Efficient Generation of Mandelbrot Set using Message Passing Interface

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

***Abstract*—With the increasing need for safer and reliable systems, Mandelbrot Set's use in the encryption world is evident to everyone. This document aims to provide an efficient method to generate this set using data parallelism. First Bernstein's conditions [1] are used to ensure that the Data is parallelizable when generating the Mandelbrot Set. Then Amdhal's Law is used [2] to calculate the theoretical speed up, to be used to compare three partition schemes. The three partition schemes discussed in this document are the Naïve Row Segmentation, the First Come First Served Row Segmentation and the Alternating Row Segmentation. The Message Parsing Interface (MPI) library in C [3] is used for all of the communication. After testing all the implementation on MonARCH, the results demonstrate that the Naïve Row Segmentation approach did not perform as par. But the Alternating Row Segmentation approach performs better when the number of tasks are < 16, where as the First Come First Served approach performs better when the number of tasks is ≥ 16.**

***Index Terms*—mandelbrot set, data parallelism, row segmentation, alternative, naive, first come first served, MonARCH, MPI, distributed computing**

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 $z = 0$ [4]. Aside from the inherent beauty of the Mandelbrot Set, some of the mysteries of this set is still unknown to Mathematicians [5]. So having a way to generate this set efficiently can increase the research in this area and improve our understanding of this set.

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

In order to generate the Mandelbrot set for a $m \times n$ image I, we need to execute equation 1 on each pixel of the image. The complex part of the equation is determined using the m and n values of the image. Then the Iteration at which either the magnitude (z^2) exceeding the escape radius or the *iterationMax*, is recorded. Both the *iterationMax* and Escape Radius are pre-determined and can affect the time taken to generate the set. The iteration is then used to determine the pixel color in the image I. Therefore each pixel of the image can be represented using equation 2.

This method is known as Mandelbrot generation with Boolean escape time. Figure 1 shows the image of a Mandelbrot Set generated using this method.

$$I_{i,j} = Iteration_{i,j} \tag{2}$$
$$\text{where } i = 1...m, j = 1...n$$

There are many uses of the Mandelbrot Set in the field of encryption such as Image Encryption [6], Key Exchange Protocols [7] and Key Encryption [8]. With the increasing need for cyber-security in the modern era, safer and reliable encryption techniques will play a major role in the future.

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 generation of Mandelbrot Set is data parallelizable [1]. Then the theoretical speed up is calculated using Amdhal's law [2], which is used as the objective for each partition scheme. The workload is divided among N number of tasks, in different segmentation configurations for each of the partition schemes. Then the actual speed up is obtained for all of the partition schemes with different task configurations. Using the theoretical speed up and the actual speed up, the percentage difference of each of the partition schemes are then obtained and used for the comparison. All of the inter process communication is done using the Message Parsing Library (MPI) in C [3].

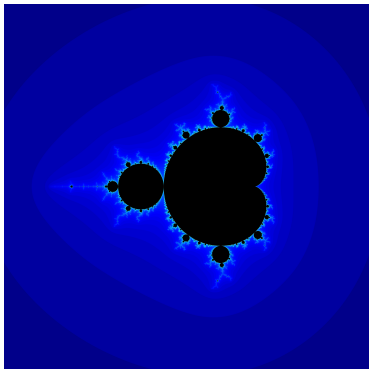


Fig. 1: Mandelbrot Set generated using Boolean Escape Time algorithm [9]

II. Preliminary Analysis

In this section, we examine whether the generation of the Mandelbrot Set is data parallelizable and if there is any advantage of doing it.

A. Data Parallelizability of the Generation of the Mandelbrot Set

In order to ensure that the process of generating the Mandelbrot Set is data parallelizable, we can use Bernstein's conditions [1]. Bernstein's conditions are a simple verification for deciding if operations and statements can work simultaneously without changing the program output and allow for data parallelism [10]. According to Bernstein, if two tasks satisfy the following equations, then they are parallelizable.

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

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

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

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

Using equation 2, we can derive the input and output equations for any two neighboring pixels in Image I which will be computed by two tasks P_1 and P_2.

$$I_1 = \textit{Iteration}_{i,j} \tag{6}$$

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

$$I_2 = \textit{Iteration}_{i,j+1} \tag{8}$$

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

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

$$\textit{Iteration}_{i,j} \quad \cap \quad I_{i,j+1} = \emptyset \tag{10}$$

$$\textit{Iteration}_{i,j+1} \quad \cap \quad I_{i,j} = \emptyset \tag{11}$$

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

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

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 the 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_s). Then using equation 13 we can get the r_p value.

$$r_p = \frac{t_p}{t_s} \tag{13}$$

Then after calculating the r_p value, we can use Amdahl's Law [2] to calculate the theoretical speed up to generate the Mandelbrot set. Amdahl 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 [11], then the speed up can be stated as

$$S_p = \frac{1}{r_s + \frac{r_p}{p}} \tag{14}$$

Using Amdahl's law, we are able to generate Table I with the theoretical speed up values.

TABLE I: Number of tasks vs Theoretical Speed up factor

Number of Tasks	Speed up Factor
2	1.99440
4	3.96659
8	7.84583
16	15.35350
32	29.43820
∞	356.22764

More details in the Appendix - Figure 7

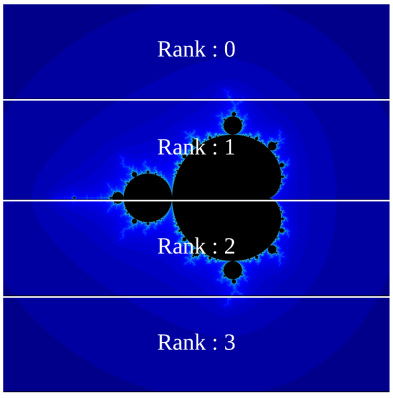
So having determined that the generation of the Mandelbrot Set is embarrassingly parallelizable and having calculated the theoretical speed up, next we look into the designs of the partition schemes for parallelizing data.

III. Design of the Partition Schemes

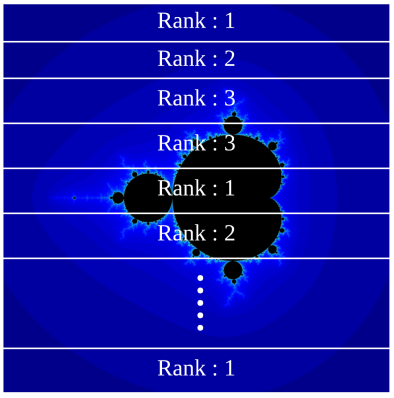
For the design of the partition schemes, we will be comparing three partition schemes. In all of these schemes, we will be writing the Mandelbrot Set to the file in only the master node. So each approach will send the data back to the master node, therefore all of them will be using a Master-Slave Architecture. 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

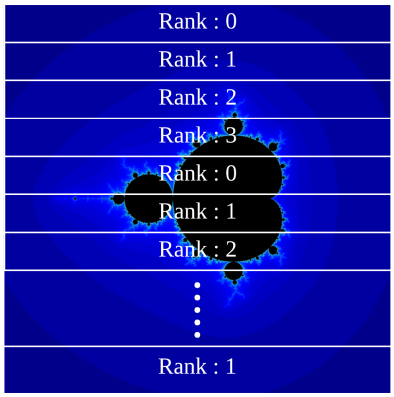
As shown in Figure 2, the final image with $iYmax$ rows is divided equally among N number of processors. So each process will pre-calculate its start and end positions and generate the Mandelbrot Set for that portion. Equation 15 , 16 and 18 are used to calculate the start (S_r) and end point (E_r) for each processor with rank (r) out of N tasks.



(a) Naive : Row Based Partition Scheme



(b) First Come First Served : Row Based Partition Scheme



(c) Alternating : Row Based Partition Scheme

Fig. 2: Partition Schemes when $N = 4$

$$S_r = \frac{iYmax}{N} \times r \tag{15}$$

$$E_r = \frac{iYmax}{N} \times (r \times 1) - 1 \tag{16}$$

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

$$rows\ per\ task = \frac{iYmax}{number\ of\ tasks} \tag{17}$$

In the case where the number of rows cannot be equally divided among the number of tasks (N), the last node (rank = N - 1) will calculate the remaining set of rows. This will not effect the overall performance significantly since the top and bottom section of the Mandelbrot Set takes less time to process compared to the middle section. So the end point (E_r) for the last task can be represented by Equation 18.

$$E_r = E_r + (iYmax \mod N) \tag{18}$$

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.

$$T_m = N - 1 \tag{19}$$

Equation 19 shows the (T_M) total number of messages passed between N number of tasks, when using the Naive Partition Scheme. The technical flowchart for the Naïve row based partition scheme is shown in Figure 3.

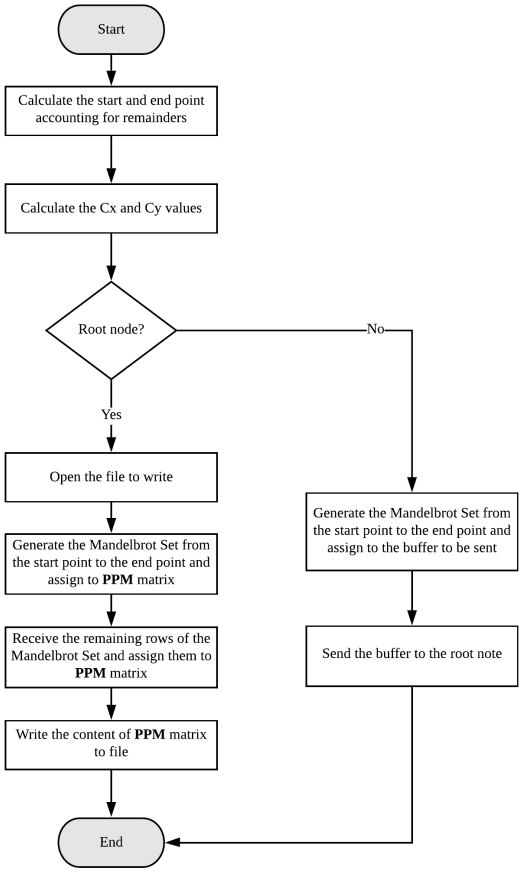


Fig. 3: Technical Flowchart for Naïve Row Based Partition Scheme

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

This partition scheme shares similarity with schedulers in operating systems, where the master node sends work, based on the availability. As shown in Figure 2, the order in which

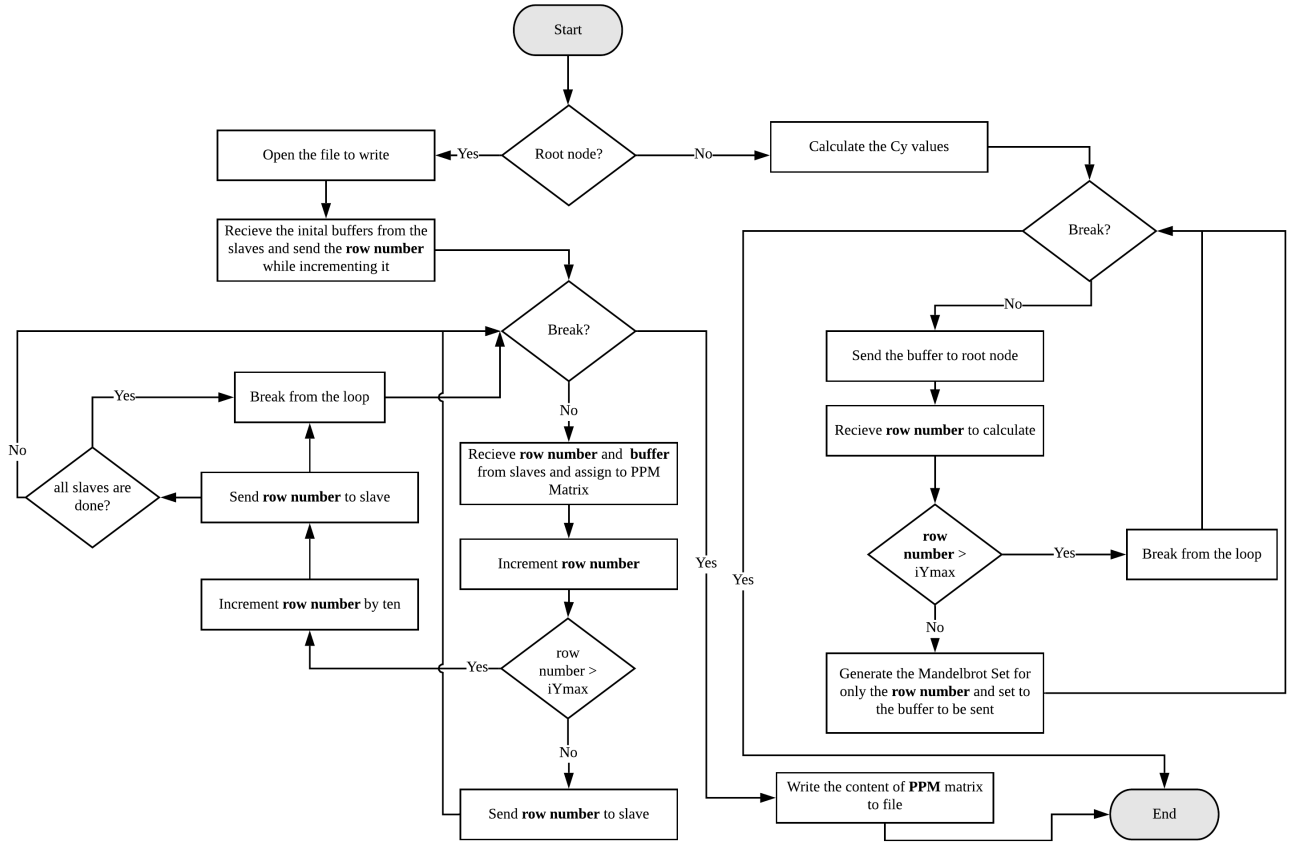


Fig. 4: Technical Flowchart for First Come First Served Row Based Partition Scheme

node performs which row is based on the availability of the node. If one node finishes the row 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 as it receives will copy each row 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 task count or odd number of rows is not present, since its based on the availability of tasks. Due to the nature of work allocation, this partition scheme is named as First Come First Served. The main thing to note is that the master node does not generate any rows. It will only be delegating work to the slave nodes. Another observation would be that this method has a high communication overhead as the master node has to send each row number to the slaves and then receive each row from the slaves.

$$T_m = iYmax \times 2 \tag{20}$$

Equation 20 shows the (T_M) total number of messages passed between N number of tasks, when using the First Come First Served Partition Scheme. So we can observe that the total number of tasks does not effect the number of messages when using this approach.The technical flowchart for the First Come First Served row based partition scheme is shown in Figure 4.

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 2 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 starts 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 increments with the number of tasks. Similar to III-A, each process gets 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. Similar to III-A, the processing time for the

remainder is insignificant compared to the rest. 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.

$$T_m = N - 1 \tag{21}$$

Equation 21 shows the (T_M) total number of messages passed between N number of tasks, when using the Alternating Partition Scheme. The technical flowchart for the Alternative row based partition scheme is shown in Figure 5.

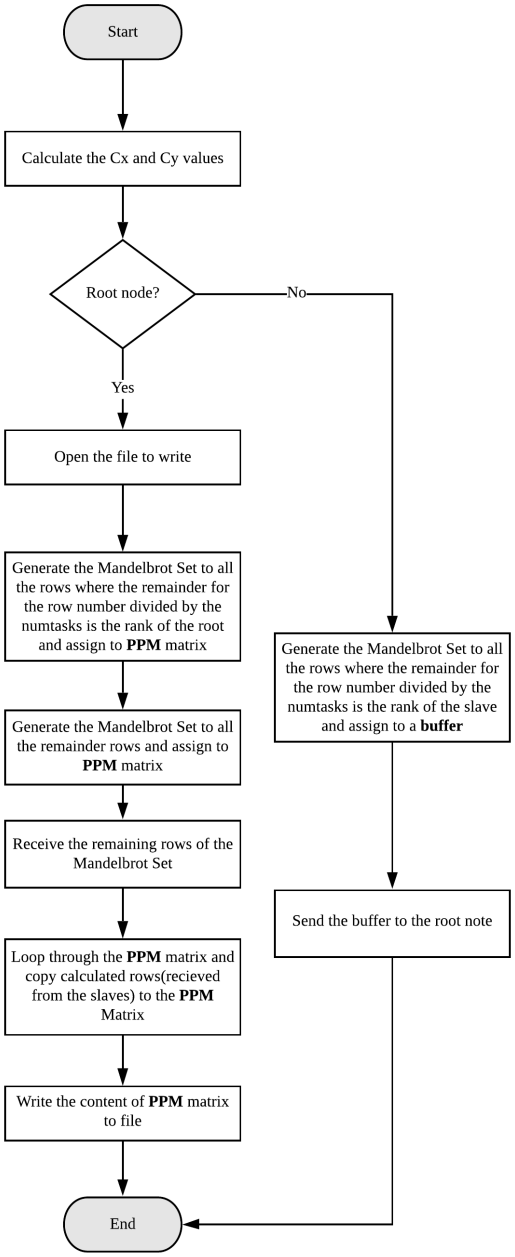


Fig. 5: Technical Flowchart for Alternative Row Based Partition Scheme

IV. Methodology

In order to calculate the run times for the three partition schemes, multiple tests were done. For each test, we set the size of the image to 8000 × 8000 with 2000 iterations and the Escape Radius is set to 400. An assumption used for the test cases was that the task count will always be a multiple of two, but all of these partition schemes will work with any other task 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 tasks.

All of the tests were done on the MonARCH (Monash Advanced Research Computing Hybrid) HPC/HTC cluster [12]. The problem with using a cluster is that the scheduler decides 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 test, two 16 core processors were used. Listing 2 and 3 of the appendix, shows the job file used to run the parallel code with 16 and 32 cores respectively. 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 time taken to write data to the file. All the results (Figure 8, Figure 9, Figure 10), and one MonARCH output (Listing 1) is 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. The actual speed up from Table IIa is calculated using equation 22.

$$\textit{Actual Speed up} = \frac{\textit{Total time by single task}}{\textit{Total time by multiple tasks}} \tag{22}$$

Then using the Theoretical Speed up for n tasks as T_n and Actual Speed up as A_n from Table IIa, we calculate the percentage difference using equation 23.

$$\textit{Percentage Difference} = \frac{|T_n - A_n|}{\frac{T_n + A_n}{2}} \times 100 \tag{23}$$

Each partition scheme is then compared and discussed based on the following criteria.

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

TABLE II: Analysis of Theoretical Speedup vs Actual Speedup

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

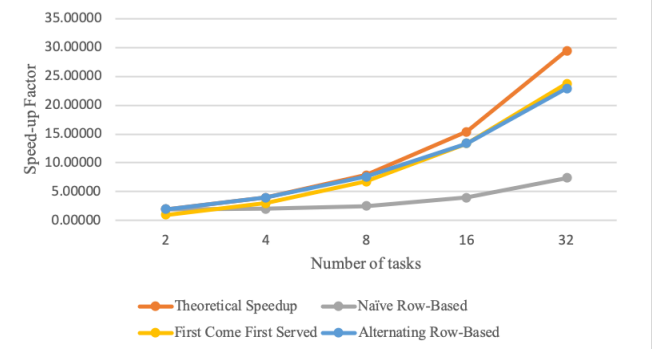
N	Theoretical Speedup	Naïve Row-Based	First Come First Served	Alternating Row-Based
2	1.99440	1.97812	1.00396	1.99000
4	3.96659	2.05984	2.98414	3.96060
8	7.84583	2.49809	6.77486	7.60671
16	15.35350	3.97305	13.33375	13.37449
32	29.43820	7.37966	23.72932	22.87749
∞	356.22764	-	-	-

(b) Percentage Difference of all the partition schemes

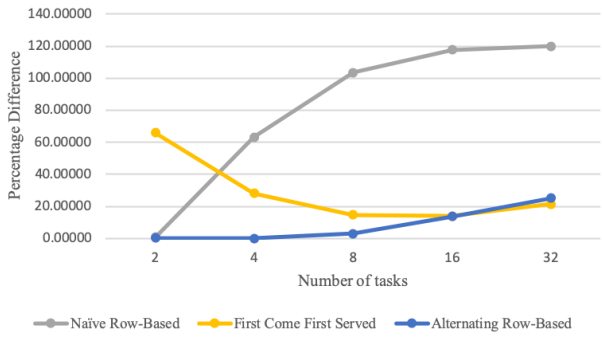
N	Naïve Row-Based	First Come First Served	Alternating Row-Based
2	0.81952	66.06515	0.22072
4	63.27981	28.26920	0.15114
8	103.39878	14.65004	3.09480
16	117.77011	14.08118	13.77758
32	119.82522	21.47507	25.08121
∞	-	-	-

Fig. 6: Plots for Theoretical Speedup vs Actual Speedup

(a) Theoretical Speedup vs Actual Speedup



(b) Percentage Difference



A. *Theoretical Speed up vs Actual Speed up*

In this section we will compare how the actual speed up compares 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 two tasks, the theoretical speed up and the actual speed up is almost equal. The main reason for this is when we are using two cores, the workload is divided equally among the two tasks. Both the tasks will be doing the high iteration and low iteration part of the image. But when the number of tasks increase, the actual speed up becomes poor. The reason for this is, even though the tasks get equal rows, 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 completed to write the file. Finally comparing the percentage difference using Table IIb we see that this partition scheme does not perform well when the task count increases.

2) First Come First Served - Row Based Partition Scheme: This method sends the rows based on the availability of the task. Looking back at Table IIa, we see that the performance of the two tasks test is almost equal to no speed up at all. The main reason for this is when we use two tasks, the slave generates all the rows in the image, while the master sends and receives data. So the actual speedup should be ≤ 1, but our actual speed up > 1. Why? According to a paper published in 2016, by a group of researchers [13], 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. Another explanation for this phenomena is that Amdal's Law does not take into account the communication overhead. Therefore, the speed up can be ≥ 1 even with a lot of communication. Next, comparing the theoretical speed up vs actual speed up for multiple tasks, 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 (r_n) of each task. Referring to Table IIa, this approach is almost equal to the theoretical speed up 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 tasks increases, this method is able to perform in a sub-linear order staying close to the theoretical limit. Referring to Table IIb, we see that the the percentage difference increases with each increment in the number of tasks.

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 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 number of tasks 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 number of tasks 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 the master receives the data at the same time from all the slaves. The problem is that, if the work load is divided equally, then all the slaves finish at the same time thus causing a bottle neck at the master node.

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 number of tasks 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 the Naïve approach? The reason is that, in the Naïve approach 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 task gets the high iterating part in the middle, so they finish at the same time, thus reducing the wait time for the slave.

Referring to Figure 6b, we see that the First Come First Served approach, is faster than the Alternating approach when the number of tasks is increased. The reason for this is explained in V-B2. We see that more communication leads to slower speed up for the First Come First Served approach but as the number of tasks increased, both Naive and Alternating methods started slowing down. This can also be due to the increase in the total number of communication between the master and slave for the Naïve and Alternating approaches, whereas it was constant for the First Come First Served approach. This was represented by equations 19, 20 and 21.

VI. Conclusion

Now that we have analyzed all the cases, we can rank each of the partition schemes based on their performance.

Referring to Figure 6b, the Naïve Row Based Segmentation performed the worst with increasing percentage difference at each doubling of the number of tasks except when the number of tasks were two. Next, the First Come First Served partition scheme performed on average for most of the test cases but performed the best when the number of tasks were > 16. Finally, the Alternating Row Based partition scheme performed the best when the number of tasks > 16.

The speed up differences between each partition scheme can be explained due to the bottle necks and communication overhead of each approach. The partition scheme that is able to reduce each of the factors is able to perform better than the rest.

Conclusively, we can state that First Come First Served partition scheme is effective when the number of tasks are high while the Alternative partition scheme is effective when the number of tasks are low.

VII. Future Work

This report opens up future research paths for each of the three methods. The effect of having the number of tasks greater than 32 can be a one of areas that can be experimented in the future. Another path is removing the assumption of having tasks as multiples of two and having an odd number of tasks. This will allow us to observe how the remainder affects each of the methods, especially Naïve and Alternative since the workload will be imbalanced for some nodes. This reports paves the way to exploring the efficient ways of generating the Mandelbrot Set in parallel architectures.

References

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

[2] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967, pp. 483–485.

[3] "Home," Nov 2018. [Online]. Available: https://www.openmp.org/

[4] 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

[5] E. Lamb, "A few of my favorite spaces: The mandelbrot set," Jan 2017. [Online]. Available: https://blogs.scientificamerican.com/roots-of-unity/a-few-of-my-favorite-spaces-the-mandelbrot-set/

[6] Y. Sun, R. Kong, X. Wang, and L. Bi, "An image encryption algorithm utilizing mandelbrot set," in *2010 International Workshop on Chaos-Fractal Theories and Applications*, Oct 2010, pp. 170–173.

[7] M. Alia and A. Samsudin, "New key exchange protocol based on mandelbrot and julia fractal set," *International journal of computer science and network security*, vol. 7, no. 2, pp. 302–307, 2007.

[8] S. Agarwal, "Symmetric key encryption using iterated fractal functions," *International Journal of Computer Network and Information Security*, vol. 9, no. 4, p. 1, 2017.

[9] R. Code, "Mandelbrot set." [Online]. Available: http://rosettacode.org/wiki/Mandelbrot_set#PPM_non_interactive

[10] 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

[11] J. L. Gustafson, "Reevaluating amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[12] MonARCH, "Welcome to the monarch documentation!" [Online]. Available: https://docs.monarch.erc.monash.edu/

[13] 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. 7: Test Results : Computation vs Total Time for Serial Generation

***All times are expressed in seconds*

Listing 1: Sample MonARCH output file for 32 cores

```
Alternative : 32 cores
Run 1
File: Mandelbrot.ppm successfully opened for writing.
Computing Mandelbrot Set. Please wait...
Mandelbrot computational process time: 2.855784
Completed Computing Mandelbrot Set.
File: Mandelbrot.ppm successfully closed.
Mandelbrot total process time: 4.238303
Run 2
File: Mandelbrot.ppm successfully opened for writing.
Computing Mandelbrot Set. Please wait...
Mandelbrot computational process time: 2.857663
Completed Computing Mandelbrot Set.
File: Mandelbrot.ppm successfully closed.
Mandelbrot total process time: 3.709431
Run 3
File: Mandelbrot.ppm successfully opened for writing.
Computing Mandelbrot Set. Please wait...
Mandelbrot computational process time: 2.858120
Completed Computing Mandelbrot Set.
File: Mandelbrot.ppm successfully closed.
Mandelbrot total process time: 3.726426
Run 4
File: Mandelbrot.ppm successfully opened for writing.
Computing Mandelbrot Set. Please wait...
Mandelbrot computational process time: 2.855122
Completed Computing Mandelbrot Set.
File: Mandelbrot.ppm successfully closed.
Mandelbrot total process time: 3.742856
Run 5
File: Mandelbrot.ppm successfully opened for writing.
Computing Mandelbrot Set. Please wait...
Mandelbrot computational process time: 2.855696
Completed Computing Mandelbrot Set.
File: Mandelbrot.ppm successfully closed.
Mandelbrot total process time: 3.735431
```

Listing 2: 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 3: 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 **b. Logical Cores** - 36 logical cores but 18 cores used because hyperthreadding in turned off **c. RAM** - 32GB **d. Network Speed** - Gigabit Ethernet									
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. 8: Test Results : Naïve Row Based Segmentation

***All times are expressed in seconds*

First Come First Served Row Based Segmentation												
Computer specifications			**a. CPU** - Intel® Xeon® Gold 6150 Processor **b. Logical Cores** - 36 logical cores but 18 cores used because hyperthreadding in turned off **c. RAM** - 32GB **d. Network Speed** - Gigabit Ethernet									
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. 9: Test Results : First Come First Served Row Based Segmentation

***All times are expressed in seconds*

Alternating Row Based Segmentation												
Computer specifications			**a. CPU** - Intel® Xeon® Gold 6150 Processor **b. Logical Cores** - 36 logical cores but 18 cores used because hyperthreadding in turned off **c. RAM** - 32GB **d. Network Speed** - Gigabit Ethernet									
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. 10: Test Results : Alternating Row Based Segmentation.

***All times are expressed in seconds*