used without braces then it will default to the line of code below it, so in a similar manner to how the scope of `if` statements operate.

- `omp_get_thread_num()` – A function which returns the `ID` of the thread the program is executing in.
- `#pragma omp barrier` – Instruct the OpenMP runtime to wait for all threads to reach this point before continuing. This is how we can force all threads to print *"Hello, world!"* before thread 0 reports the total number of threads.
- `omp_get_num_threads()` – A function which reports the number of threads which the OpenMP runtime has spawned in a parallel region.

As always, if you have any questions at this point, make sure you ask one of the tutors for help.

2 Parallel Loops

Defining parallel regions with `#pragma omp parallel is` just one method by which OpenMP allows parallelism to be exploited; another mechanism which OpenMP provides allows iterations of a loop to be *scheduled* across available threads. This is done by proceeding a for loop with the following pragma:

```
#pragma omp parallel for
```

There are several options to this pragma, including those which allow the selection of scheduling algorithm and the scoping of variables. These two options will be detailed here, but it is recommended to review them all by reading the appropriate sections of the OpenMP specification.[1].

- schedule (static) – Iterations are divided up into *chunks*, and these consecutive groups of iterations are spread among the threads using a round robin policy. This method tends to be good when you want good cache access patterns.
- schedule (dynamic) – Iterations are again split into chunks, but when a thread finished a chunk of it will request the *next* chunk. In this way you can load balance if your iterations take varying amounts of time to complete.
- private([list_of_variable_names,...]) – It is common to require variables which are local to a loop, such as the loop counter and any partial results.

The remainder of this lab will focus on the code in Listing 2, which is a serial implementation of integration using the Trapezoidal Rule. We will step through the parallelisation of this code in the tasks which follow.

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

float f(float x);

int main(int argc, char** argv) {
    int a = 0.0;        // Left endpoint
    int b = 1.0;        // Right endpoint
    long n = 1024000000;        // Number of trapezoids
    double h = (b - a) / ((double) n);                    // Base length
    double integral = (f(a) + f(b)) / 2.0;
    long i;
    double x = a;

    for (i = 1; i <= n; i++) {
        x = x + h;
        integral = integral + f(x);
    }

    integral = integral*h;

    printf("With n = %d trapezoids, estimate: %f \n", n, integral);
}

// The function to integrate, here we use 2 / (x^4 + 1)
float f(float x){
    return 2 / ((x*x*x*x) + 1);
}
```

Listing 2: Integration using the Trapezoidal Rule

[1]http://openmp.org/

Task 1 Compile the code in Listing 2 as you have done with the previous examples, execute the resultant binary, and record the values of the following: `a, b, n` and the computed area. Additionally, it is useful to be able to record the amount of time a piece of code has taken to execute, such as for comparison against other versions of the same code. OpenMP conveniently provides the following function:

- `double omp_get_wtime()` – This function returns the time elapse in seconds from some fixed point (which is consistent for multiple calls in a program run).

Using `omp_get_wtime()` instrument the code in Listing 2 such that it reports the time taken to execute the for loop and the following operation on the `integral` variable. Once this is done, execute the program and note down the runtime; you may need to adjust the value of `n` such that the code takes several seconds to complete. It may also be worth modifying the program output to report the values of `a, b` and `n` in order to make recording easier.

Task 2 Now we have a way to easily compare runs based on the input parameters, runtime and the result, we can begin parallelising this code. The obvious place to start is the for loop as this is where all the computation is done. Place the following pragma above the for loop:

- `#pragma omp parallel for private(i) schedule(static)` – Instruct the OpenMP runtime to schedule iterations of the loop statically between the available threads. Additionally, specify that the loop counter i is private, so that each thread has its own copy.

With this pragma in place, we must now consider how to modify the loop body to cope with iterations being scheduled. That is, we can no longer assume that iterations will be run in any kind of order, therefore to compute the value of x incrementally is invalid. Your next action for this task is to modify the computation of x to be in terms of `a, h` and i.

Compile and run the code, then compare the result and the runtime to that of the serial version.

Task 3 As you can see, the addition of the OpenMP pragma ruined the result. This is because the program is summing into the integral from all threads - it is an unguarded critical section. Place the following pragma above the integral summation `(integral = integral + f(x))`:

- `#pragma omp critical` – Instruct the OpenMP runtime to only allow a single thread to run this code at any given time.

Again, compile and run the code, and compare the result and runtime to that of the serial code. You will probably notice that while the result is now the same, the performance of the application has been greatly affected by the critical section.

3 Reductions

Luckily the computational pattern of summing between threads is a common one, so there is support for this in the OpenMP runtime. This type of operation is know as a reduction and is specified as another option to `#pragma omp parallel for`.

- `reduction(operator:variable)` – The reduction operator takes an argument of the form `operator:variable` where operator is for example +,-,/,* and variable is the identifier of the variable you wish to reduce over.

Task 4 Using the knowledge of reductions which you have just gained, save your code to **Task 3** in a new file and modify it to use reductions rather than critical sections. Compile and run the code, and then compare the result and runtime to that of the code from **Task 3.**

Task 5 (Optional) Using what you have learn in the lab, write a parallel implementation of the Trapezoidal Rule using OpenMP which avoids the use of a reduction every iteration. Compare this runtime of this against your implementation from **Task 4.**

References

[1] Peter Pacheco, *An Introduction to Parallel Programming, 2011.*