

Lebanese American University

Course: CSC447 Parallel Programming for Multicore and Cluster Systems

Doctor. Hamdan

Youssef Idress

202104162

Spring 2024

1. **Parallelizing Mandelbrot:**
2. **Introducing the problem:**

Mandelbrot is a way to present a bit-mapped image where it has to undergo a series of computations which are known to be extensive and heavy. The bits are represented by sets of points in a complex plane where they will fluctuate up and down (increase and decrease), yet, within a limit whenever it is computed within a function; the function is usually represented as: Zk+1 = Zk2 + C where the Z is a complex number (a + bi) and C is a complex number that represents the position on the complex plane. Zk+1 is the K+1 iteration of Z and Zk is the Kth iteration.

1. **Strategy taken when parallelizing the code:**

The following assignment will be tackling this problem and will be attempting to parallelize it in two ways, Statical Task Assignment and Dynamic Task Assignment. I will be first introducing the thought process behind the implementations and then explain each task.

1. **General Thought Process:**
   1. After running and understanding the serial code, I started debugging it entirely and understand which part of computationally intensive. After that I realized it was necessary to divide the array into chunks and distribute them to the available processors.
   2. Each chunk will represent a sub-plane and the assigned processors will individually compute them and then at the end all the sub-results will be combined to form the final image.
   3. It was necessary to assign a correct and suitable communication functions; the main tool/library that was used is MPI (Message Passing Interface) and the list of communication functions that it supports.
2. **First Parallel Technique: Static Task Assignment (STA):**

Static Task Assignment is a parallelizing technique that aims to statically assign tasks to the processes or thread that are used during the executions.

The way this strategy was implemented is that the array was divided into equal chunks and then they are distributed to the available processers. This process is fairly simple and effective, yet it is not the most effective when it comes to parallelizing techniques.

With the Mandelbrot set, it was not the most effective as it known that some parts of it is more computationally intensive than the others; meaning, that some processors will have to execute more steps and take longer to compute than others. This mismatch will cause some of the assigned processors to be idle as they finish their tasks faster compared to the rest.

1. **Second Parallel Technique: Dynamic Task Assignment (DTA):**

Dynamic Task Assignment, from its name, it aims to assign the divided tasks dynamically to the available processors and threads that are used during the execution of the program. This technique is known to be more efficient than STA as it balances the workload in a fairer manner and decreases the idle times of the processors significantly.

The way this technique was implemented is that the array was divided into smaller initial chunks (I could’ve sent one row at a time, but my technique was to decrease the communication to reduce the timing) and then the Master node sends the data to the available processors. Then once a processor become available it sends a request to the Master where it sends it another chunk until the array is done.

The implementation of DTA is more complex than STA as more communication was implemented and it was prone to serious problems such as deadlocks or infinite loops as the image might be done processing, but the communication is still going.

1. **The Setup and Environment used:**

The programming language that was mainly used was C; and all the code was written and tested via Google Collab as it provided the ability to test the programs written with more resources present on the laptop that I was working on. (The link of both GitHub and Google Collab are present in the below section).

The library used for parallelization, as mentioned previously, was MPI (Message Passing Interface) library and it provided a series of functionalities that aimed to parallelize the code. The main communication functions that were called are:

* MPI\_Init(to initialize MPI)
* MPI\_Comm\_rank(the communicator: MPI\_COMM\_WORLD)
* MPI\_Comm\_size(to get the number of processes)
* MPI\_send(to send data from one process to the other)
* MPI\_recv(to receive data from one process to another)
* MPI\_Gather(to gather all the results that each processor computed into one)
* MPI\_Wtime(used to calculate the time needed by the program and each processor)

As mentioned previously, the software used was Google Collab and we do not have direct access to the resources available, so whenever we execute the program, we specify the number of processors that we needed.

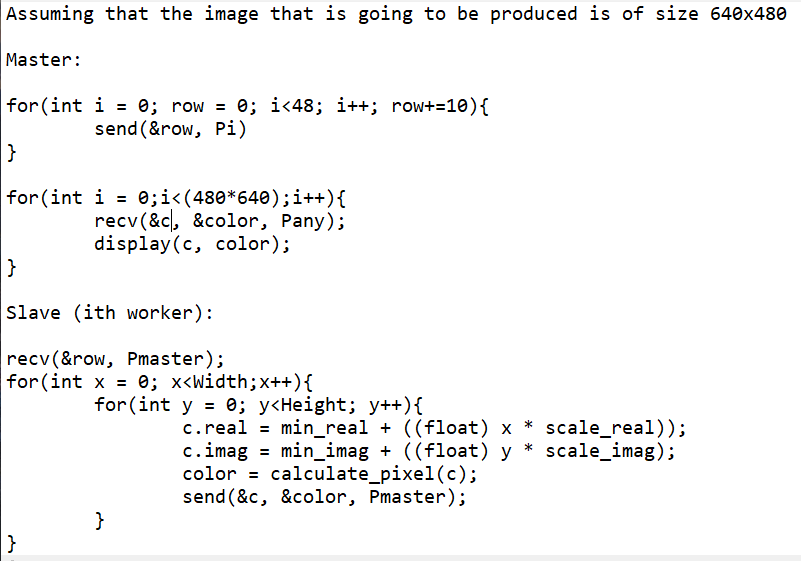
Yet, from this is the name of the Machine that was used and the information that was given to use by Google Collab:

* Name: Python 3 Google Compute Engine Backend
* Resources:
  + System Ram:12.7 GB
  + System Storage: 107.7 GB
  + As mentioned previously, throughout the execution we specify the number of processors needed.

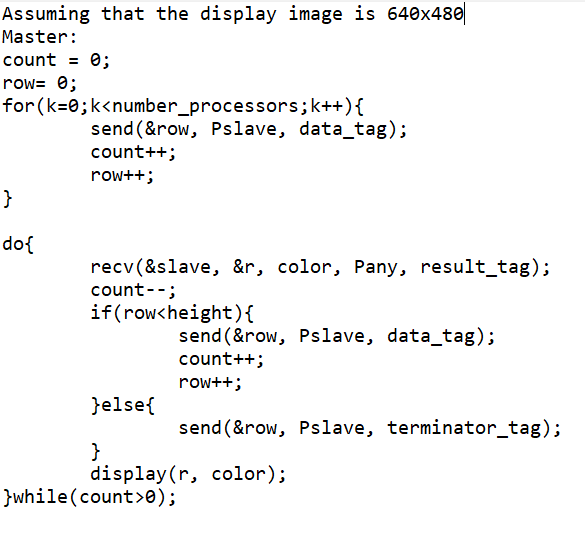
1. **Pseudo Codes for each technique:**

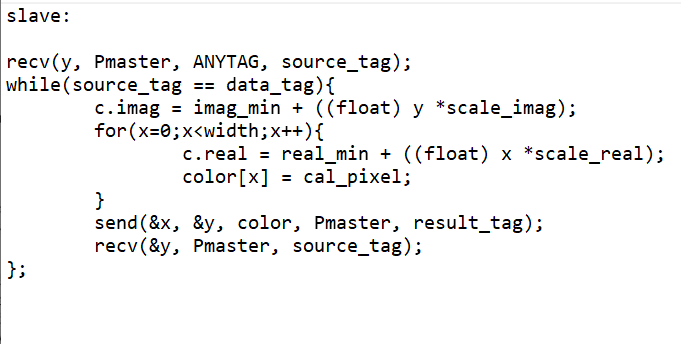
These pseudo codes are mainly gotten and followed from the book that was given to us in class:

**Static:**



**Dynamic:**

****



The GitHub Link:

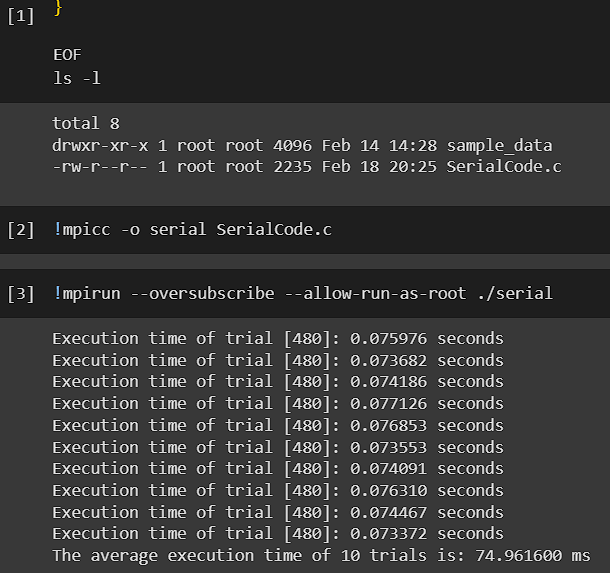
<https://github.com/youssef-idress/Parallel-MandelBrot.git>

The Google Collab Link:

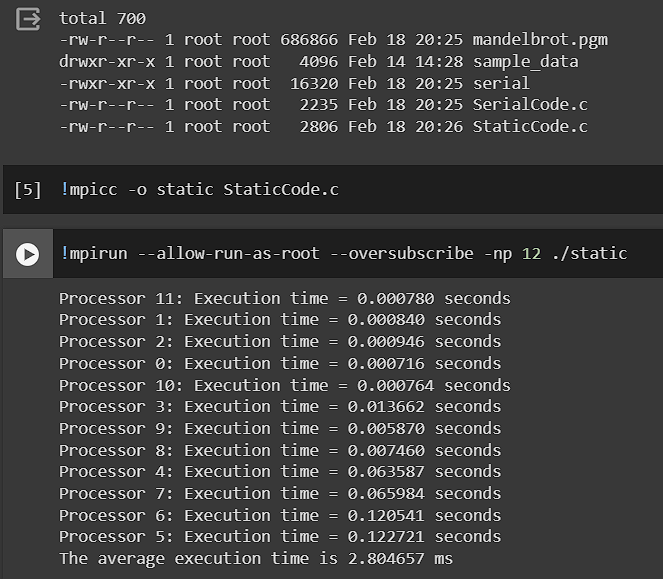
[Parallel Assignment1.ipynb - Colaboratory (google.com)](https://colab.research.google.com/drive/1iEl_I5rdsuf1fDCykwSmjUz37JfFdm_F?authuser=0#scrollTo=mFh9FpVj5yy9)s

1. **Proof of execution, Results:**

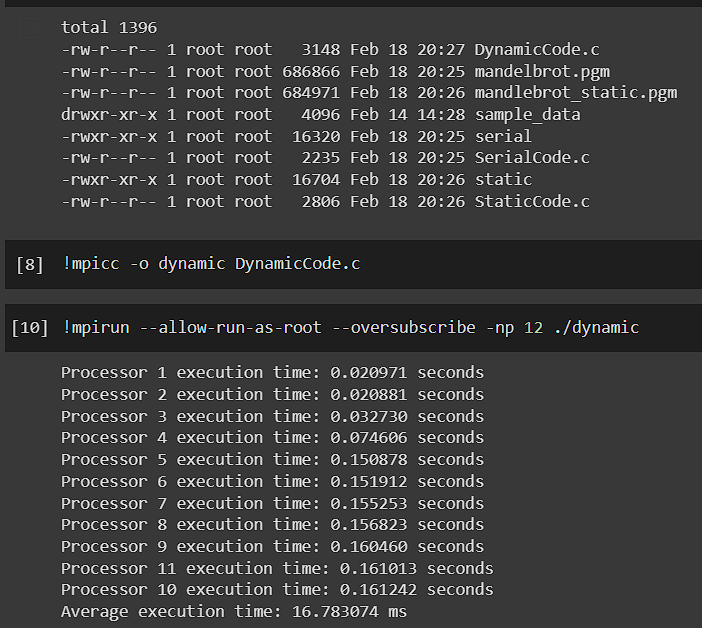
For the proof of serial execution, this is the result after running the code:



For the proof of the Static Execution, this is the result after running the code:



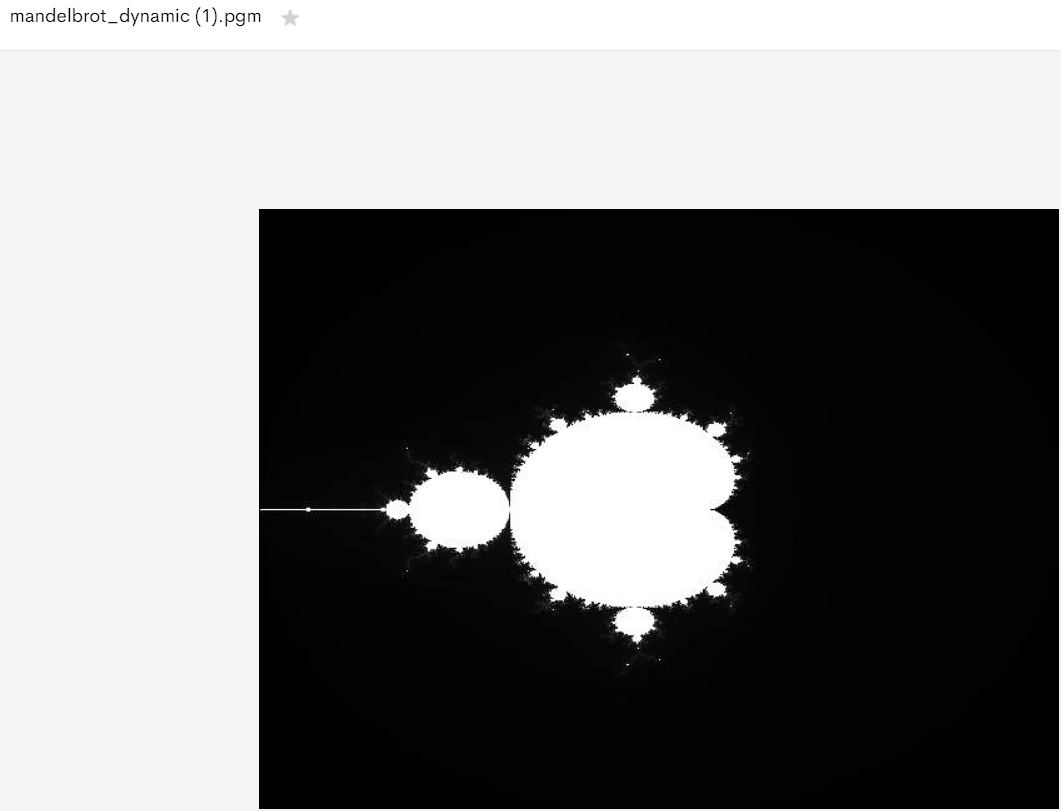
For the Dynamic Execution, this is the result after running the code:



These are the respective images of each after execution (Serial, Static, Dynamic):







1. **Performance Analysis:**

Sequential:

According to the results from running the code, I got: 74.961600 ms.

The speedup factor depending on the number of processes: (static)

S(2) = 0.0749616/0.042194551 = 1.77

S(8) = 0.0749616/ 0.048298487 = 1.55

S(12) = 0.0749616/0.03297 = 2.27

S(24) = 0.0749616/0.02919 = 2.56

As shown, as we increase the number of processors, we get a higher speedup factor with the static implementation.

The speedup factor depending on the number of processes: (dynamic)

S(12) = 0.0749616/0.01678 = 4.46

S(24) = 0.0749616/ 0.00997 = 7.51

Efficiency

Efficiency(E) = Execution time using one processor/ Execution time using a multiprocessor x number of processors

Static:

E(2) = S(2)/2 = (1.77/2) \* 100 = 0.885 \* 100 = 88.5

E(8) = S(8)/8 = (1.55/8) \* 100 = 19.37

E(12) = S(12)/12 = (2.27/12)\*100 = 18.91

E(24) = S(24)/24 = (2.56/24) \*100 = 10.6

Dynamic:

E(12) = (4.46/12) \*100 = 37.1

E(24) = (7.51/24)\*100 = 31.29

Computation Ratio:

Static:

Processor number: 8

Total Speedup in seconds = 0.045838

Computation time without communication = 0.042012 seconds

Ratio = 0.916

Dynamic:

Processor number: 8

Ratio: 0.041

Scalability:

As the number of processors increase in both cases, the efficiency and scalability decrease. However, there is a sharper decrease in the dynamic approach whereas in the static there is more of smoother downward slope (there was an error in the graphical interpretation, the above analysis is based on the numbers, due to the lack of time unfortunately I was not able to fix it)

Conclusion:

As proven, both approaches are a great way to parallelize this problem. However, there is a drawback in both. The static’s drawback is that some processors will be idle, and, in the dynamic, there is more of a communication overhead.

Choosing the implementation depends solely on the requirements needed and the usage. If you are aiming for a decent improvement with a simple implementation where there aren’t much of risks when it comes to communication, it is best to go for the Static Approach. Yet, if you want to go for the more efficient and faster approach, the Dynamic Approach is the one you should follow, yet, this approach has the drawbacks in the sense it is harder to implement but the results might be worth it.

Also, it is shown that the efficiency is higher with small processors, this is due to the fact that no matter how much we increase the number of processors, it will always reach a plateau where it won’t be beneficial anymore if we parallelize it. This was proven with the efficiency numbers that are stated above; another reason for such a drawback is due to the communication overhead that occurs when we increase the number of processors.