NOVA SCHOOL OF SCIENCE & TECHNOLOGY
COMPUTER SCIENCE DEPARTMENT

Concurrency and Parallelism 2022-23 Lab 03

Lab 3: Calculation of π Using the Monte Carlo Method using C and OpenMP

Hervé Paulino & Alex Davidson

March 2023

Abstract

In the this third lab assignment, you will recall some basic concepts about developing concurrent/parallel programs using the `C` programming language, when extended with the `OpenMP` programming framework.

1 Introduction

In this assignment we will be using the OpenMP framework to allow us to write programs that execute certain functionality across multiple threads. OpenMP is a framework that is compatible with C, C++, and Fortran, amongst other languages. In the case of C, it allows writing compliant C code, with certain annotations that tell the compiler to split activities across a defined number of threads.

1.1 Background reading

Start by studying OpenMP:

`https://www.openmp.org/resources/tutorials-articles/`

The following slides (up to page 20) and the proposed exercises (up to page 20) are useful:

`https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf`

These slides also provide some specific example implementations while writing to different types of global variables:

`https://courses.grainger.illinois.edu/cs484/sp2020/6_merged.pdf`

You may also follow this tutorial in youtube:

`https://www.youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG`

And/or study this book:

`https://libgen.lc/ads.php?md5=D27D7EE55FA748118FEA185BC27D6FB5`

Then step into the next section.

2 OpenMP Parallel Version

Open the following link, and log on to GitHub if requested

`https://classroom.github.com/a/LQwJ3YEq`

and then follow the instructions below.

The given C program is a working version of a sequential C program to calculate Pi using the Monte Carlo method. You may start from this version or replace it with your own version. The README.md file in your repository provides instructions on how to compile this code (see below for more details). Running the code follows the same instructions as in the previous assignment.

You may compile the C source file with:

```
gcc -O3 -fopenmp -o approxPi approxPi.c
```

of course, the given code is a sequential version and the number of threads will not affect the behaviour of the program.

2.1 Main task

Use OpenMP to parallelize your program. The following two-step process may be the best approach:

1. an OpenMP parallel version that makes as few changes as possible of the base sequential version;
2. an OpenMP parallel version that is optimized for speed.

To set the number of OMP threads, you may use:

```
omp_set_dynamic(0);
omp_set_num_threads(nThreads);
```

Remember to keep on using GIT. :)

2.2 Executing the Parallel Version

The program accepts 2 command line options and 2 mandatory arguments.

The usage is:

```
./approxPi [-d] [-s] nPoints nThreads
            -d: Enable debugging more info is printed on the screen.
            -s: Silent mode just print numbers with no explanation
                of their meaning.
       nPoints: The total number of points to be used in the simulation
                (to be divided by the working threads).
     nThreads: The number of working threads.
```

You can also use the provided `run-tests.sh` file.

3 Task: Analysing and Visualising Results

Once your program is running correctly:

1. Try your program with some values (e.g., 100, 1000, 10000, etc) and build a plot with the execution time (t) as a function of the number of points (n) and of the number of processors/threads (p). **Remember to run each configuration in a way that allows you to establish scientific certainty of your results.**[1]
2. Answer the following questions:
   - The parallel version with two threads is faster than with one?
   - When you double the number of threads does it take half the time? More? Less?
   - Which of your implementations is faster? Java, C, or using OpenMP? Why?

3.1 Performance Profiling

In order to answer the questions above, it is necessary to understand exactly how your code works. This requires *profiling* your code. In the README.md file of your repository, instructions are provided on how to profile your code using the `pprof` tool. You may also use alternatives such as `gprof`. The final part of this assignment involves producing visualisations that help to convince both yourself, and others, of why your code execution performs as it does.

Using `pprof`, try constructing flowcharts or *flame graphs* to visualise your program. Can you see which parts of your program take longest to execute?

References

[1] A. M. Johansen. "Monte Carlo Methods". In: *International Encyclopedia of Education (Third Edition)*. Ed. by Penelope Peterson, Eva Baker, and Barry McGaw. Third Edition. Oxford: Elsevier, 2010, pp. 296–303. ISBN: 978-0-08-044894-7. DOI: 10.1016/B978-0-08-044894-7.01543-8.

[2] S. L. Zabell. "Alan Turing and the Central Limit Theorem". In: *The American Mathematical Monthly* 102.6 (1995), pp. 483–494. ISSN: 00029890, 19300972. URL: http://www.jstor.org/stable/2974762.

Appendix

A Recall: Monte Carlo Methods

Monte Carlo methods [1] can be thought of as statistical simulation methods that utilize a sequence of random numbers to perform some kind of simulation. The name Monte Carlo was coined by Nicholas Constantine Metropolis (1915-1999) and inspired by Stanislaw Ulam (1909-1986), because of the similarity of statistical simulation to games of chance, and because Monte Carlo is a centre for gambling and games of chance. In a typical two-dimensional

[1] For example, think about the number of times that you run it, and under what conditions the program should run.

process, one computes the number of points within a shape A that lies inside a larger shape R. The ratio of the number of points that fall inside A to the total number of points tried is equal to the ratio of the two areas (or volume in a three-dimensional process). The accuracy of the ratio ρ depends on the number of points used, with more points leading to a more accurate value.

A.1 Estimating the value of π

Figure 1 shows a circle with radius $r = 1$ inscribed within a square. The area of the circle is

$$\mathrm{A}_{\bigcirc} = \pi \cdot r^2 = \pi \cdot 1^2 = \pi$$

and the area of the square is

$$\mathrm{A}_{\square} = (2 \cdot r)^2 = 2^2 = 4.$$

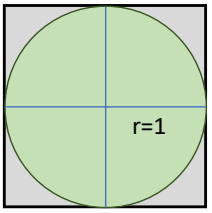

Figure 1: A circle within a square.

The ratio ρ of the area of the circle to the area of the square is

$$\rho = \frac{\mathrm{A}_{\bigcirc}}{\mathrm{A}_{\square}} = \frac{\pi}{4} = 0.7853981634.$$

A simple Monte Carlo simulation to approximate the value of π can first estimate ρ and then π. Given ρ then

$$\pi = \rho \times 4 = 0.7853981634 \times 4 = 3.1415926536.$$

One particularly simple way to compute ρ is to pick random points in the square and count how many of them lie inside the circle Figure 2).

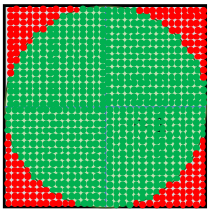

Figure 2: Ratio between areas of circle and square.

A Monte Carlo simulation to approximate the value of π will then involve randomly selecting points $(x_i, y_i)_{i=1}^{n}$ in the unit square and determining the ratio $\rho = \frac{m}{n}$, where n is the total number of points and m is number of points that satisfy $x_i^2 + y_i^2 \leq 1$ (see Figure 3).

In a simulation of sample size $n = 1000$, there were 787 points satisfying that equation. Using this data we obtain

$$\rho = \frac{m}{n} = \frac{787}{1000} = 0.787$$

and

$$\pi = \mathrm{A}_{\bigcirc} = \rho \times \mathrm{A}_{\square} = \rho \times 4 = 0.787 \times 4 = 3.148.$$

If the numbers used in the Monte Carlo simulation are random, every time a simulation is made using the same sample size n, it will arrive at a slightly different value. The values converge very slowly with order $O(n^{-1/2})$. This property is a consequence of the Central Limit Theorem [2].

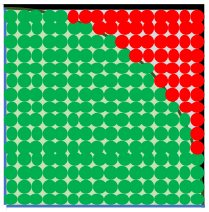

Figure 3: Monte Carlo method.

You may find a web simulation of this method at
`https://academo.org/demos/estimating-pi-monte-carlo/`.