Assignment 1

Objectives

This assignment serves a basic 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.

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 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. The mathematical properties of MCI (and its relative error function) are well known and you can investigate them further online[1] if you choose.

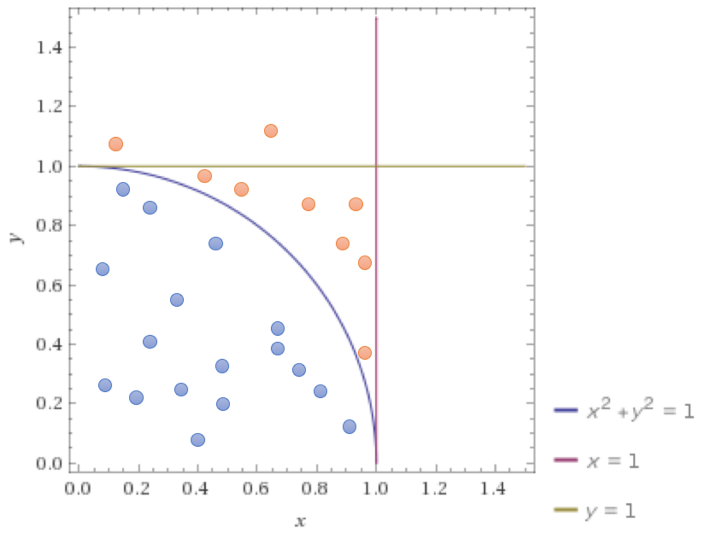

Figure 1 - Monte Carlo Integration for the Unit Circle

[1] http://mathworld.wolfram.com/MonteCarloIntegration.html

Assignment - Part 1

Notice that in the case shown in Figure 1, we can gain 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 shows how we obtain the approximation $\pi = 3.2$, assuming the ratio of points contained by the quarter-circle to the total number of points is 4/5.

$$A_c/_S \approx \frac{1}{4}\pi 1^2/_1 \approx 4/_5 \quad ; \quad \pi \approx \frac{16}{5}$$

Equation 1 - Approximation of Pi

For the first part of the assignment, you will have to write a parallel program to approximate π, that does the following basic tasks:

- Reads the number of threads to use, and the number of random samples to calculate, from command line parameters.
- Uses OpenMP to calculate, in parallel, the total number of points that fall inside the quarter-circle as described above.
- Prints an approximation of π and the time taken in your program, to standard output.

We will be testing your program’s correctness with automated infrastructure, so it **must** comply to 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 **exactly** the sample output 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 7.405s.
```

Development Infrastructure

On the course webpage, we have provided the tarball *A1.tgz,* which contains 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: If you are 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*[2] to install the latest version of *gcc-7*, and then modify the `Makefile` to read:

```
CC = gcc-7
```

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 we provide as a workspace in which to perform your development.

Assignment - Part 2

In this part of the assignment, we are going to extend the first program to provide a basic numerical integration function for arbitrary functions using MCI. You can find many online resources on the formal methods for MCI. To integrate any general function `f()`, on the domain [a,b], the process for MCI 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()`. See https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-in-practice/monte-carlo-integration for more details.

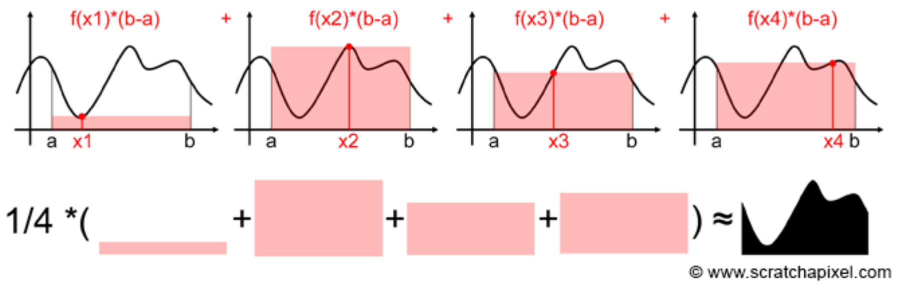

Figure 2 - MCI As a Limit of Rectangular Areas[3]

Part 2 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

[2] https://brew.sh
[3] Taken from https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-in-practice/monte-carlo-integration

function to be integrated into the program ($x^2 + y^2 = 1$ in the first part), you need to update 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. In order 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 various 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] = <NUM> computed in 7.405s.
```

Deliverables to Hand In

For Assignment 1, you need to turn in the following things:

1. Completed code for `pi.c` and `integral.c`. Remember to check that 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 following questions and explains your findings during development. The report name should be `a1_yourSCIPER.pdf`

To submit your code and your report, create a tarball archive called `a1_yourSCIPER.tgz` and upload it on Moodle[4].

[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/

Report

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

1- Explain how you solved each problem (of Parts 1 & 2) in a parallel fashion, what parts of the program remain serialized, and how will performance improve with parallelization in the context of Amdahl’s Law. Feel free to roughly estimate the speedup you expect, and you don’t need to provide real run-times for this task. The purpose of the exercise is to get you to reason about the speedup of the program based on how you parallelized it.

2- In a table, present the execution times, and speedups over the single threaded version, you measured for the following set of thread counts {1,2,4,8,16,32,48,64}. 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 the course webpage.

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