CS 420, Parallel Programming, Spring 2023
MP2

Due Date: March 10th, 2023, 11:59PM CDT

1 Overview

The purpose of this assignment is straightforward – for you to gain experience using `c++11 threads` and OpenMP, and assess the performance of a parallel program. To achieve this, you will parallelize an implementation of the Mandelbrot Sequence and benchmark it.

2 Getting Started

2.1 Skeleton Programs

Pull the skeleton for MP2 from the `release` repository: https://gitlab.engr.illinois.edu/sp23-cs420/turnin/<NETID>/mp2.git

3 Parallelizing the Mandelbrot Set

3.1 The Mandelbrot Set

The Mandelbrot Set is a set of complex numbers whose computation represents an embarrassingly parallel problem; this means that no synchronization or communication among threads is required. We have provided a skeleton program, `mandelbrot`, that contains a serial implementation of the Mandelbrot Set which produces a PNG image, an example of which is shown in Figure 1. The skeleton program can be found in `tools/mandelbrotTools.cpp`. The parameters for the program are as follows:

- `-o <file>` – Specify the output file (by default: `out.png`).
- `-w <width>` – Specify the width of the generated image in pixels (by default 480).
- `-h <height>` – Specify the height of the generated image in pixels (by default 480).
- `-n <N>` – Run the program with N threads (your task is to implement this by completing `parallelMandelbrot` function).
- `-O` – Run the program with OMP (your task is to implement this functionality by completing `OMPMandelbrot` function).

This program will later be used by the `batch_script.run` file to generate images of multiple dimensions.

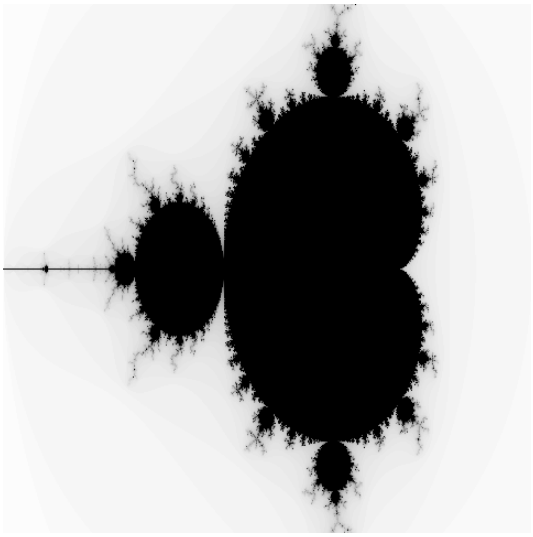

Figure 1: An example image of the Mandelbrot Set.

3.2 Your Task

In the sections marked in the `mandelbrot.cpp`, fill out the functions `parallelMandelbrot` and `OMPMandelbrot`.

In `parallelMandelbrot` function, create `kData.numThreads` threads and wait for their completion; hint, you could use `std::thread` and `std::thread::join`.

These threads should run `generateMandelbrot` with `startX` and `endX` representing a unique, non-overlapping sub-range of `0` to `kData.width`. You are free to divide up this workload anyway you choose; however, it is worth noting that the number of threads may not evenly divide with the image width (i.e. `kData.width`). *You should use* `generateMandelbrot` *in the function that is run by every thread. An example use of this function can be found in* `main.cpp line 70`

Note: A sample program that sums the elements of an array and uses `std::thread` **and** `std::thread::join` **is included as comments in the** `mandelbrot.cpp` **file.** The sole purpose of this example program is for you to refer to the syntax for `std::thread` and `std::thread::join` functions if necessary.

Alternatively, if interested you could also use `pthread` functions to achieve the same functionality. But we recommend using `std::threads`

In `OMPMandelbrot` function, use `OMP` pragmas to parallelize the included loop.

3.2.1 Building the program

We have provided a `cmake` files and a script for you to use to build your program. To build, enter your root directory and run:

```
. scripts/compile_script.sh
```

Please note the command is slightly different this MP
This will make the programs in `build/mandelbrot`. If you want to manually build the program, please run the commands below.

```
module load intel/18.0
module load gcc/7.2.0
module load cmake/3.18.4

mkdir build
cd build
cmake ..
cmake --build .
```

3.2.2 Testing Correctness

Once you have implemented your parallel version, you can test its correctness with a small script that we have included. It will test your implementation against some ground truth images and check for any discrepancies. You can run the test with:

```
./tests/test.sh
```

This will run the program with a number of test cases, and if correct, produce results along the lines of:

```
Running test 1
Running test 2
Running test 3
Running test 4
Running test 5
Cleaning pngs

All tests passed
```

3.2.3 Testing Multi-threading

Once you have implemented your parallel version and verified its correctness, you are to run the small benchmarking program we have included. You are required to compile the program first in order to benchmark. Run the benchmark with:

```
python benchmark.py
```

This will run the program with a number of threads, and produce results along the lines of:

```
Generating a XxX mandelbrot with thread counts: [1, 2, 4]
The serial version ran for X s.

The parallel version, with 1 core(s), ran for X s, a speedup of Xx.
The parallel version, with 2 core(s), ran for X s, a speedup of Xx.
The parallel version, with 4 core(s), ran for X s, a speedup of Xx.
```

NOTE: This is only a small benchmark to test whether you program correctly multi-threads. This is NOT the benchmark that you will using for measuring speedup. Results may vary between runs and may not always show an improvement even with correct implementation. Larger image sizes (∼1024/2048) are more reliable in showing improvement if present.

3.2.4 Benchmarking

Once you have implemented your parallel version and verified its functionality, you are to submit the benchmarking jobs we have included. If you are on the Campus Cluster, the you can submit the jobs with:

```
./scripts/submit_bash.sh
```

This will run the program with a number of threads with a variety of image sizes. The files it produces will have following the naming scheme of

```
benchmark_XxX.txt
```

Where XxX represents the size of the image used. The contents of the files will be in the format:

```
Generating a XxX mandelbrot with core counts: [1, 2, 4, ..., 24, 32, 40, 64]
The serial version ran for X s.

The parallel version, with 1 core(s), ran for X s, a speedup of Xx.
The parallel version, with 2 core(s), ran for X s, a speedup of Xx.
The parallel version, with 4 core(s), ran for X s, a speedup of Xx.
...
The OpenMP version, with 64 core(s), ran for X s, a speedup of Xx.
```

Question 1: Briefly comment about how changing the data size affects the performance of the parallel threaded version with four threads, and include the output of the benchmarking program for data size 2048 in your report.

Question 2: For data size 2048... Plot *Speedup vs. Number of Cores* using the results of the benchmark program. Be sure to specify which is OpenMP and the manually threaded.

Question 3: For data size 2048... How did the performance of the parallel version with multiple threads compare to serial version? Briefly explain your results.

4 Submission Guidelines

4.1 Expectations For This Assignment

The graded portions of this assignment are your parallelized Mandelbrot program and answers to the short-answer questions. Submit your report to GradeScope.

4.1.1 Points Breakdown

- $3 * 16.\bar{6} pts.$ — Each Short Answer Question
- $50 pts.$ — Parallelized Mandelbrot Program

4.2 Pushing Your Submission

Follow the git commands to commit and push your changes to the remote repo and submit the writeup through Gradescope. Ensure that your files are visible on the GitLab web interface, your work will not be graded otherwise. Only the most recent commit before the deadline will be graded.

4.2.1 Files to change

- `mandelbrot.h`
- `mandelbrot.cpp`

5 References

We encourage you to start with the following links if you would like any additional information about the topics discussed in this MP:

- https://en.cppreference.com/w/cpp/thread/thread
- https://en.wikipedia.org/wiki/Embarrassingly_parallel
- https://www.geeksforgeeks.org/fractals-in-cc/
- http://books.gigatux.nl/mirror/kerneldevelopment/0672327201/ch09lev1sec1.html
- https://en.cppreference.com/w/c/atomic