NOVA SCHOOL OF SCIENCE & TECHNOLOGY
COMPUTER SCIENCE DEPARTMENT

Concurrency and Parallelism 2022-23

Lab 2: Calculation of π Using the Monte Carlo Method using C

Hervé Paulino & Alex Davidson

March 2023

Abstract

With this second lab assignment, you will learn/recall some basic concepts about developing concurrent/parallel programs using the C programming language and the `pThreads` interface and library.

1 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 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.

1.1 Estimating the value of π

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

$$A_{\bigcirc} = \pi \cdot r^2 = \pi \cdot 1^2 = \pi$$

and the area of the square is

$$A_{\square} = (2 \cdot r)^2 = 2^2 = 4.$$

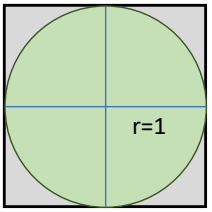

Figure 1: A circle within a square.

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

$$\rho = \frac{A_{\bigcirc}}{A_{\square}} = \frac{\pi}{4} = 0.7853981634.$$

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).

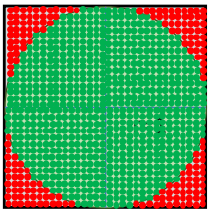

Figure 2: Ratio between areas of circle and square.

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].

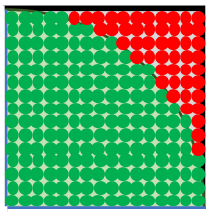

Figure 3: Monte Carlo method.

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

2 Lab Work

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

`https://classroom.github.com/a/Z2b6gvy0`

and then follow the instructions below.

In the give C program, you are expected to implement the functions

```
void *worker (void *arg)
```

and

```
double calculate_pi (long nPoints, int nThreads)
```

You may compile the C source file with:

```
gcc -o approxPi approxPi.c -lpthread
```

2.1 Parallel Version of the Monte Carlo Simulation

Design and implement a C (parallel) program named `approxPi` that makes use of the `pThreads` library to approximate the value of π by using the Monte Carlo method. In the command line, the program accepts 2 options and must receive 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.
```

Examples below (the `$` is the shell prompt and is not part of the command):

```
$ ./approxPi 100000000 8
Pi: 3.312500000
Real Time: 0.261323
 CPU Time: 0.261323

$ ./approxPi -s -d 100000000 8
 nPoints: 100000000
nThreads: 8
  Worker: 12500000
  Worker: 12500000
  Worker: 12500000
  Worker: 12500000
  Worker: 12500000
  Worker: 12500000
  Worker: 12500000
  Worker: 12500000
     Win: 12500000
     Win: 12500000
     Win: 10156250
     Win: 10937500
     Win: 10937500
     Win: 11718750
     Win: 10937500
     Win: 11718750
nPnt       nThr  Real      CPU       Pi
100000000  8     0.267139  1.902000  3.656250000

$ ./approxPi -s 100000000 8
100000000 8 0.250770 1.889024 3.437500000
```

You may use the script `run_tests.sh` given in the repository to help executing the tests. The options given to the script will be passed to the `approxPi` program. For example, you may run it as:

```
$ ./run_tests.sh -s
```

Remark: *Remember to keep using these simple lab exercises to learn how to use `git`. Use the given repository (and the give starting code) to manage the versioning of your code. Remember to do frequent commits with clear commit messages.*

2.2 Analysing 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? C or Java? Why?

2.3 Extension: Performance Profiling

To go deeper into analysing why the performance of your program is a certain way, you will need to spend some time profiling your compiled code to see which functions take the longest. You can a variety of tools for doing, for example the CLion IDE allows you to profile certain things directly. On the command line, there are a variety of tools that can help you profile and visualise your program at a functional level (such as `perf`, `dtrace`, etc.) This blog post explains some different ways of doing this.

Final Note

There is a large bibliography of work on this problem, the corresponding Monte Carlo simulation, and its implementation in a variety of programming languages. Please understand that the objective of this homework is for you to learn about concurrency and this lab assignment is **not** graded. So, there is no value in cheating by searching the web for working solutions of this problem! *Just be honest with yourself and develop your own solution, or just ignore it!*

Acknowledgments

The text from the first two sections is an adaptation of the text found at `http://mathfaculty.fullerton.edu/mathews/n2003/montecarlopimod.html`.

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.

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