# Efficient Generation of Mandelbrot Set using Message Passing Interface

28993373
Samarasekara Vitharana Gamage, Bhanuka Manesha
*School of Information Technology*
*Monash University*
Malaysia
bsam0002@student.monash.edu

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

***Index Terms*—component, formatting, style, styling, insert**

## I. Introduction

The Mandelbrot set, named after Benoit Mandelbrot is the set of points c in the complex plane which produces a bounded sequence with the application of equation 1 repeatedly to the point $z = 0$ [1].

$$f(z) = z^2 + c \tag{1}$$

So in order to generate the Mandelbrot set for a $m \times n$ image $I$, we need to execute equation 1 on each pixel. So we derive the following equation.

$$M_{i,j} = z_{i,j}^2 + c_{i,j} \tag{2}$$

$$\text{where } i = 1...m, j = 1...n$$

This report aims to provide an efficient partition scheme to generate the Mandelbrot Set. For this, three partition schemes are compared and contrasted against each other and then ranked based on their performance. First Bernstein's conditions are used to determine whether the Mandelbrot Set is data parallelizable [2]. Then the theoretical speed up is calculated using Amdhal's law [], which is used as the target. The percentage difference of each of the partition schemes against the theoretical limit is then obtained and used for the comparison. The Theoretical Row Based Partition Scheme is used as the base line and

## II. Preliminary Analysis

### A. Parallelizability of the Generation of Mandelbrot Set

In order to ensure that the process of generating the Mandelbrot Set is data parallalizable, we can use Bernstein's conditions [2]. Bernstein's condition is a simple verification for deciding if operations and statements can work simultaneously without changing the program output and allow for data parallelism [3]. According to Bernstein, if two processes satisfy the following equations, then they are parallalizable.

$$I_1 \cap O_2 = \emptyset \tag{3}$$

$$I_2 \cap O_1 = \emptyset \tag{4}$$

$$O_1 \cap O_2 = \emptyset \tag{5}$$

$I_0$ and $I_1$ represents the inputs for first and second process while $O_0$ and $O_1$ represents the outputs from first and second process. Equation 3 also known as anti in-dependency, states that the input of the first process should not have any dependency with the output of the second process, while equation 4, also known as flow in-dependency, states that the input of the second process should not have any dependency with the output of the first process. And finally equation 5 also known as output in-dependency, states that both the outputs of the two processes should not be equal.

Using equation 2 we can derive the input and output equations for any two neighboring pixels which will be computed by two processes $P_1$ and $P_2$.

$$I_1 = z_{i,j}^2 + c_{i,j} \tag{6}$$

$$O_1 = I_{i,j} \tag{7}$$

$$I_2 = z_{i,j+1}^2 + c_{i,j+1} \tag{8}$$

$$O_2 = I_{i,j+1} \tag{9}$$

Applying Bernstein conditions for equations 6, 8, 7 and 9, we get,

$$z_{i,j}^2 + c_{i,j} \quad \cap \quad I_{i,j+1} = \emptyset \tag{10}$$

$$z_{i,j+1}^2 + c_{i,j+1} \quad \cap \quad I_{i,j} = \emptyset \tag{11}$$

$$I_{i,j} \quad \cap \quad I_{i,j+1} = \emptyset \tag{12}$$

Since all the above conditions are satisfied, we can state that the generation of Mandelbrot Set satisfies the Bernstein conditions, thereby the data can be parallelized and computed concurrently.

Next we calculate the theoretical speed up to determine whether there is a benefit of running the code in parallel.

### B. Theoretical Speed Up of the Generation of Mandelbrot Set

To determine the theoretical speed up of the Mandelbrot Set Generation, we first determine the pararalizable portion of the serial implementation of the code. This can be achieved by calculating the total time for generating the Mandelbrot set ($t_p$) and the total time to execute the whole serial code ($t_t$). Then using equation 13 we can get the $r_p$ value.

$$r_p = \frac{\text{time taken for parallelizable portion of the code}}{\text{total time for serial execution}} \quad (13)$$

Then after calculating the $r_p$ value we can use Amdahl's Law () to calculate the theoretical speed up value to generate the Mandelbrot set. Amdah stated that if $p$ is the number of tasks, $r_s$ is the time spent on the serial part of the code and $r_p$ is the amount of time spent on the part of the program that can be done in parallel, () then the speedup can be stated as

$$S_p = \frac{1}{r_s + \frac{r_p}{p}} \quad (14)$$

So using Amdah's law, we are able to generate the following chart with the theoretical speed up values.

TABLE I: Number of tasks vs Speed up factor

| Number of Tasks | Speed Up Factor |
|---|---|
| 2 | 1.99440 |
| 4 | 3.96659 |
| 8 | 7.84583 |
| 16 | 15.35350 |
| 32 | 29.43820 |
| $\infty$ | 356.22764 |

More details in the Appendix - Table ???.

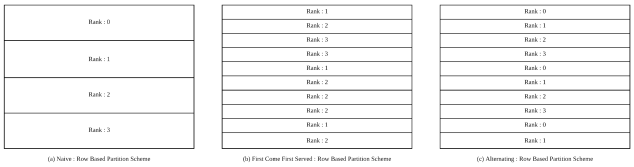
So having determined that the generation of Mandelbrot Set is embarrassingly parallelizable and calculated the theoretical speed up, next we look into the design of the partition scheme for parallelizing data.

## III. Design of Partition Schemes

For the design of the partition schemes, we will be comparing three partition schemes. In all of these schemes, we will be writing the Mandelbrot Set to the file in only the master node. So each approach will send the data back to the master node, therefore all of them will be using a Master-Slave Architecture. Therefore, each partition scheme will be under a controlled environment, thus allowing us to compare the actual Mandelbrot Generation speed up.

### A. Naive : Row Based Partition Scheme

As shown in ¡enter¿, the final image is divided equally among N number of processors. So each process will pre-calculate its start and end positions and generate the Mandelbrot Set for that portion. The equation used to calculate the start ($S_r$) and end point ($E_r$) for each processor with rank ($r$) is shown below.




(a) Naive : Row Based Partition Scheme (b) First Come First Served : Row Based Partition Scheme (c) Alternating : Row Based Partition Scheme

Fig. 1: Partition Schemes when $N = 4$

$$S_r = \frac{iYmax}{\text{number of tasks}} \times \text{rank} \quad (15)$$

$$E_r = iYmax\%\text{number of tasks} \quad (16)$$

Using this partition scheme each process get an equal amount of rows to work with, which can be written using the following equation.

$$\text{rows per task} = \frac{iYmax}{\text{numer of tasks}} \quad (17)$$

In the case where the number of rows cannot be equally divided among the number of tasks, the last node (rank = $N$ - 1) will calculate the remaining set of rows.
After generating the $S_r$ and $E_r$ points, all of the nodes will generate the $Cy$ and $Cx$ arrays. Then the master node will open the file and it will start generating the Mandelbrot Set for the rows allocated for it. The other nodes will also use the $S_r$ and $E_r$ points to generate their specific rows. Finally, all the slave nodes will send their generated values to the master node. The master node will then receive the chunks of rows and write it to the file. The technical flowchart for the Naïve row based partition scheme is shown in Figure 2.

### B. First Come First Served : Row Based Partition Scheme

This partition scheme shares similarity with Schedulers, where the master node sends work based on the availability.As shown in Fig ¡enter¿ the order in, which node performs which row is based on the availability of the node. If one node finishes the work allocated to it, then it will proceed to the next row. So the master node keeps track of the rows that are processed so far and it will send the next row to the next available task. The slave node will then receive the row number and calculate the Mandelbrot set for that row. This will be done until the master node sends a row number larger than $iYmax$. At that point the node has finished all of the work so it will stop and exit the program. The master node will copy each row received from the slaves and use the row number to decide which row it is, and add it to the memory. Then after receiving all the rows, it will write the data from memory to file. For this partition scheme, the issue of having an odd core count or row values is not present since its based on the availability. Due to the nature of work allocation, this partition scheme is named as First Come First Served. The main this to note is that the master node does not perform in generation. It will only be delegating work to the slave nodes. The technical flowchart for the First Come First Served row based partition scheme is shown in Figure 3.

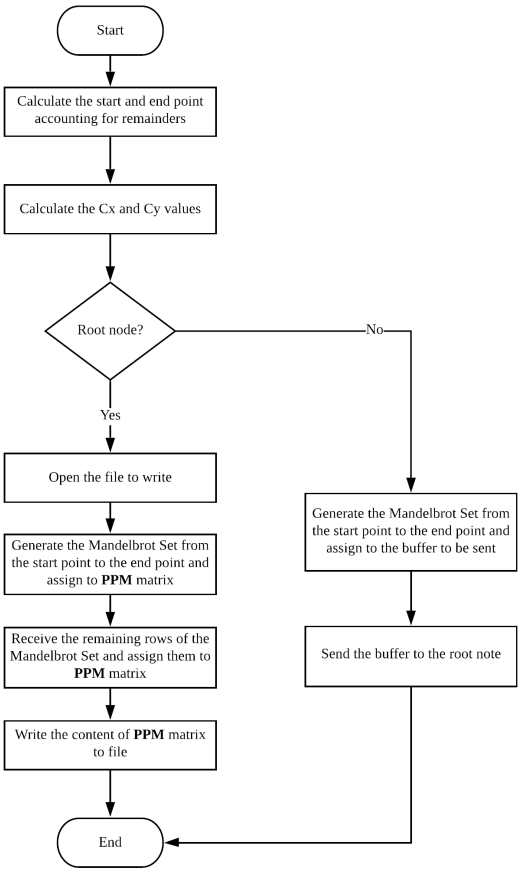



Fig. 2: Technical Flowchart for Naïve Row Based Partition Scheme

### C. Alternating : Row Based Partition Scheme

This partition scheme is similar to III-A, where all the tasks divide the work among themselves initially.Figure ¡¿ shows that each task will perform the alternating rows, thus dividing the number of rows equally. In the case where the number of rows cannot be equally divided, the master node will then perform the remainder. Initially all the nodes calculate the $Cy$ and $Cx$ values. The master node then opens the file and start calculating the rows starting from the $0^{th}$ index until the end, with increments of the number of tasks. Similarly, each node starts from their respective rank and increment with the number of tasks. Similar to III-A, each process get an equal amount of rows to work with, which can be written using equation 17. Then all the slave nodes sends back the generated rows to the master node. Before receiving, if there is a remainder, the master node will increment one by one to calculate it. After receiving all the rows, the master node will loop though the $0^{th}$ index till $iYmax$ and reconstruct the image collecting alternate rows from the received arrays. Then it will write all the data into the file. The technical flowchart for the Alternative row based partition scheme is shown in Figure 4.

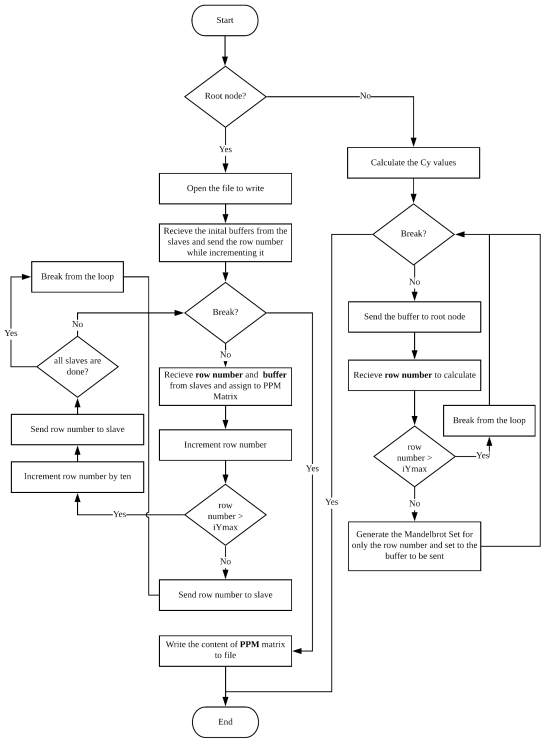
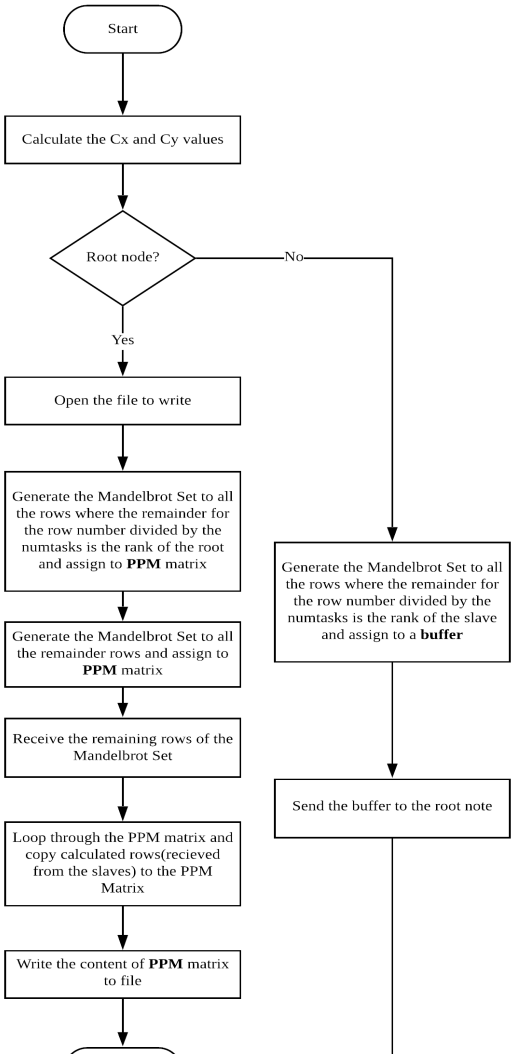



Fig. 3: Technical Flowchart for First Come First Served Row Based Partition Scheme



## IV. Test Methodology

In order to calculate the run times for the three partition schemes multiple tests were done. For each tests we set the size of the image to 8000 $\times$ 8000 with 2000 iterations. An assumption taken for the test cases was that the core count will always be a multiple of 2, but all of these partition schemes will work with any other core count, as all of them have a method to handle remainders. Each partition scheme was executed 5 times, with 2, 4, 8, 16 and 32 cores.

All of the tests were done on the MonARCH (Monash Advanced Research Computing Hybrid) HPC/HTC cluster. The problem with using a cluster is that the scheduler decide which CPU to use based on some predefined rules such as availability. So for our tests to be equal, a constraint was added in order to limit the tests to only one CPU type. All the tests were done on Intel® Xeon® Gold 6150 Processors with 36 logical cores. But since hyper threading was turned off, each processor had only 18 cores. So in order to run the 32 task process, two 16 core processors were used. Listing 2 of the appendix, shows the job file used to run the parallel code with 32 cores. The RAM was set to 32GB for all the tests.

For each test, we calculated the time taken for generating the Mandelbrot Set and also the total time taken including the writing of data to the file. All the results and the server output files are attached in the appendix.

When generating the Mandelbrot Set, the main property to note is the time taken to process each section of the image. The top and bottom parts of the image takes less time compared to the middle portion of the image. So each of the partition schemes will require to balance the work load not only by the amount of rows, but also based on the partition scheme that makes use of all the tasks equally.

## V. Results and Discussions

Now let us discuss about the results obtained from the testing phase. Each partition scheme will be compared and discussed based on the following criteria.

- Theoretical Speed-up vs Actual Speed-up
- Actual Speed-up between each partition scheme

### A. Theoretical Speed-up vs Actual Speed-up

In this section we will compare how the actual speed up compared against the theoretical speed up established in Section II-B

*1) Naive - Row Based Partition Scheme :* This method divides the whole image into $N$ number of parts, where $N$ is the number of tasks. Referring to Table IIa, for 2 cores, the theoretical speed and the actual speed-up if almost equal. The main reason for this is when we are using two cores, the workload is divided equally among the two processes. Both the tasks will be doing the high iteration and low iteration part of the image. But when the number of cores increase, the actual speed up becomes poor. The reason for this, even though the tasks get equal rows, is the number of iterations needed for some rows are greater than others, as mentioned previously in Section IV. Due to this, the tasks in-charge of the top and bottom quarter of the image takes less time compared to the middle part, so the master node has to wait until all the tasks are complete to write the file. Finally comparing the percentage difference IIb we see that this partition scheme does not perform well when the core count increases.

*2) First Come First Served - Row Based Partition Scheme:* This method sends the row based on the availability of the task. Looking back at Table IIa, we see that the performance of the 2 core is almost equal to no speed up at all. The main reason for this is when we use a two core the slave generates all the rows in the image, while the master sends and receives data. So the actual speedup should be $\leq 1$, but our actual speedup $> 1$. Why?. According to a paper published in 2016, by a group of researchers [4], this behavior is common in parallel systems as the serial program needs more RAM to store the data while generating the image, the parallelized system can use the cache as it can store the small chunks of data received at each send and receive. Therefore the speedup can be $\geq 1$ even with a lot of communication. Next, comparing the theoretical speed up vs actual speed up for multiple cores, we see that this partition scheme is sub-linear. The percentage difference is at a reasonable difference, but when two cores are used, it does not perform well because of the reasons mentioned above. But the percentage difference decreases with each increment of the core count.

*3) Alternating - Row Based Partition Scheme:* This method divides the rows based on the rank number of each task. Referring to Table IIa, this approach is almost equal to the theoretical speedup when two cores are used. The reason is the same as V-A1, where the work load is divided equally among the two tasks. Even when the number of cores increases, this method is able to perform in a sub-linear order staying close to the theoretical limit. Referring at Table IIb, we see that the the percentage difference increases with each increment in the number of cores.

Now that we have an idea about the difference between the Theoretical Speed-up and Actual Speed-up , let us look into how they compare up against each other.

### B. Actual Speed-up between each partition scheme

*1) Naïve - Row Based Partition Scheme :* Comparing the Naïve approach with the rest, we immediately notice the performance is poor. But evaluating dual core performance, this approach performs better than the First Come First Served approach. But as the core count increases, this approach cannot keep up with the other two partition schemes. Therefore we can conclude that this partition scheme performs the worst out of the three approaches.

*2) First Come First Served - Row Based Partition Scheme:* Comparing this approach with the rest, we see that it performs average, but as the core counts increases, we see that this performs better than any of the other methods. The reason for this is, both the other methods cause a bottle neck when they send the data at the same time to the master. If the work load is divided equally, then all the slaves finishes at the same time thus causing a bottle neck at the master node.

TABLE II: Analysis of Theoretical Speedup vs Actual Speedup

(a) Theoretical Speedup vs Actual Speedup of all the partition schemes

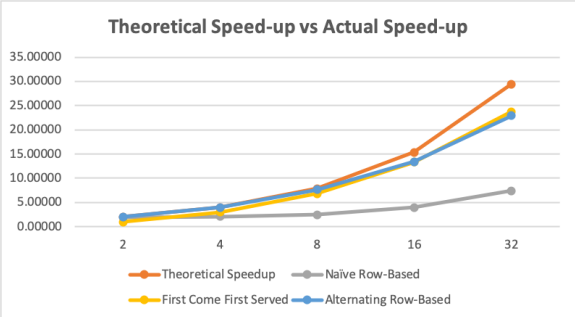
| N | Theoretical Speedup | Naïve Row-Based | First Come First Served | Alternating Row-Based |
|---|---|---|---|---|
| 2 | 1.99440 | 1.97812 | 1.00396 | 1.99000 |
| 4 | 3.96659 | 2.05984 | 2.98414 | 3.96060 |
| 8 | 7.84583 | 2.49809 | 6.77486 | 7.60671 |
| 16 | 15.35350 | 3.97305 | 13.33375 | 13.37449 |
| 32 | 29.43820 | 7.37966 | 23.72932 | 22.87749 |
| $\infty$ | 356.22764 | - | - | - |

(b) Percentage Difference of all the partition schemes

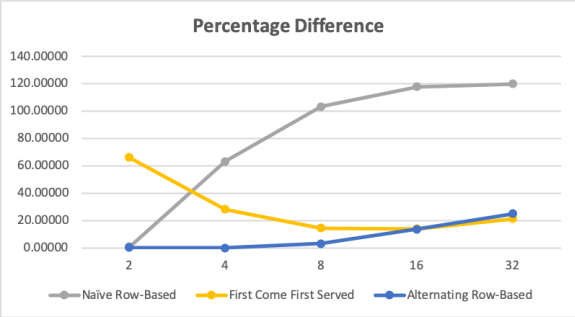
| N | Naïve Row-Based | First Come First Served | Alternating Row-Based |
|---|---|---|---|
| 2 | 0.81952 | 66.06515 | 0.22072 |
| 4 | 63.27981 | 28.26920 | 0.15114 |
| 8 | 103.39878 | 14.65004 | 3.09480 |
| 16 | 117.77011 | 14.08118 | 13.77758 |
| 32 | 119.82522 | 21.47507 | 25.08121 |
| $\infty$ | - | - | - |

Fig. 5: Plots for Theoretical Speedup vs Actual Speedup

(a) Theoretical Speedup vs Actual Speedup




(b) Percentage Difference




But First Come First Served approach uses more communication to ensure that the bottle neck is amortized over all the communication instead of all at once. So the master node does not have to copy all the rows from each task at the end. It will do it as the tasks finish each row. This leads to the First Come First Served partition approach to be faster when the core count is higher.

*3) Alternating - Row Based Partition Scheme:* Comparing this approach with the Naïve approach, for two cores we see that the work load distribution is the same. But why is it faster compared to Naïve approach?. The reason is the master node is computing the high iteration part at the end, so even though the slave finishes at the same time, it has to wait till the master node is ready to receive.In the alternating approach, each process get the high iterating part at middle, so they finish at the same time, thus reducing the wait time for the slave.

Referring to Figure 5b, we see that the First Come First Served approach, is faster than the Alternating approach when the number of tasks is increased. The reason of this is explained in V-B2.

## VI. Conclusion

Now that we have considered all the cases

## VII. Future Work

## References


[1] C. Clapham and J. Nicholson, "Mandelbrot set," 2014. [Online]. Available: https://www.oxfordreference.com/view/10.1093/acref/9780199679591.001.0001/acref-9780199679591-e-1759

[2] A. J. Bernstein, "Analysis of programs for parallel processing," *IEEE Trans. Elec. Comp. EC*, vol. 15, pp. 746–757, 1996.

[3] P. Feautrier, *Bernstein's Conditions*. Boston, MA: Springer US, 2011, pp. 130–134. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_521

[4] S. Ristov, R. Prodan, M. Gusev, and K. Skala, "Superlinear speedup in hpc systems: Why and when?" in *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, 2016, pp. 889–898.

## APPENDIX

| **Time Taken for Serial Code** | | |
|---|---|---|
| **Number of Processors** | **Computation Time** | **Total Time** |
| **Run #1** | 87.3900 | 87.5900 |
| **Run #2** | 87.4100 | 87.6600 |
| **Run #3** | 87.3800 | 87.6400 |
| **Run #4** | 87.3700 | 87.6300 |
| **Run #5** | 87.3800 | 87.6400 |
| **Average Time** | 87.3860 | 87.6320 |
| **Theoretical Speedup Calculation** | | |
| **rp** | 0.997192806 | |
| **rs** | 0.002807194 | |

Fig. 6: Property profile of the diverse library compared to the compound pool.

Listing 1: MonARCH job script for 16 cores

```
#!/bin/bash
#SBATCH --job-name=alter_16
#SBATCH --time=00:30:00
#SBATCH --mem=32G
#SBATCH --ntasks=16
#SBATCH --cpus-per-task=1
#SBATCH --ntasks-per-node=16
#SBATCH --account=fit3143
#SBATCH --constraint=Xeon-Gold-6150
#SBATCH --output=alter_16.out
module load openmpi/1.10.7-mlx

echo "Alternative : 16 cores"

echo "Run 1"
srun mandelbrot_parallel_alternating

echo "Run 2"
srun mandelbrot_parallel_alternating

echo "Run 3"
srun mandelbrot_parallel_alternating

echo "Run 4"
srun mandelbrot_parallel_alternating

echo "Run 5"
srun mandelbrot_parallel_alternating
```

Listing 2: MonARCH job script for 32 cores

```
#!/bin/bash
#!/bin/bash
#SBATCH --job-name=alter_32
#SBATCH --time=00:30:00
#SBATCH --mem=32G
#SBATCH --ntasks=32
#SBATCH --cpus-per-task=2
#SBATCH --ntasks-per-node=16
#SBATCH --account=fit3143
#SBATCH --constraint=Xeon-Gold-6150
#SBATCH --output=alter_32.out
module load openmpi/1.10.7-mlx

echo "Alternative : 32 cores"

echo "Run 1"
srun mandelbrot_parallel_alternating

echo "Run 2"
srun mandelbrot_parallel_alternating

echo "Run 3"
srun mandelbrot_parallel_alternating

echo "Run 4"
srun mandelbrot_parallel_alternating

echo "Run 5"
srun mandelbrot_parallel_alternating
```

| Naïve Row Based Segmentation | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Computer specifications** | | | **a. CPU** - Intel® Xeon® Gold 6150 Processor<br>**b. Logical Cores** - 36 logical cores but 18 cores used because hyperthreadding in turned off<br>**c. RAM** - 32GB<br>**d. Network Speed** - 10GB/s | | | | | | | | | |
| **Value of iXmax** | | | 8000 | | | | | | | | | |
| **Value of iYmax** | | | 8000 | | | | | | | | | |
| **Value of IterationMax** | | | 2000 | | | | | | | | | |
| | **Serial Program** | | **Parallel Program : Naïve Row Based Segmentation** | | | | | | | | | |
| | | | **MPI** | | | | | | | | | |
| | | | **2** | | **4** | | **8** | | **16** | | **32** | |
| | **Computation** | **Total** | **Computation** | **Total** | **Computation** | **Total** | **Computation** | **Total** | **Computation** | **Total** | **Computation** | **Total** |
| Run #1 | 87.3900 | 87.5900 | 43.9425 | 44.3018 | 42.2008 | 42.5519 | 34.6063 | 35.0334 | 21.1970 | 22.0120 | 10.9668 | 11.9910 |
| Run #2 | 87.4100 | 87.6600 | 43.9461 | 44.3432 | 42.1253 | 42.5322 | 34.5715 | 35.0636 | 20.9942 | 21.8928 | 10.9562 | 11.8866 |
| Run #3 | 87.3800 | 87.6400 | 43.9391 | 44.2852 | 42.1180 | 42.5501 | 34.6613 | 35.1447 | 21.2866 | 22.1248 | 10.9698 | 11.8214 |
| Run #4 | 87.3700 | 87.6300 | 43.9427 | 44.2417 | 42.1152 | 42.5353 | 34.6382 | 35.1386 | 21.2353 | 22.0864 | 10.9641 | 11.8484 |
| Run #5 | 87.3800 | 87.6400 | 43.9448 | 44.3308 | 42.1195 | 42.5462 | 34.5103 | 35.0181 | 21.3049 | 22.1670 | 10.9695 | 11.8265 |
| Average time | 87.3860 | 87.6320 | 43.9430 | 44.3006 | 42.1358 | 42.5432 | 34.5975 | 35.0796 | 21.2036 | 22.0566 | 10.9653 | 11.8748 |

Fig. 7: Test Results : Naïve Row Based Segmentation

| First Come First Served Row Based Segmentation | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Computer specifications** | | | **a. CPU** - Intel® Xeon® Gold 6150 Processor<br>**b. Logical Cores** - 36 logical cores but 18 cores used because hyperthreadding in turned off<br>**c. RAM** - 32GB<br>**d. Network Speed** - 10GB/s | | | | | | | | | |
| **Value of iXmax** | | | 8000 | | | | | | | | | |
| **Value of iYmax** | | | 8000 | | | | | | | | | |
| **Value of IterationMax** | | | 2000 | | | | | | | | | |
| | **Serial Program** | | **Parallel Program : First Come First Served Row Based Segmentation** | | | | | | | | | |
| | | | **MPI** | | | | | | | | | |
| | | | **2** | | **4** | | **8** | | **16** | | **32** | |
| | **Computation** | **Total** | **Computation** | **Total** | **Computation** | **Total** | **Computation** | **Total** | **Computation** | **Total** | **Computation** | **Total** |
| Run #1 | 87.3900 | 87.5900 | 86.7560 | 87.0793 | 28.9501 | 29.3563 | 12.4648 | 12.8833 | 5.8283 | 6.5745 | 2.8155 | 3.7470 |
| Run #2 | 87.4100 | 87.6600 | 86.8915 | 87.2788 | 28.9540 | 29.3673 | 12.4806 | 12.9451 | 5.8286 | 6.5626 | 2.8122 | 3.6985 |
| Run #3 | 87.3800 | 87.6400 | 87.0907 | 87.5975 | 28.9526 | 29.3488 | 12.4622 | 12.9608 | 5.8332 | 6.5652 | 2.8114 | 3.6603 |
| Run #4 | 87.3700 | 87.6300 | 86.8756 | 87.3341 | 28.9479 | 29.3443 | 12.4654 | 12.9414 | 5.8327 | 6.5661 | 2.8209 | 3.6930 |
| Run #5 | 87.3800 | 87.6400 | 86.7836 | 87.1402 | 28.9617 | 29.4129 | 12.4897 | 12.9438 | 5.8440 | 6.5926 | 2.8118 | 3.6662 |
| Average time | 87.3860 | 87.6320 | 86.8795 | 87.2860 | 28.9532 | 29.3659 | 12.4725 | 12.9349 | 5.8334 | 6.5722 | 2.8144 | 3.6930 |

Fig. 8: Test Results : First Come First Served Row Based Segmentation

| Alternating Row Based Segmentation | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Computer specifications** | | | **a. CPU** - Intel® Xeon® Gold 6150 Processor<br>**b. Logical Cores** - 36 logical cores but 18 cores used because hyperthreadding in turned off<br>**c. RAM** - 32GB<br>**d. Network Speed** - 10GB/s | | | | | | | | | |
| **Value of iXmax** | | | 8000 | | | | | | | | | |
| **Value of iYmax** | | | 8000 | | | | | | | | | |
| **Value of IterationMax** | | | 2000 | | | | | | | | | |
| | **Serial Program** | | **Parallel Program : Alternating Row Based Segmentation** | | | | | | | | | |
| | | | **MPI** | | | | | | | | | |
| | | | **2** | | **4** | | **8** | | **16** | | **32** | |
| | **Computation** | **Total** | **Computation** | **Total** | **Computation** | **Total** | **Computation** | **Total** | **Computation** | **Total** | **Computation** | **Total** |
| Run #1 | 87.3900 | 87.5900 | 43.7684 | 44.1128 | 21.7582 | 22.1766 | 10.9706 | 11.4860 | 5.5723 | 6.3163 | 2.8558 | 4.2383 |
| Run #2 | 87.4100 | 87.6600 | 43.3013 | 43.6628 | 21.7036 | 22.1388 | 10.9876 | 11.5328 | 5.7514 | 6.5377 | 2.8577 | 3.7094 |
| Run #3 | 87.3800 | 87.6400 | 43.3493 | 43.7057 | 21.7034 | 22.1025 | 10.9759 | 11.5218 | 5.7864 | 6.6517 | 2.8581 | 3.7264 |
| Run #4 | 87.3700 | 87.6300 | 43.8933 | 44.3467 | 21.7056 | 22.1142 | 10.9882 | 11.5390 | 5.8019 | 6.6328 | 2.8551 | 3.7429 |
| Run #5 | 87.3800 | 87.6400 | 43.9282 | 44.3524 | 21.7035 | 22.0973 | 10.9799 | 11.5222 | 5.7990 | 6.6224 | 2.8557 | 3.7354 |
| Average time | 87.3860 | 87.6320 | 43.6481 | 44.0361 | 21.7149 | 22.1259 | 10.9804 | 11.5203 | 5.7422 | 6.5522 | 2.8565 | 3.8305 |

Fig. 9: Test Results : Alternating Row Based Segmentation.