Parallel Implementation and Analysis of Mandelbrot Set using Message Passing Interface

Siyuan Yan
Monash University
E-mail: syan0009@student.monash.edu

***Abstract*—The Mandelbrot Set is a mathematical application that has become famous outside mathematics because of its appeal and as an example of a complex structure producing from the application of simple rules. This report presents a parallel partition scheme for efficient computing the Mandelbrot Set using Message Passing Interface (MPI) library in C language. To verify the parallelizability of the algorithm, Bernstein Condition is employed. The performance of the algorithm is based on a proposed partition scheme chosen after comparison is used to studied and analyzed using Amdahl's law. The partition scheme is implemented and tested on the monarch clusters. A near-linear speed up for an increasing number of logical processors in terms of the comparison between the partition scheme using MPI and the serial implementation will indicate the success of the partition scheme.**

Keywords- Mandelbrot Set, MPI, Parallel computing, partition scheme, Linear speed up.

I. INTRODUCTION

The Mandelbrot set is a famous mathematical object defined by a very simple rule. Nevertheless, the Mandelbrot set itself possesses interesting and complex properties which can be seen graphically, shown in Figure 1. This is related to the fact that it is a fractal object. [1]. A fractal is a never-ending pattern. Fractals are infinitely complex patterns that are created by repeating a simple process over and over in an ongoing feedback loop [2]. On the other hand, one of the most appealing features about the Mandelbrot set is its self-similarity, which means you can find the geometry in nature.

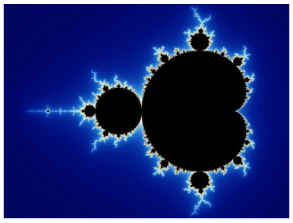


Fig. 1 Mandelbrot Set

Mathematically, the Mandelbrot set is a set of complex numbers c for that a quadratic recurrence equation (1). The equation does not diverge when iterated from z = 0. The Mandelbrot set images may be created by sampling the complex numbers and testing. Given an X-Y coordinate, for each point treating the real and imaginary parts of c as image coordinates c_x, c_y on the complex plane. Treating the real and imaginary parts of z as z_x, z_y that starts at (0,0), during each iteration, the new z is calculated by the old z squared plus c, the value z is checked whether it reaches a critical escape condition. If it has reached the condition, the iteration is stopped and the point color is decided based on how fast it reached the condition. If the value z successful reaches the maximum iteration without reaching the critical escape condition, it is the member of the Mandelbrot set, shown as black.

$$z_{n+1} = z_n^2 + c \tag{1}$$

In this report, the parallelization is realized using the Message Passing Interface's (MPI) collective communication routines. Two partition schemes called row segmentation partition and fine-grain, static, row region data partitioning is used to divide the image into sub-blocks, mapping the sub-blocks into various processors so that each processor can compute the part assigned to it. The comparison between the computation time of two partition schemes will be used to observe and analyse, then the partition scheme that has the best performance will be chosen.

The performance of the Mandelbrot set algorithm for the parallelized partition scheme chosen is evaluated by calculating the speed up factor based on the execution time of the parallelized program. The calculated speed up factor is then used to be compared with the theoretical speed up obtained from the Amdahl' law analysis to check whether the partition scheme can reach linear speed up. A hypothesis is applied, which is the balanced computation distribution between processors can significantly affect the total time spent.

II. PRELIMINARY ANALYSIS

A. Parallelizability of Matrix Multiplication

To verify whether each pixel in the Mandelbrot set image can be processed independently. The Bernstein's condition is employed, which two processes can be proved as independent to each other and can be parallelizable if the three conditions are satisfied [3]:

$I_0 \cap O_1 = \emptyset$ (anti independency)

$I_1 \cap O_0 = \emptyset$ (flow independency)

$O_0 \cap O_1 = \emptyset$ (output independency)

with $I_{0\ and}\ O_0$ represent the input and output for the first process P_0, I_1 and O_1 represent the input and output for the second process P_1.

Equation (2) and (3) denote the formula for calculating two adjacent pixels of the images, representing the P_0 and P_1 respectively. (x, y) and (x, y+1) are the coordinates in the image, which means the formula calculates two adjacent pixels. z_{n+1} is got by recursive calculating the equation n times.

$$z_{n+1(x,\,y)} = z_{n\,(x,\,y)}{}^2 + c_{(x,\,y)} \tag{2}$$

$$z_{n+1(x,\,y+1)} = z_{n\,(x,\,y+1)}{}^2 + c_{(x,\,y+1)} \tag{3}$$

Whereby

$I_0 = z_{n\,(x,\,y)}{}^2 + c_{(x,\,y)}$ $\quad O_0 = z_{n+1(x,\,y)}$

$I_1 = z_{n\,(x,\,y+1)}{}^2 + c_{(x,\,y+1)}$ $\quad O_1 = z_{n+1(x,\,y+1)}$

By applying Bernstein's conditions, we obtain three conditions such that

$$z_{n\,(x,\,y)}{}^2 + c_{(x,\,y)} \cap z_{n+1(x,\,y+1)} = \emptyset$$

$$z_{n\,(x,\,y+1)}{}^2 + c_{(x,\,y+1)} \cap z_{n+1(x,\,y)} = \emptyset$$

$$z_{n+1(x,\,y)} \cap z_{n+1(x,\,y+1)} = \emptyset$$

Since all three conditions are satisfied, P_o and P_1 can be executed in parallel, which means the calculation of individual element in the image can be parallelizable.

B. Theoretical Speed Up of Mandelbrot algorithm

To compute the theoretical speed up of the Mandelbrot algorithm in multiple processes, a series-based Mandelbrot algorithm is used. The algorithm consists of two parts. (a) Perform Mandelbrot set computation (b) Write all results into a .ppm file. All two parts of the algorithm are implemented serially. The computation and write time is recorded and tabulated in table A in appendices.

Then, Amdahl's law as am equation (4) is used to computed the theoretical acceleration when implementing Mandelbrot set algorithm using multiprocessors.

$$S(p) = 1 + \frac{1}{r_s + \frac{r_p}{p}} \tag{4}$$

With

r_p = parallel ratio (parallelizable portion)

r_s =serial ratio (non – parallelizable portion)

p = number of processors

Table I shows the calculated theoretical speed up factors S(p) when using the number of processors p=2,4,6 and 8 (i.e. for a single multicores computer with 2,4,6,8 logical processors)

TABLE I
THEORETICAL SPEED UP USING ADMDAHL'S LAW

Number of Processors, p	2	4	6	8
Speed Up Factors, S(p)	1.983	3.8994	5.7526	7.5455

The r_s in our case is the portion (a) for computing the Mandelbrot set, the r_p is the portion (b) for writing all calculated results into a .ppm file. The portion (a) can be parallelizable using multiple processors.

III. DESIGN OF PARTITION SCHEME

For both row segmentation partition and fine-grain, static, row region data partitioning schemes, multiple 1D dynamic arrays are used to store the data that are computed by the Mandelbrot algorithm. The reasons for using 1D dynamic array are it can save memory as the data are stored in contiguous memory location, the memory can be allocated manually during run time and it is also easy to manage and access data. Each pixel is stored in a struct (a composite data type in C programming) that consists of three variables r, g, b, the purpose of using struct is to store data easily. After the completion of the computation part, the data are collected by a 1D dynamic array of the root node (i.e. the processor with rank 0). Finally, the collected data in the array is written in a .ppm file by the root node.

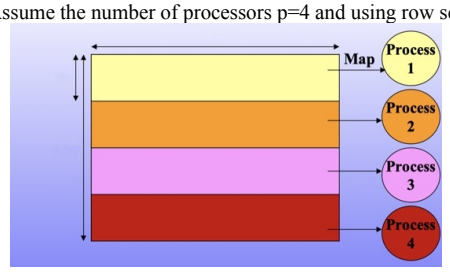
A. Row Segmentation Partition Scheme

In the row segmentation partition scheme (see Figure 2), the image will be generated is divided into multiple equal sub-blocks that are assigned to all processors, each processor is responsible for its own part and compute all the pixel in the part. The rows each processor assigned are calculated using the equation (5) and are stored using 1D dynamic array respectively with size *Rows per processor × Columns of Image.*

$$Rows\ per\ processor = \frac{Rows\ number\ of\ Image}{Number\ of\ processors} \qquad (5)$$

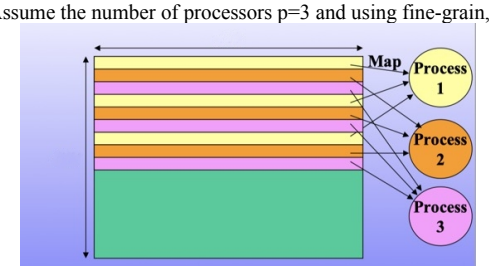
If the image cannot be divided equally, the remaining part can be assigned to the root node. It will not affect the efficiency as the maximum number of remaining part is only *Number of processors -1* rows. Then, all sub blocks calculated are collected in order by the root node using MPI_Gather that can gather distinct messages from each task in the group to a single destination task, the remaining part is added to last rows, the gathered data are stored a 1D dynamic array with size *Rows of Image × Columns of Image.* Upon the completion of receiving data by root node, the data in the 1D dynamic array are written in a .ppm file, the write operation is done by the root node. Figure 3 illustrates the technical flowchart for row segmentation partition scheme

Assume the number of processors p=4 and using row segmentation partition



(a)Row Segmentation Partition Scheme

Assume the number of processors p=3 and using fine-grain, static, row region data partitioning



(b)fine-grain, static, row region data partitioning

Fig .2 Partition Schemes

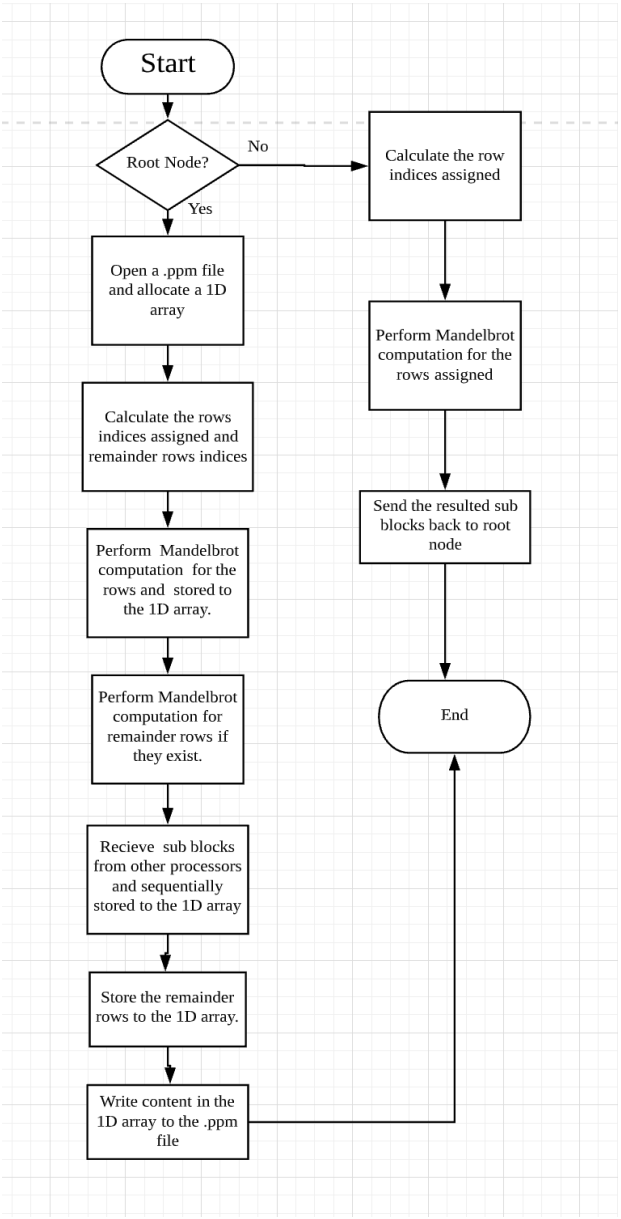


Fig. 3 Flowchart of Row Segmentation Partition Scheme

B. Fine-grain, static, row region data partitioning

For fine-grain, static, row region data partitioning (see Figure 2b), the image will be computed is divided into *Rows number of image* rows, the first *Number of Processors* rows are assigned sequentially to processors, then at each iteration, the row should be assigned into each processor is calculated using the equation (6).

$$Row_i = Row_{i-1} + Number\ of\ Processors \tag{6}$$

The stride of each iteration is *Number of Processors*, this approach can make sure that equally computation is performed for each processor. Similarly, if the image cannot be divided equally, the remaining part could be assigned to the root node and we have proved it will not affect the efficiency of the algorithm. For each processor, the resulted data can be stored sequentially to a 1D dynamic array respectively. The remaining part is stored sequentially into an extra 1D dynamic array so that it can be easily accessed. Then, the root node receives all data by MPI_Gather illustrated in Part A in Section II from other processors and stores the data sequentially to a receiving buffer (i.e. a 1D dynamic array) just like the Figure 4.

Assume the number of processors p=4

Rank 0	Rank 1	Rank 2	Rank 3

Fig.4 The receiving buffer

There are two important points for the receiving buffer when receiving the data from other processors. (1) The order of data in each sub-block cannot be changed. (2), The sub-blocks need to be stored sequentially into the receiving buffer. To make sure the image can be generated successfully, the two conditions need to be satisfied. If remaining rows exist, the remaining part collected by the root node is stored into an extra array. Upon the completion of receiving data by the root node, the data in the receiving buffer will be written into a .ppm file by the root node. The root node will first write the first data (the data consist of r, g, b using struct has mentioned) in each sub-block in the receiving buffer into the .ppm file in order. Then, the root node will write the second data in each-sub-block in order into the .ppm file. Following that, the root node will repeat the same process until all data in each sub-block is written into the .ppm file. If the array of remaining part exists, the data is written sequentially into the .ppm file at the end. The technical flowchart of the fine-grain, static, row region data partitioning scheme is depicted in Figure 5.

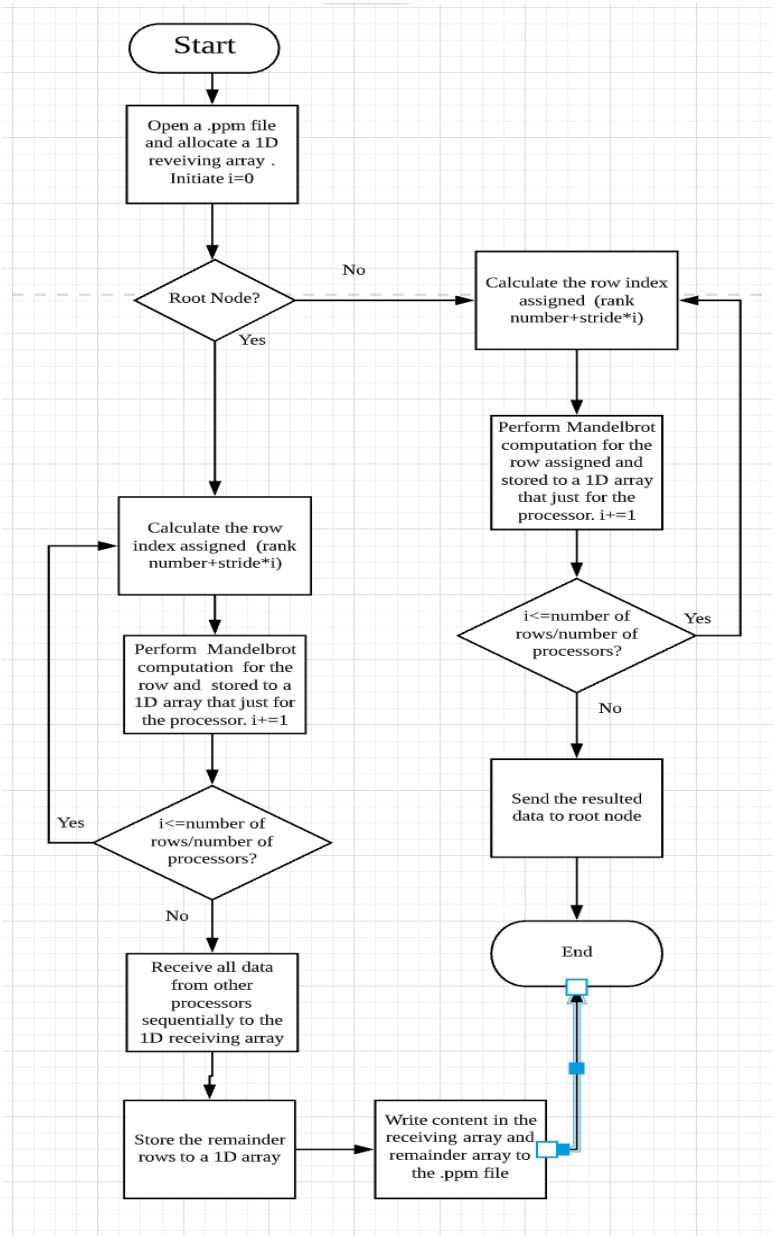


Fig .5 Flowchart of Fine grain, static, row region partitioning scheme

IV. IMPLEMENTATION OF PARTITION SCHEMES

A. Implementation

The size of the image will be generated is 8000×8000, it will be used to test the performance of our parallelized algorithm using the fine grain, static, row region data partitioning. The colour of each pixel in the image is decided by the R, G, B colour value. Each parameter of R, G, B is *unsigned char* (i.e. 1 byte), they define the intensity of the colour as an integer between 0 and 255. A struct in C language is defined to store the R, G, B so that the pixels can be computed and stored easily. For the partition schemes illustrated in Section III, the implementation of the parallelized algorithms is based on C language with the use of MPI library. The writing of resulted data into the file is implemented serially.

B. Comparison Between Different Schemes

The Row Segmentation Partition divides the image into equally sub-blocks. But it cannot make sure the equally computation distribution for all processors. For the Mandelbrot Set algorithm, the back part will cost much more computation time than other parts. Figure 6 shows the row segmentation partition scheme with 4 processors. In terms of the figure, the rank 1 and rank 2 do most of the job, whereas the rank 0 and rank 3 do very few jobs. The unequal computation distribution increases the total computation time.

Also, Table 2 shows the computation and communication time (expressed in seconds) with 4 processors (i.e. for a single multicores computer with 4 logical processors) using row segmentation partition (Partition 1 in the table) and Fine-grain, static, row region data partition (Partition 2 in the table).

TABLE II

Communication and Computation time in two Partitions

	Rank0	Rank1	Rank2	Rank3	Total
Partition1	2.143	46.12	46.36	2.26	46.46
Partition2	23.61	23.59	23.61	23.97	23.99

The computation and communication time of each processor are recorded so that we can observe the computation time of each processor. Also, the total time (computation time plus communication time plus writing time) is recorded. By observing the table, the computation and communication time of rank 0 and rank 1 is very fast and they have to wait for rank 1 and rank 2 without doing any computation.

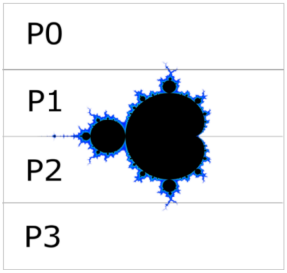


Fig .6 Row Segmentation Partition

Compared with partition 1, partition 2 has more balanced computation among processors, it decreases rapidly the total computation time [4].

There exists an even more balanced partition scheme namely dynamic row region data partitioning. It is modified based on the Fine-grain, static, row region data partitioning by randomized assign rows to processors, but there exists a problem that it is very hard to trace the row indices and write data sequentially into the file. And actually, this scheme almost does not decrease the total computation time.

After the comparison between these schemes, the Fine-grain, static, row region data partitioning is chosen as it is a balanced partition scheme and the implementation of it is easy.

V. RESULTS AND DISCUSSIONS

The Fine-grain, static, row region data partitioning is tested by running the programs on the hc* node from Monarch cluster. All tests in the experiment are running on the 24 core Xeon-E5-2680-v3 @ 2.50GHz servers. The parallelized computation part

is assigned to 2,4,6 and 8 processors respectively. And only the computation and communication time is recorded for the parallelized algorithm. All test case is run 5 times and the average computation time is also recorded. The results are recorded in Table B in Appendices.
The experimental speed up factors of the Mandelbrot computation using this partition scheme is computed using equation (7). The actual speed up is calculated by using the average computation time in Table B in Appendices. The results of actual Speed up factors are recorded in Table III.

$$S(p) = \frac{t_s}{t_p} \tag{7}$$

Where

t_s: execution time using one processor
t_p: execution time using multiple processors

In terms of the Table III, it can be observed that the actual speed up for p=2,4,6 and 8 is very close to the theoretical speed up calculated using Amdahl's law in Table I, which is S (2) =1.983, S (4) =3.8994, S (6) =5.7526, S (8) =7.5455.

TABLE III
ACTUAL SPEED UP WITHOUT IO

Partition Scheme	**Speed up factor, S(p)**			
	P=2	**P=4**	**P=6**	**P=8**
Row-grain	1.9637	3.8618	5.7451	7.4227

The actual speed up does not reach the theoretical speed up as there exists some communication time between processors, the resulted data in all processors except root node will be gathered by the root node, it will cost some time, but the communication time is not included in the theoretical speed up. Also, the actual computation and communication time can be influenced by the network speed as the results tested in a different time is quite different. Overly, the results of actual speed up have proved our parallelized algorithm using the Fine grain, static, row region partitioning scheme can get a near-linear speed up for multiple processors, as shown in Figure 7. The scalability analysis of the scheme is also needed. An important aspect of performance analysis is the study of how algorithm performance varies with parameters such as problem size, processor count [5]. In our problem, with the number of processors increases, the actual speed up is still a near-linear speed up. Thus, the speed up scalability of the algorithm is good. Besides, Table II in Section IV has proved our hypothesis, which is a balanced computation distribution can significant affect the total time spent.

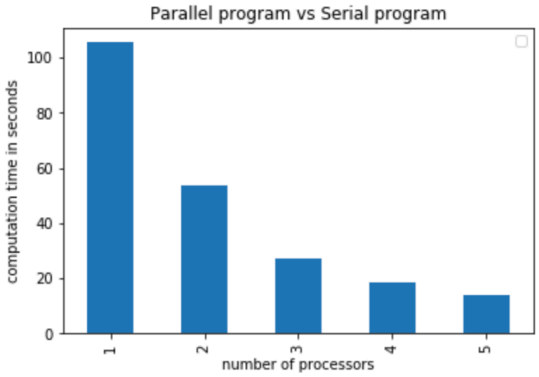


Figure .7 Partition scheme chosen vs Serial program

VI. CONCLUSIONS

The report describes the parallelized Mandelbrot Set algorithms using row segmentation partition scheme and Fine grain, static, row region partitioning scheme. The Fine grain, static, row region partitioning scheme is finally chosen to implement as it can achieve a balanced computation distribution for all processors, which means it has a better performance. The actual speed up is very close to the theoretical speed up when the number of processors p=2, 4, 6 and 8. The scalability of the scheme is also good as with the number of processors increases, the actual speed up is still a near-linear speed up. In conclusion, the balanced computation distribution is a very important factor for the Mandelbrot Set algorithm, which can decrease significantly the computation time. Besides, the network speed, communication time and different hardware can affect the total time spent. However, there exist limits in the experiment. (1) All the tests are only run 5 times. (2) The experiment is only run on hc* node

from monarch cluster. The future plan is to further test the scheme on different hardware and run it more times, getting a more accurate result.

REFERENCES

[1] "Mandelbrot Set Definition (Illustrated Mathematics Dictionary)", *Mathsisfun.com*, 2019. [Online]. Available: https://www.mathsisfun.com/definitions/mandelbrot-set.html. [Accessed: 06- Sep- 2019].

[2] "What are Fractals? – Fractal Foundation", *Fractalfoundation.org*, 2019. [Online]. Available: https://fractalfoundation.org/resources/what-are-fractals/. [Accessed: 06- Sep- 2019].

[3] "Design and Implementation of Parallel Matrix Multiplication Algorithms using Message Passing Interface", 2019. [Online]. Available: https://lms.monash.edu/pluginfile.php/9291353/mod_page/content/1/Parallel_Matrix_Multiplication_Report.pdf. [Accessed: 06- Sep- 2019].

[4] P. Kacsuk, "Embarrassingly Parallel Computations Computations Creating the Mandelbrot set", 2019. [Online]. Available: http://users.iit.uni-miskolc.hu/~szkovacs/LevParhRendszSeg/LevParhE1_2.pdf. [Accessed: 06- Sep- 2019].

[5] "3.4 Scalability Analysis", *Mcs.anl.gov*, 2019. [Online]. Available: https://www.mcs.anl.gov/~itf/dbpp/text/node30.html. [Accessed: 06- Sep- 2019].

APPENDIX

Table A (Serial Implementation)

***All the times are expressed in seconds*

Computer specifications	a) CPU: 24 core Xeon-E5-2680-v3 @ 2.50GHz servers b) Number of logical processors: 24 c) Memory size: 237624MB usable Memory d) Network speed: Gigabit Ethernet		
Value of iXmax	8,000 (default)		
Value of iYmax	8,000 (default)		
Value of IterationMax	2,000 (default)		
	Serial Program		
	Calculation & Communication	Write	Overall time (Calculation& Write)
Run #1	105.87	1.08	106.95
Run #2	105.65	0.77	106.42
Run #3	105.58	0.82	106.4
Run #4	105.29	0.99	106.28
Run #5	104.93	0.92	105.85
Average time	105.46	0.916	106.38

Table B (The Implementation of Fine grain, static, row region data partitioning vs Serial Implementation)

****only record the computation and communication time in the Table B.**

**** using same hardware in Tbale A**

	Serial program	Parallel Program			
		MPI			
		2 logical processors	4 logical processors	(e.g., 6 logical processors)	(e.g., 8 logical processors)
Run #1	105.87	54.791439	27.917122	18.623913	14.033621
Run #2	105.65	53.625294	26.868074	17.930181	14.495052
Run #3	105.58	53.655979	27.975194	18.670672	13.982424
Run #4	105.29	52.807125	26.931119	18.634785	14.037002
Run #5	104.93	53.632890	26.852602	17.923615	14.490029
Average time	105.46	53.70254	27.30882	18.35663	14.20762