

Lebanese American University

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

CSC447

Parallel Programming / Multicore Architecture & Cluster Programming

Dr. Hamdan Abdellatef

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

ASSIGNMENT 01

Spring 2023

Sunday, 26th February, 2023

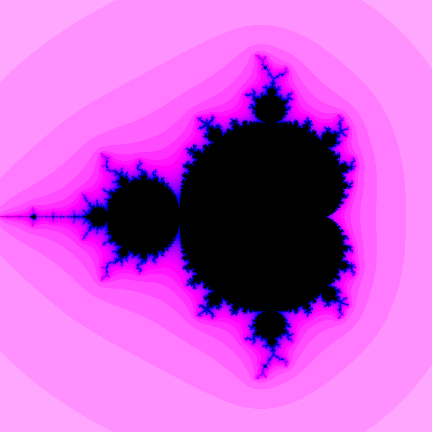
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Jalal El Zein

202105966

[jalal.elzein@lau.edu](mailto:jalal.elzein@lau.edu)

***I – Parallelizing the Computation***

1. ***Introduction to the Problem***

A Mandelbrot Set is the set of complex numbers *c* for which the function does not diverge to infinity when iterated from .

For our purposes, we can think of the Mandelbrot set as a fractal set that can be visualized by generating a complex plane of points, then iterating a complex function over each point to determine whether or not it belongs in the set.

Figure 1. Example of a Mandelbrot Set

This is obviously very computationally expensive, which is why we need to parallelize such a program to yield results within reasonable constraints like time and hardware resources.

In this assignment, we will see the implementation of this task with two methods of parallelization; Static Task Assignment and Dynamic Task Assignment.

1. ***Parallelization Strategy***
   1. *General Strategy*

In general, we will follow five main steps:

* + 1. I will try to divide the plane into smaller parts that can be processed independently of one another.
    2. Each sub-plane would be processed on a separate node, whether that be a process, thread, or different machine.
    3. Define an appropriate function or algorithm to calculate the Mandelbrot set.
    4. We would use a parallelization framework like OpenMP or MPICH to write a parallel algorithm to distribute the work among our computational nodes.
    5. At the end, we can “collect” and combine the results from all the computation nodes to create the desired image.
  1. *Static Task Assignment (STA) Strategy*

STA is a parallelization method in which tasks are statically assigned to threads/processes/machines at the beginning of the program execution.

To implement STA in this project, we will specify further the behavior our program will follow in the fourth step; distributing the work among the computational nodes. Namely, our implementation will divide the computation of the set function into equal-sized tasks for the nodes.

STA is a very simple and effective way to parallelize tasks like the generation of a Mandelbrot set, however, it may not be the best in situations where some tasks take longer than others to complete. In our example, we know that, naturally, some parts of a Mandelbrot image will be brighter than others, so we have a varying number of computations between different sub-planes or sub-images. These will lead some nodes be in idle while waiting for others to complete their assigned tasks.

* 1. *Dynamic Task Assignment (DTA) Strategy*

DTA is a parallelization method in which tasks are dynamically assigned to processes/threads/machines during the program execution. This method is usually more efficient than STA because it is able to balance the workload fairly among the nodes of computation and reduce idle time.

To implement DTA here, we will need to create a dynamic task scheduler. This scheduler will be responsible for assigning tasks to nodes once they become available while taking into account the size of the workload.

Implementing DTA is a lot more complex than STA, largely thanks to the task scheduler. That said, the tradeoff of achieving better load balancing and lower idle time among nodes is worth the trouble!

***II – The Setup and Environment***

1. ***Programming Language***

Out of comfort and experience, I chose C++ as the language to implement my solution with.

1. ***Software, Libraries, Frameworks, and Packages***

I used MPICH as a parallel processing library and implementation of MPI.

I also used Oracle VirtualBox for virtualization.

1. ***Specifications of the Computational Nodes***

All nodes were run as virtual machines with Oracle VirtualBox.

|  |  |
| --- | --- |
| CPU Cores | 2 |
| RAM | 2048 MB |
| Virtual Disk Space | 10 GB |
| Operating System | Ubuntu Server 22.04.2 (x64) |
| EFI Enabled | False |

1. ***Network Settings***

Master Node - Adapter 1

|  |  |
| --- | --- |
| Type | Bridged |
| Name | Realtek 8822CE Wireless LAN 802.11ac PCI-E NIC |

Master Node – Adapter 2

|  |  |
| --- | --- |
| Type | Internal Network |
| Name | intnet |

Slave Node – Adapter 1

|  |  |
| --- | --- |
| Type | Bridged |
| Name | Realtek 8822CE Wireless LAN 802.11ac PCI-E NIC |

Slave Node – Adapter 2

|  |  |
| --- | --- |
| Type | Internal Network |
| Name | intnet |

***III – Source Code***

Source code can be found on my GitHub repository: [Link](https://github.com/jalal-elzein/Mandlebrot-Set-Parallelization)

***IV – Proof of Usability***

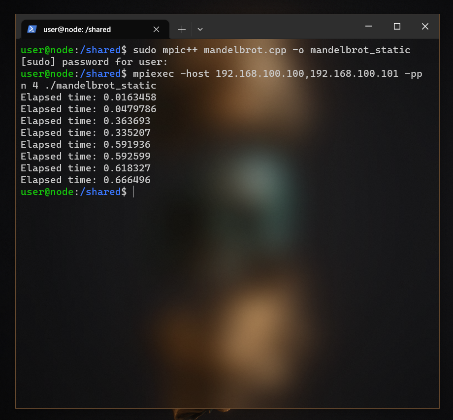
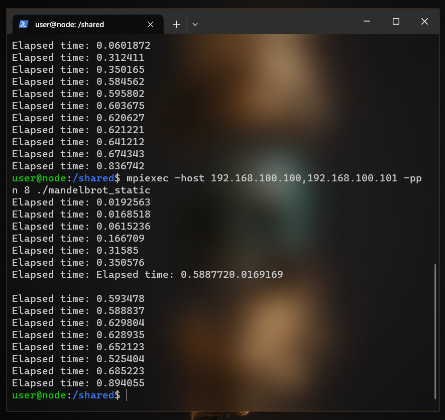
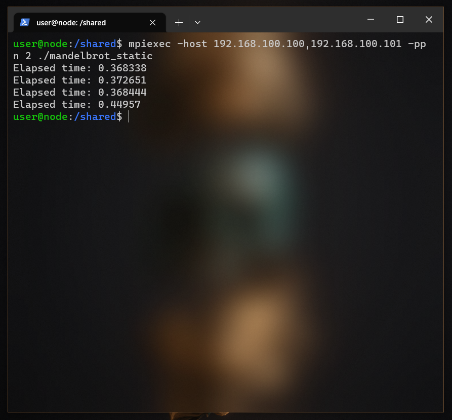
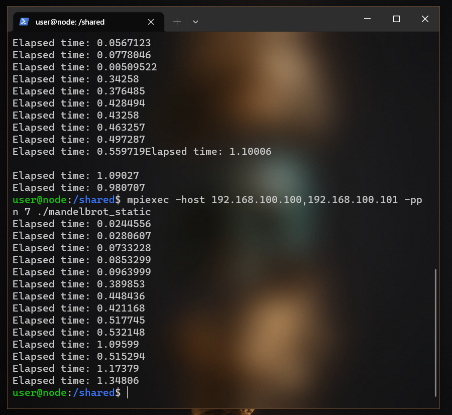
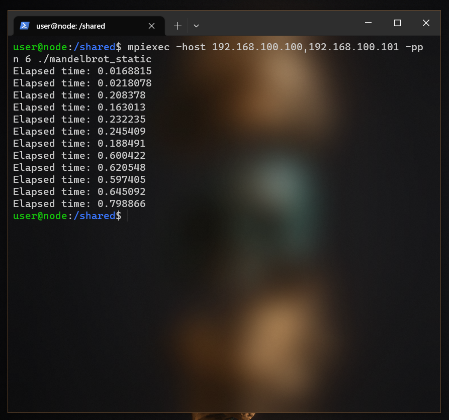
 Code was compiled and executed using the MPICH compiler. Figure 2 provides a compilation of images describing some examples of running the code. More details and screenshots of the setup and running process can be provided upon request.

Figure 2. Executing the program

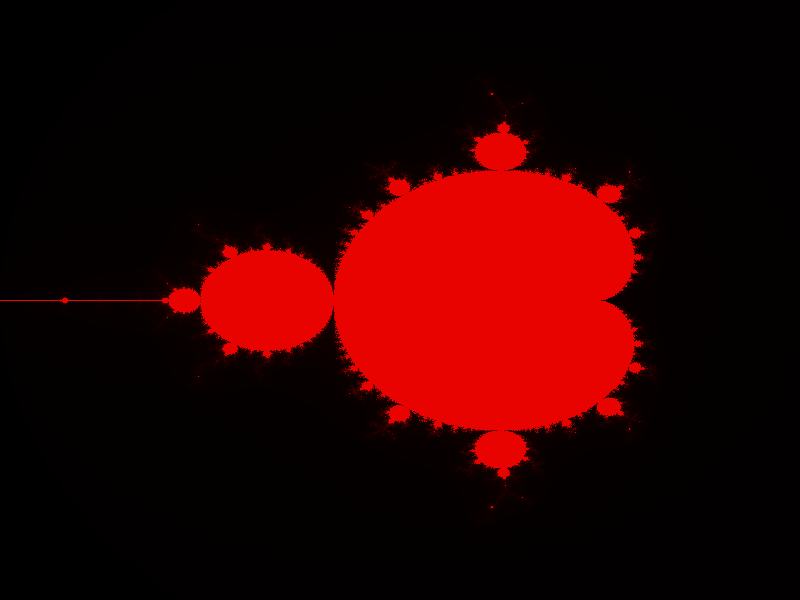


Figure 3. Resulting image from the program

***V – Performance Evaluation***

Unfortunately, I was unsuccessful at developing a working model with dynamic task assignment. So the evaluation of this parallelization will be limited to the scope of STA and the effect of computational nodes on performance.

1. ***Recording the Data***

The program was run five times for each “level”. “Level” here referring to the number of processes per node. Data was recorded for Levels one through eight. Since my setup involved only two nodes, we can conclude the range of nodes was two to sixteen, with counting-steps of two. Figure 4 displays a table showing the recorded data.



Figure 4. Recorded Data

1. ***Visualizing the Data***

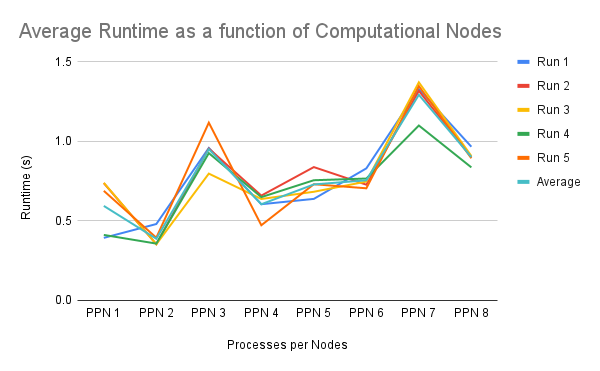


Figure 5. Runtimes of Different PPN

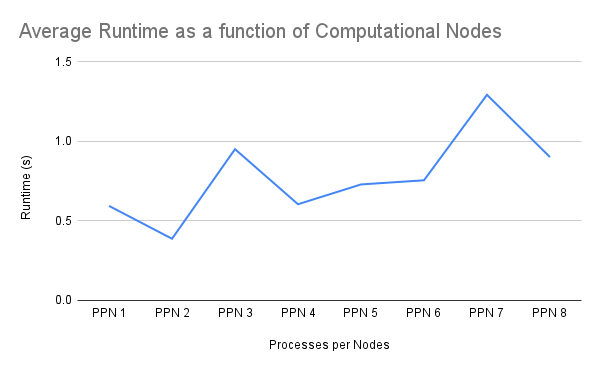


Figure 6. Average Runtime of Different PPN

1. ***Speed Up Factor***
   1. *Control Experiment*

Of course, to evaluate the speedup and efficiency of this parallelization, we first need to create a completely serial program that achieves the same purpose. So, I also wrote a program to generate an image of the same size (800×600 px) with C++ as well.

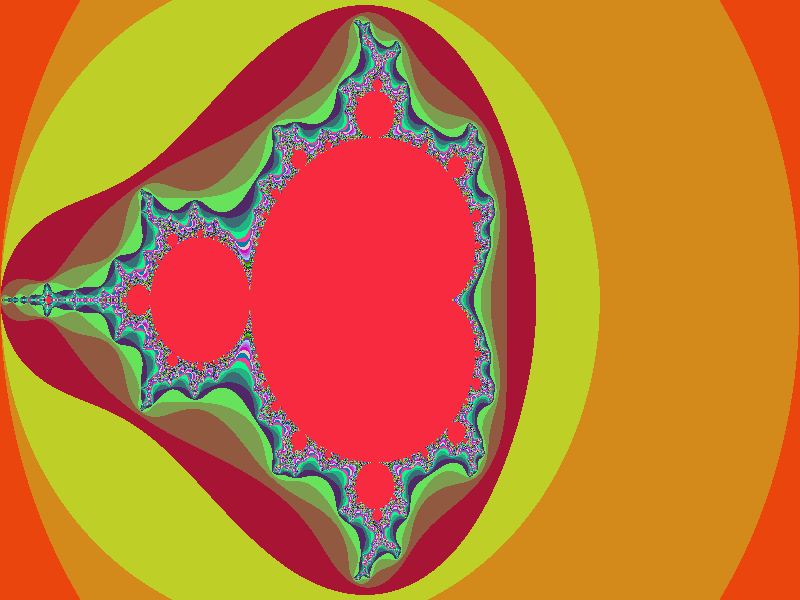
The only difference in the setup for the serial program is that it complies with C++20 standards, because I am more comfortable writing them, while the parallel implementation complies with C++11 standards because that is what’s supported with MPICH. In this problem, the only difference the updated standards affect is the declaration of the constants at the beginning of the program. While List Initialization is safer to use and considered best-practice with newer versions of C++, the performance increase in our case is negligible, and this was tested and compared with the C++20 implementation to verify that the average over 5 runs is basically identical.

The data recorded by the serial implementation can be seen in Figure 7 below, and we can confirm our hypothesis from the introduction; this is indeed a very computationally expensive task.

The resulting image from the serial implementation is described in Figure 8.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average |
| Serial | 10.886 | 10.61 | 10.652 | 11.385 | 10.852 | 10.877 |

Figure 7. Runtimes of the Serial Implementation

**

* 1. *Calculation*

The Speedup Factor is defined by , where is the serial runtime of the program. And is the parallelized runtime with processors. Therefore, we can calculate the Speedup Factor for the parallel implementation for different PPNs. The results of this calculation are displayed in Figure 9, then visualized in Figure 10.

Figure 8. Resulting Image of Serial Implementation

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 PPN | 2 PPN | 3 PPN | 4 PPN | 5 PPN | 6 PPN | 7 PPN | 8 PPN |
| *S(p)* | 18.29709353 | 28.03415543 | 11.4342601 | 17.97720496 | 14.91008984 | 14.39533505 | 8.410983691 | 12.06441602 |

Figure 9. Recorded Runtimes for the Serial Implementation

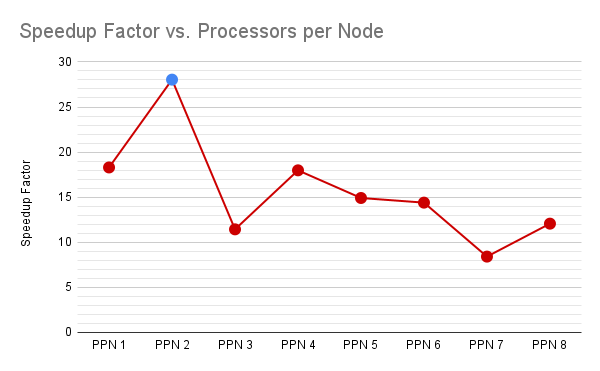


Figure 10. Line Graph Showing the Speedup Factor in relation with the number of Processors per Node

1. *Efficiency*

Efficiency is concerned with studying the relation between the number of processors and the speedup achieved. In layman terms, we aim to answer the question: “Was adding these processors worth it?”.

Below is the equation to calculating Efficiency:

 The efficiency for each PPN is calculated according to the above formula. The results are shown in Figure 11 and graphed in Figure 12.

Figure 11

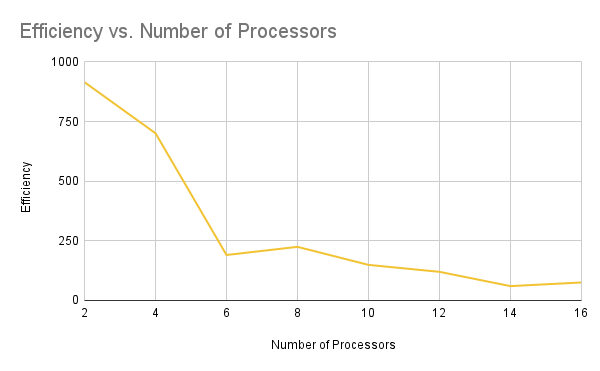


Figure 12

***VI – Discussion***

1. ***Conclusion***

By examining the graphs for the average runtime of the parallel implantation displayed in Figure 6, we can see the general trend in the graph follows a positive slope; increasing the runtime of the program when we increase the number of processors. The initial hypothesis to this phenomenon would probably be that my code simply is too inefficient at handling the communication between the different processes, and the overhead created by the Message Passing Interface is stacking as we increase the number of processes running our program.

We can notice results that agree with the aforementioned hypothesis in Figure 10, describing the speedup factor. We notice a peak in speedup at 2 Processors Per Node, i.e. at 4 processors (marked in blue). After that point, we see a clear and sudden dive in speedup that slightly recovers at 4 PPN only to fall back again.

Finally, Figure 12 ties the whole story together. We observe that the peak of efficiency was at 2 processor, this value decreases at 4 processors, only to take a dramatic dive at 6 processors and continues to decline steadily after that.

We can conclude by looking at the bigger picture, that our code takes very good advantage of parallelization with four processors. We can explain this range by the following hypothesis;

At 2 processors, our program runs the most efficiently in regards to minimizing overhead, with the tradeoff of having too few processors to take advantage of parallelization to the maximum. At 6 or more processors, the overhead created by communication and combination of results far outweighs the advantages of parallelization.

This leaves us with the “sweet spot” of 4 processors, which takes the most advantage of parallelization without sacrificing too much runtime to communication and combination overhead.

1. ***Final thoughts and Future Studies***

There is definitely more to be said about the parallelization of the Mandelbrot Set problem. Mainly examining the effect of DTA. In addition to that, there is more to be studied when it comes to load balancing, which was not done in this paper because this is more noticeable with DTA than STA.

It is disappointing to find that my code is the direct cause of the “failures” of parallelization on this project but I man enough to admit I am not surprised. This is quite literally my first implementation of a parallelized algorithm, so I don’t hold myself to golden standards just yet. I just hope I can look back at this in a few years’ time and laugh about how bad my code was, knowing I could now do better.