Programming Assignment II: Multi-thread Programming

Parallel Programming by Prof. Yi-Ping You

Due date: 23:59, Nov 3, Thursday, 2022

The purpose of this assignment is to familiarize yourself with Pthread and std::thread programming in C and C++, respectively. You will also gain experience measuring and reasoning about the performance of parallel programs (a challenging, but important, skill you will use throughout this class). This assignment involves only a small amount of programming, but a lot of analysis!

TABLE OF CONTENTS

---

1. Part 1: Parallel Counting PI Using Pthreads

1.1 Problem Statement

This is a follow-up assignment from Assignment ZERO. You are asked to turn the serial program into a Pthreads program that uses a Monte Carlo method to estimate PI. The main thread should read in the total number of tosses and print the estimate. You may want to use `long long ints` for the number of hits in the circle and the number of tosses, since both may have to be very large to get a reasonable estimate of PI.

Your mission is to make the program as fast as possible. You may consider a lot of methods to improve the speed, such as SIMD intrinsics or a faster random number generator. However, you cannot break the following rules: you need to implement the Monte Carlo method using Pthreads.

Hint: You may want to use a reentrant and thread-safe random number generator.

You are allowed to use third-party libraries in this part, such as pseudorandom number generators or SIMD intrinsics.

1.2 Requirements

- Typing `make` in the `part1` directory should build the code and create an executable called `pi.out`.
- `pi.out` takes two command-line arguments, which indicate the number of threads and the number of tosses, respectively. The value of the first and second arguments will not exceed the range of `int` and `long long int`, respectively. `pi.out` should work well for all legal inputs.
- `pi.out` should output (to stdout) the estimated PI value, which is accurate to three decimal places (i.e., 3.141xxx) with at least `1e8` tosses.

Example:

```
$ make && ./pi.out 8 1000000000
3.1415926....
```

2. Part 2: Parallel Fractal Generation Using std::thread

2.1 Problem Statement

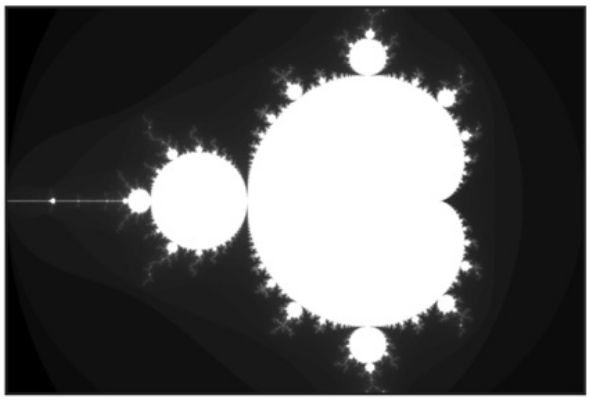
```
$ cd <your_workplace>
$ wget http://nycu-sslab.github.io/PP-f22/assignments/HW2/HW2.zip
$ unzip HW2.zip -d HW2
```

```
$ cd HW2/part2
```

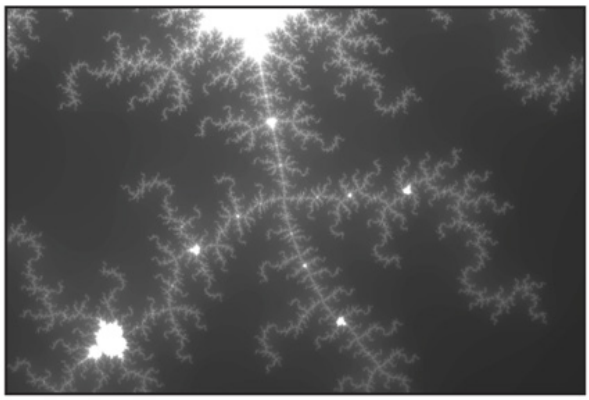
Build and run the code in the `part2` directory of the code base. (Type `make` to build, and `./mandelbrot` to run it. `./mandelbrot --help` displays the usage information.)

This program produces the image file `mandelbrot-serial.ppm`, which is a visualization of a famous set of complex numbers called the Mandelbrot set. [Most platforms have a `.ppm` viewer. For example, to view the resulting images, use tiv command (already installed) to display them on the terminal.]

As you can see in the images below, the result is a familiar and beautiful fractal. Each pixel in the image corresponds to a value in the complex plane, and the brightness of each pixel is proportional to the computational cost of determining whether the value is contained in the Mandelbrot set. To get image 2, use the command option `--view 2`. (See function `mandelbrotSerial()` defined in `mandelbrotSerial.cpp`). You can learn more about the definition of the Mandelbrot set.



View 1



View 2
(66x zoom)

Your job is to parallelize the computation of the images using std::thread. Starter code that spawns one additional thread is provided in the function `mandelbrotThread()` located in `mandelbrotThread.cpp`. In this function, the main application thread creates another additional thread using the constructor `std::thread` (function, args...). It waits for this thread to complete by calling `join` on the thread object.

Currently the launched thread does not do any computation and returns immediately. You should add code to `workerThreadStart` function to accomplish this task. You will not need to make use of any other `std::thread` API calls in this assignment.

2.2 Requirements

You will need to meet the following requirements and answer the questions (marked with “Q1-Q4”) in a REPORT using HackMD.

1Modify the starter code to parallelize the Mandelbrot generation using two processors. Specifically, compute the top half of the image in thread 0, and the bottom half of the image in thread 1. This type of problem decomposition is referred to as *spatial decomposition* since different spatial regions of the image are computed by different processors.

2Extend your code to use 2, 3, 4 threads, partitioning the image generation work accordingly (threads should get blocks of the image). Q1: In your write-up, produce a graph of speedup compared to the reference sequential implementation as a function of the number of threads used FOR VIEW 1. Is speedup linear in the number of threads used? In your writeup hypothesize why this is (or is not) the case? (You may also wish to produce a graph for VIEW 2 to help you come up with a good answer. Hint: take a careful look at the three-thread data-point.)

3To confirm (or disprove) your hypothesis, measure the amount of time each thread requires to complete its work by inserting timing code at the beginning and end of `workerThreadStart()`. Q2: How do your measurements explain the speedup graph you previously created?

4Modify the mapping of work to threads to achieve to improve speedup to at about 3-4x on both views of the Mandelbrot set (if you’re above 3.5x that’s fine, don’t sweat it). You may not use any synchronization between threads in your solution. We are expecting you to come up with a single work *decomposition policy* that will work well for all thread counts—hard coding a solution specific to each configuration is not allowed! (Hint: There is a very simple static assignment that will achieve this goal, and no communication/synchronization among threads is necessary.). Q3: In your write-up, describe your approach to parallelization and report the final 4-thread speedup obtained.

5Q4: Now run your improved code with eight threads. Is performance noticeably greater than when running with four threads? Why or why not? (Notice that the workstation server provides 4 cores 4 threads.)

3. Grading Policy

NO CHEATING!! You will receive no credit if you are found cheating. Don’t take any chances. 😉

Total of 100%:

- Part 1 (40%):

- Correctness (30%): The requirements should be met, and your parallelized program should run faster than the original (serial) program. Notice that you will receive no credit if one of the two aforementioned conditions fails.
- Scalability (30%): We will evaluate your program with 2, 3, 4 or more threads. Your program is expected to be scalable.
- Performance (40%): See the metrics below for details. The maximum time limit for running `$time (./pi.out 3 100000000; ./pi.out 4 100000000)` on the workstation is `T=1.00`.

- Part 2 (60%):
  - Correctness (30%): Your parallelized program should pass the verification (written in `main.cpp`) and run faster than the original (serial) program. Notice that you are not allowed to modify files other than `mandelbrotThread.cpp` and you will receive no credit if one of the two aforementioned conditions fails.
  - Performance (30%): See the metrics below for details. The maximum time limit for running `$./mandelbrot -t 3` plus `$./mandelbrot -t 4` on the workstation is `T=0.375` (considering only the time of `mandelbrot thread`).
  - Questions (40%): Each question contributes 10%. Answers to each question will be classified into one of the four reward tiers: excellent (10%), good (7%), normal (3%), and terrible (0%).

  Performance metrics (100%):

  - Maximum time limit (60%): Your program need to run within the running time threshold, `T` seconds.
  - Competitiveness (40%): You will compete with the “fast threshold” to get the other 40%: Your score is calculated by `(T-Y)/(T-F) * 40%`, where `Y`, `T`, and `F` indicate the execution time of your program, the threshold, and the fast threshold, respectively (F = 0.5 for part 1, 0.28 for part 2).

4. Evaluation Platform

Your program should be able to run on UNIX-like OS platforms. We will evaluate your programs on the workstations dedicated for this course. You can access these workstations by `ssh` with the following information (`g++-10` and `clang++-11` have been installed).

IP	Port	User Name	Password
140.113.215.197	10002-10010	{student_id}	{your_passwd}

Login example:

```
$ ssh <student_id>@140.113.215.197 -p <port>
```

5. Submission

All your files should be organized in the following hierarchy and zipped into a `.zip` file, named `HW2_xxxxxxx.zip`, where `xxxxxxx` is your student ID.

Directory structure inside the zipped file:

- `HW2_xxxxxxx.zip` (root)
  - `part1` (directory)
    - `Makefile`
    - `pi.c` or `pi.cpp`
    - other C/C++ files
  - `part2` (directory)
    - `mandelbrotThread.cpp`
  - `url.txt`

Notice that you just need to provide the URL of your HackMD report in `url.txt`, and enable the write permission for someone who knows the URL so that TAs can give you feedback directly in your report.

You can use the testing script to check your answer *for reference only*. Run `test_hw2` in a directory that contains your `HW2_XXXXXXX.zip` file.

```
$ test_hw2
```

Please make sure you pass the testing script and then upload your zipped file to new E3 e-Campus system by the due date.

You will get *NO POINT* if your ZIP's name is wrong or the ZIP hierarchy is incorrect.

6. References

There are some some resources for you:

- POSIX Threads Programming

- 簡易 Pthreads 平行化範例與效能分析
- 深入理解 Reentrancy 和 Thread-safe

Back to top