TÉCNICO
LISBOA

Assignment 1 (part 1)

High Performance Computing Systems

Sistemas de Computação de Elevado Desempenho (SCED)

2020/2021

Overview

This assignment is intended to help you develop an understanding of the two primary forms of parallel execution present in a modern multi-core CPU:

1. Parallel execution using multiple cores (You'll see effects of Intel Hyper-threading as well.)

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!

Deadlines and Submission

Next week, we will distribute the second part of Lab1. You will submit both parts of Lab1 via Fenix (until 11th of October at 23h59, i.e., Sunday).

The submission should be made in a single `zip` file with the following content:

- Your report (writeup) in a file called `report.pdf` (it must be pdf)
- Your implementation (codes) for all programs (please do `make clean` before submission)

Your report and codes should correspond to both parts of Lab1. When handed in, all codes must be compilable and runnable out of the box on the `adriana` machine!

Your report should have at most 10 pages and should be well structured. For each problem (program) you should: elaborate your solution, explain your rationale (how did you arrive there, why it makes sense, describe all (if any) specific measurements/experiments that were conducted by you to prove your hypothesis etc). You should also discuss and analyze the obtained results!

Environment Setup

For this assignment, you will need to run (the final version of) your code on `adriana` machine (hostname: `adriana.inesc-id.pt`). This machine contains four-core 3.5 GHz Intel Core i7 processor (although dynamic frequency scaling can take it to 3.9 GHz when the chip decides it is useful and possible to do so). Each core in the processor can execute AVX2 vector instructions which describe simultaneous execution of the same operation on multiple single-precision data values. For the curious, a complete specification for this CPU can be found at https://ark.intel.com/.../intel-core-i7-3770k-processor-8m-cache-up-to-3-90-ghz.html.

The access to `adriana` machine is performed through `ssh`, by using the command:

```
ssh HPCS_g<group_number>@adriana.inesc-id.pt
```

where you should substitute *<group_number>* with your group number (e.g., the username for the group number 3 is `HPCS_g3`). The passwords for each group have the flowing format `g<group_number>hpcs` (again, the password for the group number 3 is `g3hpcs`).

<u>IMPORTANT</u>: Since the machine is a shared resource, please verify if nobody else is performing experiments before running your program, in order not to affect the results of other groups (or even other researchers from INESC-ID). To check the users logged in the machine, you can use the **`who`** command. It is highly recommended to schedule the utilization of `adriana` among yourselves to avoid the simultaneous use of the machine. (you should also think why it is not a good idea that all of you run the programs at the same time – given the characteristics of `adriana`'s CPU – and how it can impact the execution of your program)

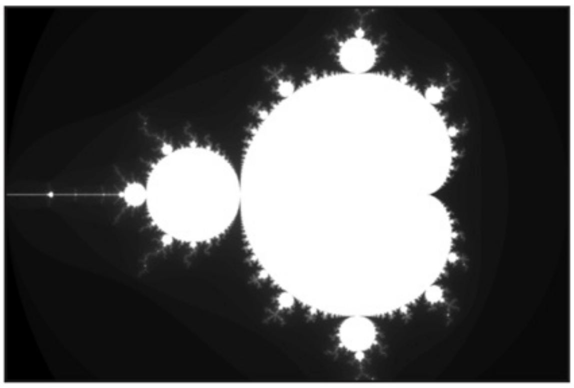
Note: For grading purposes, we expect you to report on the performance of code run on the `adriana` machine. However, for development, you may also want to run the programs in this assignment on your own machine. Feel free to include your findings from running code on other machines in your report as well, just be very clear what machine you were running on.

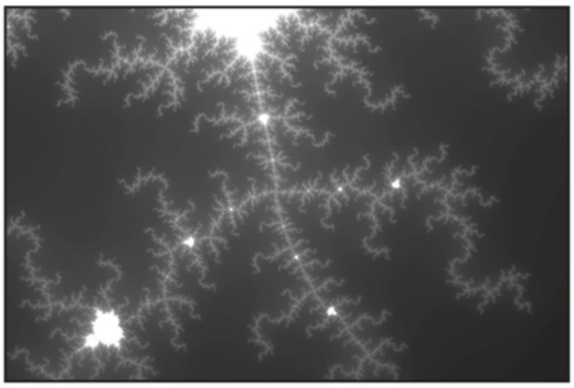
To get started:

1. The assignment starter code is available on the course webpage (section: Labs). `wget` it, `unzip` it and start having fun with it.

Program 1: Parallel Fractal Generation Using Threads

Build and run the code in the `prog1_mandelbrot_threads/` directory of the code base. (Type `make` to build, and `./mandelbrot` to run it.) 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` view. For example, to view the resulting images remotely, use `ssh -X` at the log in and the `display` command. Depending on your operating system, you might also need a X11 display server to use this command (like `XQuartz` for macOS, or `Xming` for Windows). 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 <u>the brightness of each pixel is proportional to the computational cost</u> 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 at http://en.wikipedia.org/wiki/Mandelbrot_set.


View 1


View 2
(66x zoom)

Your job is to parallelize the computation of the images using std::thread (the "thing" we elaborated on our previous lecture). 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.

What you need to do:

1. Modify 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.
2. Extend your code to use 2, 3, 4, 5, 6, 7, and 8 threads, partitioning the image generation work accordingly (threads should get blocks of the image). Note that the processor only has four cores but each core supports two hyper-threads, so it can execute a total of eight threads interleaved on its execution contents. 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 datapoint.)
3. To 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()`. How do your measurements explain the speedup graph you previously created?
4. Modify the mapping of work to threads to improve speedup to at **about 7-8x on both views** of the Mandelbrot set (if you're around 7x 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.). In your writeup, describe your approach to parallelization and report the final 8-thread speedup obtained.
5. Now run your improved code with 16 threads. Is performance noticeably greater than when running with eight threads? Why or why not?