Nihar Gupte

CSE 3320 Fall 2019

1001556441

Evaluation Report for Fractal Assignment

Purpose of experiments and Experimental Setup

The purpose of the experiments was to make a multithreaded program to create Mandelbrot images so that execution times speed up.

For the experimentation I started with a command line input of ./mandel -x 0.287 -y 0.014 -s 0.0001 -W 500 -H 500 -m 1000 which took omega about 0.25s to run, and hence I changed the s value to 0.00005; that took me about 0.26s to execute. Then I furthermore reduced the value of s to 0.00001 which took omega about 0.39s. So, I went back to s being 0.00005 but inputted the height and width as 2000 each to get about 4.18s of execution time until arriving on my final command line input of

./mandel -x 0.287 -y 0.014 -s 0.00005 -W 2048 -H 2048 -m 1000

which took omega about **5.47s** to execute.

The above-mentioned values were all single threaded, i.e. n=1.

When I started testing my program with multiple threads, here are the execution times that I got upon executing the same command of ./mandel -x 0.287 -y 0.014 -s 0.00005 -W 2048 -H 2048 -m 1000 -n 'x' where 'x' is 2,10 and 50.

Number of threads	Execution Time in seconds
1	5.47s
2	2.31s
10	1.52s
50	1.17s

It was clear to me that the execution time decreased as I increased the number of threads and hence my multithreading program was ready to be tested with the two type of configurations.

<u>My experimental setup / run through of code</u>

When number of threads = 1, I used the given function on CSE 3320's GitHub repo, so I was able to test my code, as for n=1, threads were not created and main() did all the execution as usual, calling compute_image1 function with appropriate parameters.

For number of threads>1, I edited the code to implement threads. I used a struct which contains essential parameters such as the ymin, ymax, xmin, xmax, max and thread_id. I used a simple formula which determines which pixels of height, the thread needs to draw depending on its thread ID.
For example, first, the code calculates number of divisions, so if pixel height = 500 and n=3, then the number of divisions are 166. Hence thread ID 0 runs from 0 to 165, ID 1 runs from 166 to 331 and ID 3 runs from 332 to 499. If it so happens that the last thread ID runs for more pixels than the height of the bitmap, then I reset the upper limit for that thread ID as the pixel height. The rest of the code in compute_image function is as is from the GitHub repo, with minute changes to the for loop counter variables so that each threads executes its share of pixels from y1 (height1) to y2 (height2).

After the execution of the threading process, I join the threads back in a for loop and the bitmap is saved to memory.

<u>Threading library:</u>

I used the default pthreads library available in C language. The reason to use this library was primarily that the uploaded slide notes on Canvas and content in lectures covered pthreads library. The Github repo even had some multi-threading examples that I referred to while coding my program. This made pthreads the most obvious choice to use in my assignment.

<u>Implementation:</u>

I first defined pthread_t 'n' times for 'n' number of threads. Then inside a for loop running n times, I call pthread_create() to create and activate all threads so that they run the course of their business using function *compute_image. In another for loop after creation, I join the threads for synchronization.

<u>Data Evaluations:</u>

For the given two configurations in the assignment requirements, I measured and graphed the execution times of my multithreaded Mandelbrot.

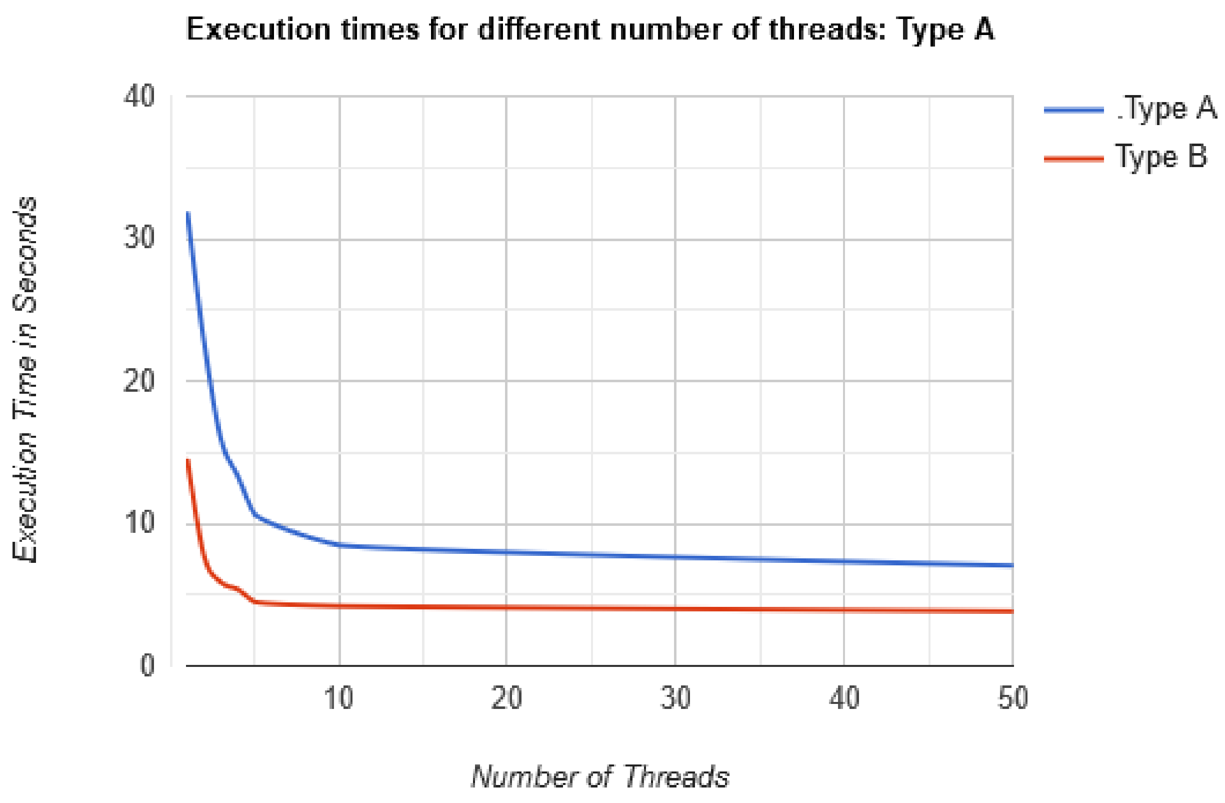
Here is the data that I obtained for configuration **type 'A' (./mandel -x -0.5 -y 0.5 -s 1 -W 2048 -H 2048 -m 2000**) :

Number of threads	Execution Time in seconds
1	31.828
2	22.703
3	15.930
4	13.326
5	10.660
10	8.485
50	7.04

Here is the data that I obtained for configuration **type 'B' (./mandel -x 0.2869325 -y 0.0142905 -s .000001 -W 2048 -H 2048 -m 1000**) :

Number of threads	Execution Time in seconds
1	14.490
2	7.698
3	5.870
4	5.380
5	4.500s
10	4.200s
50	3.845

Here is the graph that plots the **number of threads used against the execution time in seconds**. The Blue line represents type A configuration and Orange line represents type B configuration.



According to the graph, A and B have similar shapes because as you increase the number of threads, the execution time for the program decreases. As you increase the threads more towards 50, the execution time decreases, however it decreases slowly and steadily vs. compared to when you increase the threads from say, 1 to 5, the execution time decreases more rapidly. So, it is safe to assume that the execution time almost saturates when we use a greater number of threads (50).

It is difficult to pinpoint what is the optimal number of threads for both the types. However, solely based upon the execution time, I would conclude that n=50 is the optimal case in both types, A and B since n=50 results in the least execution time and therefore the fastest generation of Mandelbrot images.

Curves A and B have different start and end points, i.e. different execution times as for A, even though the scale is 1 (it is less zoomed in), m, which is the amount of work that has to be done in drawing every point = 2000 for A. Therefore, a greater number of iterations take place per pixel for every thread for curve A. Curve B in contrast, has m=1000, implying lesser iterations and therefore faster execution in general.