Assignment 1

Objectives

This assignment is an introduction to parallel programming in lower-level languages and is intended to expose you to reasoning about the parallel and sequential fractions of a program. In this assignment, we will implement a simple program for Monte-Carlo integration, parallelize it with a framework called OpenMP, and perform simple calculations to quantify the limits of parallel programming in this fashion. We will also parallelize the same program with explicit threading using pthreads (https://man7.org/linux/man-pages/man7/pthreads.7.html), and compare it against the OpenMP version in terms of programming effort and performance.

Background

Monte-Carlo Integration (MCI) is a technique for numerical integration that approximates the area underneath a curve with a non-deterministic approach. Figure 1 shows an example.

First, we pick an area that is strictly larger than the target integral (in Figure 1, this area is the unit square, bounded by the lines x=1 and y=1). Then, we randomly generate a number of points inside the domain of integration and calculate whether or not each point resides in the *target* area (1/4 of the unit circle) or the background area (the square). Therefore, the fraction of points *inside* the targeted area (in blue) to the total number of points (blue and orange) represents the area of the quarter circle.

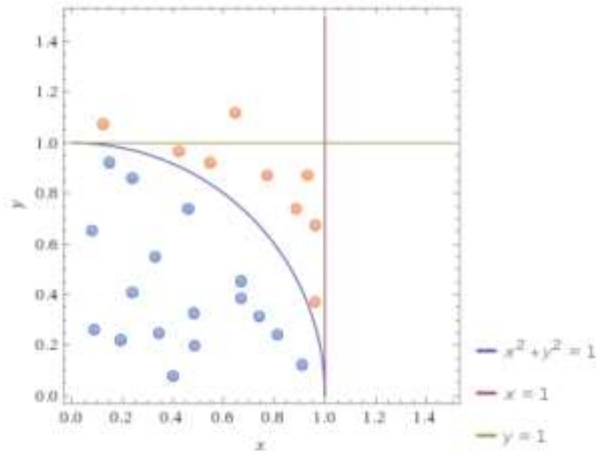

Figure 1 - Monte Carlo Integration for the Unit Circle

Assignment - Part 1

In Figure 1, we can compute an approximation of the constant π by comparing the ratio of areas between the quarter circle and the unit square to the fraction of points that we calculated are inside the quarter circle. Equation 1 computes an approximation of π, assuming the ratio of points in the quarter circle to the total number of points is 4/5.

$$A_c/_S \approx \frac{\frac{1}{4}\pi r^2}{S} \approx \pi/_4 \approx 4/_5 \quad ; \quad \pi \approx \frac{16}{5}$$

Equation 1 - Approximation of Pi. Ac is the area of the circle, S is the total area in which points were sampled, r is the circle radius

For the first part of the assignment, you write a parallel program to approximate π, following the steps below:

- Read the number of threads and the number of random samples to take from command line parameters.
- Use OpenMP to sample the points and calculate, in parallel, the total number of points that fall inside the quarter circle as described above.
- Print the computed value of π and the running time of your program to the standard output.

We will be testing your program's correctness with automated infrastructure, so it **must** comply with both the input and output specifications of our test program, which are:

1. The compiled binary must be called `pi`.
2. The first argument is the number of threads to use, and the second argument is the number of points to generate.
3. Your program must print the output in the same exact format as shown below. If it does not, you will receive no marks for the correctness portion of the assignment.

```
# /bin/bash

$ ./pi 4 500000
- Using 4 threads: pi = 3.1416045 computed in 0.405s.
```

Development Infrastructure

In the file `A1.zip`, we have provided the following files:

- `pi.c` - which contains a skeleton of how you should structure your program
- `utility.h` – which contains useful functions such as a random number generator and a timer that measures the time elapsed since the call to `set_clock()`
- `Makefile` – which compiles your program using the current version of *gcc*

Important: Ideally, you should run this program on the SCITAS cluster. But if you are temporarily working on an operating system that is non-Linux, you will need to take some extra steps to make your system work with OpenMP. On macOS, you can use Homebrew[1] to install the latest version of *gcc-7*, and then modify the `Makefile` to have *CC = gcc-7*.

[1] https://brew.sh

On Windows, you have several options. You can either use *Cygwin* to install a –Nix-like environment on your computer or use the VirtualBox VM image EPFL provides as a workspace[2] in which to perform your development.

Assignment - Part 2

In this part, you are going to repeat the approximation used in the first program while using explicit threading with the pthreads library. Please refer to the provided `pthreads_hello.c` program to understand basic pthread functionality. We provide a `pi_pthreads.c` program that is a duplicate of the original `pi.c` program. Please modify `pi_pthreads.c` as required to make it work with pthreads, and compile it to produce the `pi_pthreads` binary. The output format should be the same as in Part 1. Analyze the programming effort required to use pthreads vs. OpenMP, and study the performance difference. Discuss both these factors in your report.

Assignment - Part 3

In this part, you are going to extend the first program to provide a basic numerical integration function for arbitrary functions using MCI. The MCI algorithm to integrate any general function `f()` on the domain [a,b] is slightly different. As we don't know the range of `f()` on [a,b] a priori, and thus cannot simply bound it by a square as we did in Part 1, we leverage a well-known approximation from calculus. Say that we generate one pair `(x,f(x))` and the area of the rectangle with height `f(x)` and width `a-b`. We have just generated a rough approximation of the area under the curve, albeit a very poor one. However, Figure 2 shows how in the limit, the average area of a large number of randomly generated rectangles converges to the actual integral of `f()`.[3]

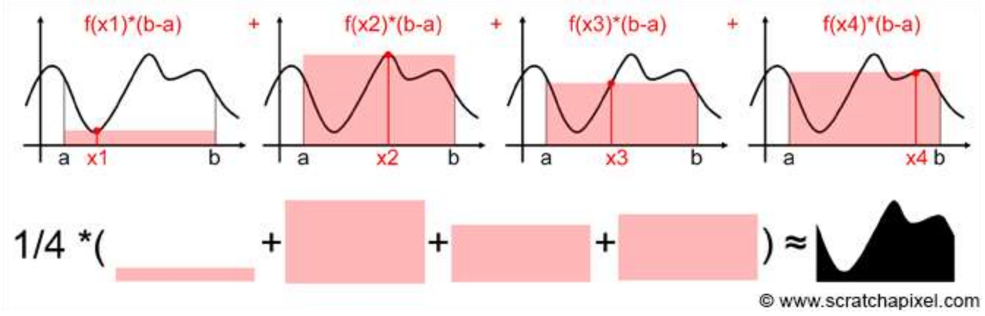

Figure 2 - MCI As a Limit of Rectangular Areas

Part 3 of the assignment requires you to implement this type of MCI using an arbitrary function as well as an arbitrary domain. For this part of the assignment, your code should be placed in the file `integral.c`, which has the same basic structure as `pi.c`. Instead of embedding the function to be integrated into the program ($x^2 + y^2 = 1$ in the first part), you need to update

[2] https://support.epfl.ch/kb_view.do?sysparm_article=KB0012496#aDownloadtheimageofthevirtualmachinerooms-jzxa

[3] For more details on this, you can visit https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-in-practice/monte-carlo-integration.

your program to operate with an abstract call to a function `f()`, which is defined in an external file. Additionally, the domain [a,b] must be passed to your code at invocation time. To test your code, you can make up your own definition of `f()`. Our tester will define the function in a file called `function.c`, so please name your test files accordingly.

In the file `integral.c`, please write the function:
`integrate(int num_threads,int samples,int a,int b, double (*f)(double) )`

The arguments represent:

- `num_threads` – the number of threads to use
- `samples` – the number of total samples to evaluate
- `a,b` – the domain of integration
- `f` – a function pointer that takes in a double *x* and returns *f(x)*

Your program must format its output **exactly** as follows:

```
# /bin/bash

$ ./integral 4 2000 5 9
- Using 4 threads: integral on [5,9] = 4.1416045 computed in 0.405s.
```

Deliverables

You need to turn in the following things for this assignment:

1. Completed code for `pi.c, pi_pthreads.c` and `integral.c`. Remember to check if your code complies to the format expected by the automated tester; otherwise, you will receive no points for correctness!
2. A report that answers the questions listed below. The report name should be `a1_GROUPID.pdf`

To submit your code and your report, create a tarball archive (**this is the only accepted format!**) called `a1_GROUPID.tgz` and upload it on Moodle[4].

Report

In the report, we expect you to perform four tasks:

1- Describe the algorithm that was implemented. Identify (for Part 1 and 3):
   a. Which parts of the program you parallelized and why. If the algorithm has several phases (parallel phases followed by serial phases), identify each of them.

[4] If you don't know how, look online for a guide like this one: https://www.howtogeek.com/248780/how-to-compress-and-extract-files-using-the-tar-command-on-linux/

    b. Identify the operations that dominate the execution time of each program phase. You should pick a single type of operation per phase. (i.e., If you believe that your program spends most of its time performing multiplications, then that operation is multiplication).
    c. For each phase, identify the program arguments that affect the number of performance-critical operations in each phase and provide the asymptotic execution time of the program in the big O notation.
    d. Estimate the speedup of the multithreaded program over the single-threaded version of it, using thread counts equal to {1, 2, 4, 8, 16, 32, 48, 64} and draw a graph showing how the execution time scales with the number threads. Ignore any hardware limitations in your estimation.

2- In a table, present the execution times and speedups over the single-threaded version you measured for the same thread counts as in 1d. Then add these results to the previous graph to see how your prediction matches with the actual execution times. Run all of your results on the SCITAS cluster using the job submission system explained in the exercise session, as well as the link on Moodle.

3- Compare the speedups you measured on the SCITAS machines to what you predicted in Task 1. If they are different, why?

4- Discuss the programming effort and performance difference between OpenMP and pthreads. When would it make sense to use explicit threading?

The report should be as precise as possible, addressing the four questions above. Keep the descriptions of the algorithm implemented and of your reasoning in parallelizing the program short. A regular report should not be much longer than 2 pages.

Grading

The code will be graded automatically by a script and checked for plagiarism. The script will check how the running time scales with the number of threads and if the results returned are consistent with what was expected. Plagiarized code will receive 0. We will escalate plagiarism cases to the student section.

The reports will be read and graded by humans and checked for plagiarism automatically.

The grade breakdown is as follows:

- Correctness: 50%
- Report: 50%