Implementation and Analysis for parallelizing Mandelbrot Algorithm with Message Passing Interface

written by Takayuki Kimura

06/09/2019.

Monash University

Malaysia

tkim0004@student.monash.edu

Abstract—Mandelbrot set is widely known for its unique characteristics in computer science field and is one of the good algorithms to attempt parallelization. This report will describe 3 partition schemes of parallelizing the computational processes of Mandelbrot algorithm with the Message Passing Interface of C language. Bernstein condition is applied in order to prove if the algorithm is parallelizable or not. In this report, Row segmentation-based partition, Tile segmentation-based partition and Round Robin segmentation schemes are analysed and used to identify the differences of the 3 schemes and those performances in terms of the speed up in computation. The results of parallelized codes are compared to the serial code computation to represent the improvement of algorithm by parallelization.

I INTRODUCTION

In order to examine the effect of parallelization, we have to have a problem that requires huge amount of computations to be parallelized into smaller segments so that each processor can execute its task.

Mandelbrot set is a fractal structure and retrieved by quadratic recurrence equation. What is remarkable of Mandelbrot is its infinitely expanding structure with its equation (Dewey, 1996).

In this assignment, Mandelbrot is depicted in each pixels which is initially stored in color[3] array.

The data for the array is calculated IYMAX times * IXMAX on total. Thus, main computations that need to be parallelized is this calculation procedures. In this report, the parallelization of Mandelbrot is done using the Message Passing Interface (MPI) structure. The partition schemes applied in this assignment is 3, row segment, tile segment and round robin segment. In order to calculate the performance, Amdahl's law is useful to obtain theoretical performance based on the ratio of the serial code and parallel code with the selected number of processors (Cho et al., 2008).

II PRELIMINARY ANALYSIS

A. Parallelizability of mandelbrot set

Bernstein's conditions to prove If it properly parallelize the code or not.

Bernstein's condition is useful to verify that steps of computing individual data of Mandelbrot is parallelizable. In Bernstein condition, whether processes is not parallelizable or not is determined based on the three conditions as shown below.

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

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

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

I_0 and O_0 is the input and output for the first process while I_1 and O_1 is the input and output for the second process. In order to meet the Bernstein condition, both processes are not dependent to each other and the algorithm is proven as it is parallelizable. (1) and (2) are the sequences of the calculation for mandelbrot set per pixel.

In the given serial code, there are two arguments for the input, iY and iX used to calculate the output, Zx2 and Zy2.

I_0 = i for iY and **j** for iX

I_1 = n for iY and **m** for iX

O_0 = $Zx2_0$ and $Zy2_0$;

Where

$Cx_0 = CxMin + (j * PixelWidth);$

$Cy_0 = CyMin + (i * PixelHeight);$

$Zx = 0.0;$

$Zy = 0.0;$

$Zx2 = Zx * Zx;$

$Zy2 = Zy * Zy;$

After that, using Cx_0 and Cy_0,

$Zy_0 = (2 * Zx * Zy) + Cy_0;$

$Zx_0 = Zx2 - Zy2 + Cx_0;$

$Zx2_0 = Zx_0 * Zx_0$

$Zy2_0 = Zy_0 * Zy_0$

(1)

On the other hand,

O_1 = $Zx2_1$ and $Zy2_1$;

Where

$Cx_1 = CxMin + (m * PixelWidth);$

$Cy_1 = CyMin + (n * PixelHeight);$

$Zx = 0.0;$

$Zy = 0.0;$

$Zx2 = Zx * Zx;$

$Zy2 = Zy * Zy;$

After that, using Cx_1 and Cy_1,

$Zy_1 = (2 * Zx * Zy) + Cy_1;$

$Zx_1 = Zx2 - Zy2 + Cx_1;$

$Zx2_1 = Zx_1 * Zx_1$

$Zy2_1 = Zy_1 * Zy_1$

(2)

By applying Bernstein's condition, three conditions are

i and j $\cap$ $Zx2_1$ and $Zy2_1$ = Φ

n and m $\cap$ $Zx2_0$ and $Zy2_0$ = Φ

i and j $\cap$ n and m = Φ

Hence, the two processes are parallelizable.

B. Theoretical speed up analysis for a Mandelbrot set

A serial code of Mandelbrot computation is used to calculate the theoretical speed up. The serial code is composed of three sections. Obtaining the iY and iX for the calculation per pixels, execute Mandelbrot computation and writing the contents into the file. The overall

time taken to compute whole processes and computational time taken by the only computation are recorded in the table.

Amdahl's law (4) is applied to obtain the theoretical speed up of the serial code.

$$S(p) = \frac{1}{r_s + \frac{r_p}{p}} \quad (3)$$

Where

r_p = parallel ratio

r_s = serial ratio

p = number of the processors

The resulted theoretical speed up is represented in the table A where the number of processor is 4 and 8 (p=4, 8).

TABLE A

THEORETICAL SPEED UP FACTOR BY AMDAHL'S LAW

The number of processors (p)	4	8
Speed Up Factor s(p)	3.988	7.944

In this serial code, the part which writes the values into file is considered as the r_s. Other parts such as calculating Mandelbrot set is parallelizable in each processors.

III PARTITION SCHEMES FOR DATA PARALLELISM

For all implementations, row, tile and round robin schemes, the code begin with setting array and variables to be used in calculation part. The array is 1 dimensional array which will stores all results of Mandelbrot set computation per processors. Thus, root node write the values into the file based on this array.

A. Row segmentation partitioning scheme.

In row segmentation partitioning scheme, each processor will take the assigned screen which is divided based on Y axis (Fig. 1). The root node plays a role of receiving the resulted values sent from other processors and writes it to the file. In the serial code, the computation part is composed of outer loop for the range of Y axis of the screen and inner loop for the range of X axis of the screen. In order to assign the parts to be processed to each processor, the start position and end position of the outer loop is divided by the number of the processors. Hence, the screen that processor computes is calculated by the size of the screen on Y axis / the number of processors * the size of the screen on X axis (5). Figure 2 visualizes the brief flow of the row segmentation partitioning scheme.

$$Screen\ per\ processor = \frac{The\ size\ of\ Y\ axis}{The\ number\ of\ processors} \times the\ size\ of\ X\ axis \quad (4)$$

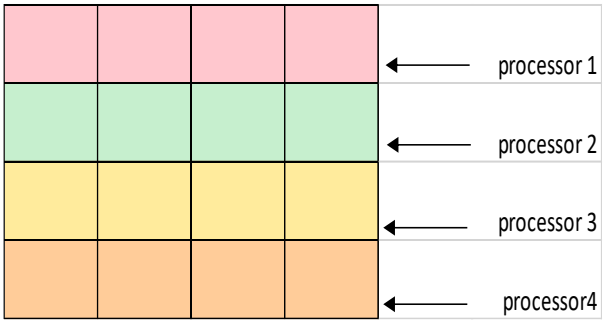

Fig 1 Row segmentation partitioning scheme

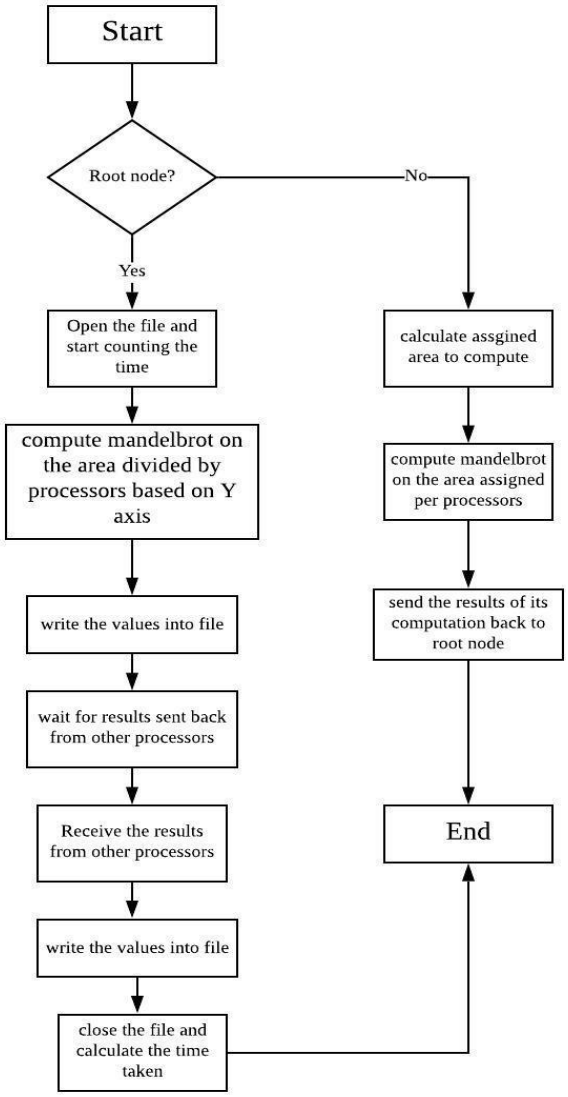

Fig. 2 Flowchart of Row segment partitioning scheme

B. Tile segmentation partitioning scheme.

In Tile segmentation portioning scheme, the computation for each processor is distributed based on the screen which is divided by Y axis and X axis. This means that the segment is divided into sub segments (Fig. 3) obtained by the screen of Y axis per processor × the screen of X axis per processor (equation 5, 6).

$$The\ Y\ screen\ size\ per\ processor = \frac{The\ size\ of\ screen\ on\ Y\ axis}{2} \quad (5)$$

$$The\ X\ screen\ size\ per\ processor = The\ size\ of\ screen\ on\ X\ \frac{axis}{the\ number\ of\ processors} \quad (6)$$

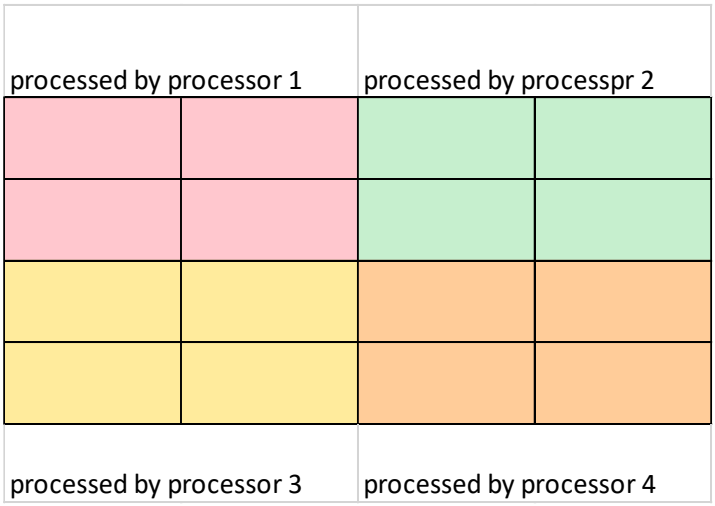

Fig. 3 Tile segmentation partitioning scheme

Tile segmentation partitioning works with the even number of the processors as it makes the even number of sub segments. However, when the number of the processors is odd, it creates 1 extra sub segment which none of processor is assigned. In this situation, root node does its computation again (Fig. 4) and it slows the computation time down as it is not designed to deal with the odd processors.

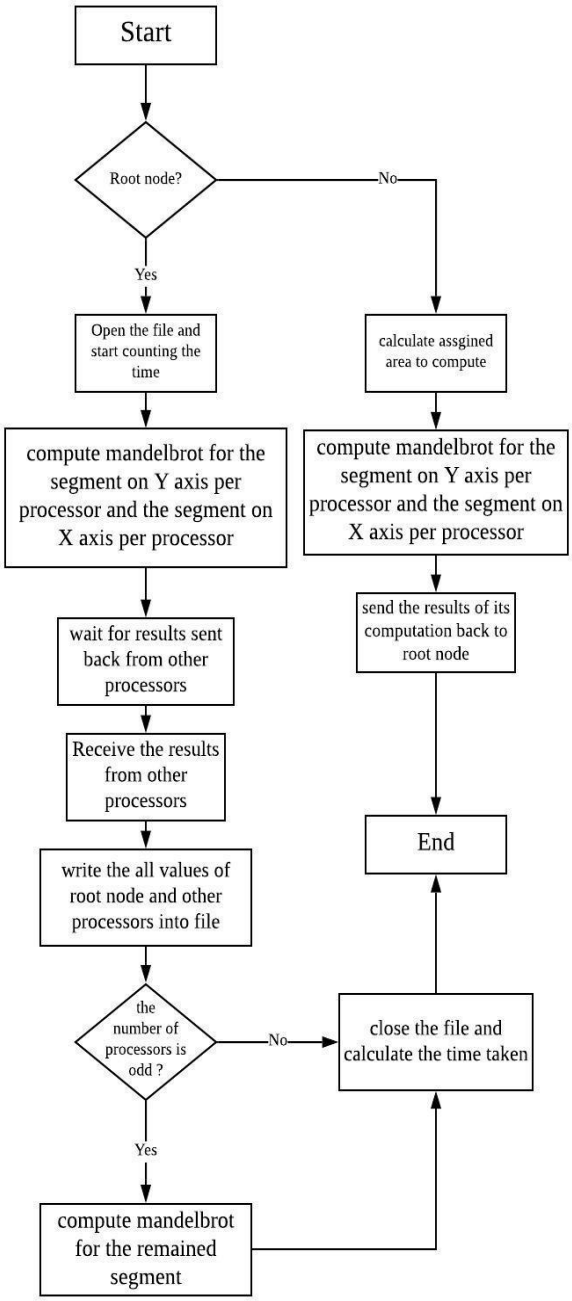

Fig. 4 Flowchart of Tile segmentation partitioning scheme

C. Round robin segmentation partitioning scheme.

In the Round robin segmentation partitioning, the processors are swapping the portion to process row by row (Fig. 5). The portion that each processor processes is around same with the row segmentation, however, every time the row of the screen is calculated it jumps to the row which is the number of the processors far from the previous row. And it repeats this process until it exceeds the size of the screen on Y axis. The brief steps are illustrated in the flowchart below (Fig. 6).

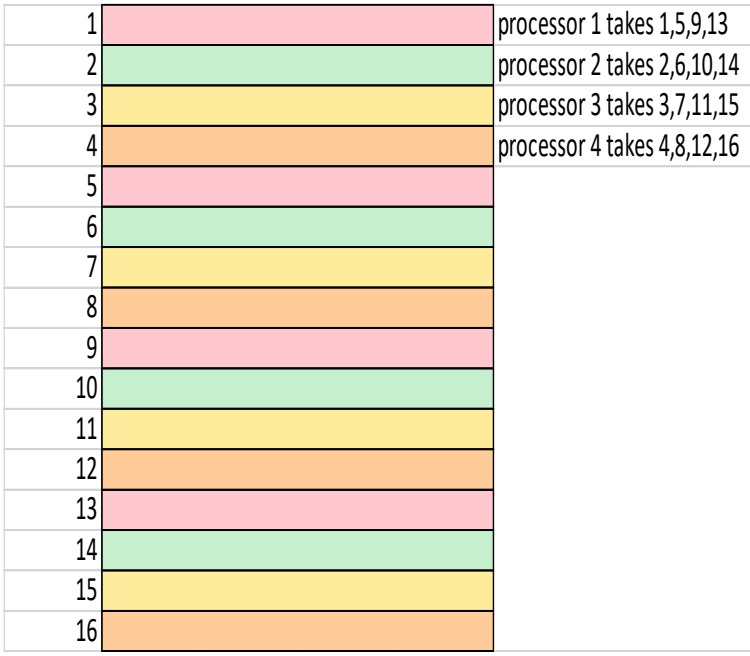

Fig.5 Round robin segmentation partitioning scheme

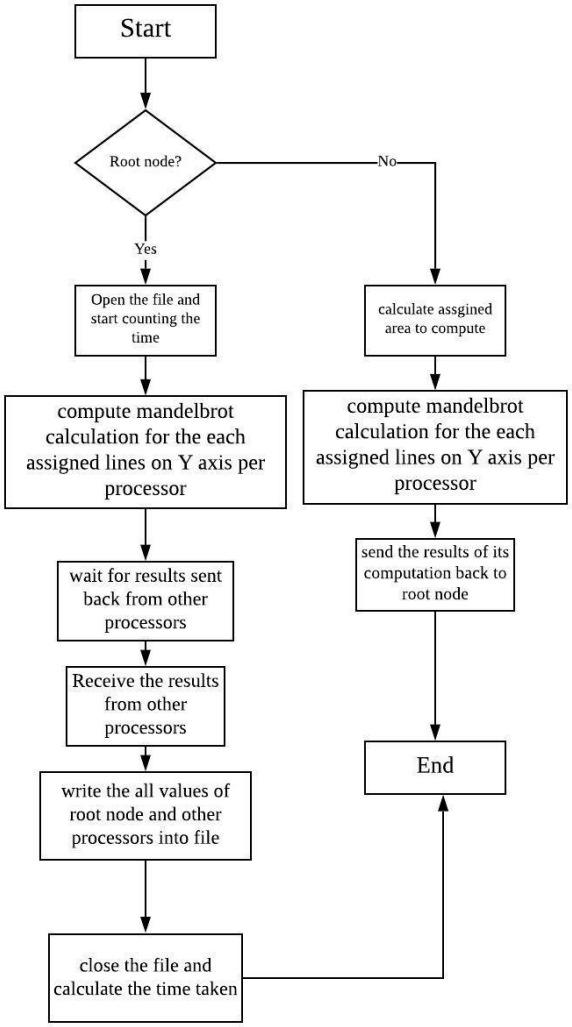

Fig. 6 Round robin segment partitioning scheme

IV IMPLEMENTATION OF PARTITIONING SCHEMES

In implementing the Mandelbrot set, the size of the screen is set based on the variables in the code, iYmax for Y axis and IXmax for X axis respectively. In the provided code, both are set 8000. The parallelization on Mandelbrot set for 3 partition schemes mentioned in previous section is implemented in C language and the code is run using the desktop PC of the data science lab in Monash university Malaysia. The actual speed up factor (Fig. 7) is obtained for all partition schemes with the number of the processors is 4 and 8.

	Speed Up Factor s(p)	
The number of processors (p)	4	8
Row segment partioning	2.007	2.447
Tile segment partitioning	2.707	2.783
Round Robin partitioning	3.858	7.615

Fig. 7 Actual speed up factors

V RESULTS AND DISCUSSIONS

The speed up factors for the partition schemes are calculated based on the equation (7) which is obtained from Table A, B and C.

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

t_s = serial code computational time

t_p =parallelized code computational time

(7)

As the line graph below (Fig. 8), the speed up is not linearly increasing as it indicates fluctuation of the values, especially when the number of the processors is odd number. This phenomena can be seen in the tile segment partitioning speed up as well as shown in a graph (Fig. 9). Aside to that, the speed up value increased when the number of processors is 8.

On the other hand, round robin segmentation indicates the perfect linear speed up throughout the processors (Fig. 10) in contrast to row partitioning and tile partitioning. This is because one processor takes extra computations for the portions of the remainders when the screen has the remainder in row and tile segmentation. For example, if the screen size called iYmax is 8000 and the number of processor is 3, there is the parts that none of processors is assigned to because of the remainder, 8000 % 3 = 2. In my

implementation, this portion of 2 is assigned to the root node and it starts the computations. This slows the speed up down when this remainder portion has the heaviest calculation time. However, round robin partitioning breaks the screens into quite small pieces of segments and distribute them to each processor and this segmentation is not critically affected by whether there is remainder or not because each node still deal with the remainder's part instead of letting one processor do all computations for the remainder part, which is the root node in row and tile partitioning. Thus, round robin partition scheme would be the best scheme of other 2 schemes in Mandelbrot set problem since it solves the remainder problem distributing the segments almost equally even if there is a remainder.

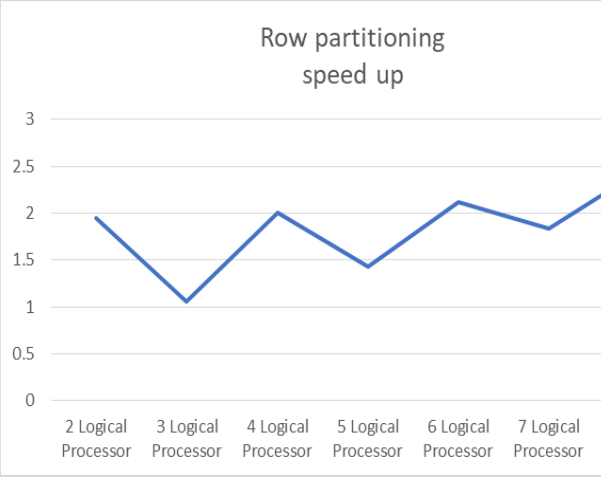

Fig. 8 row segmentation speed up Line graph

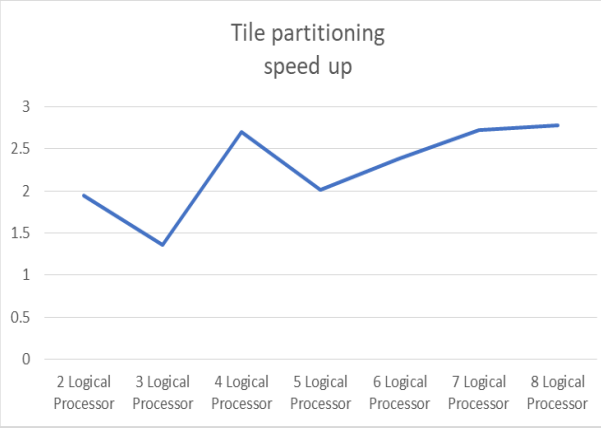

Fig. 9 Tile segmentation time Line graph

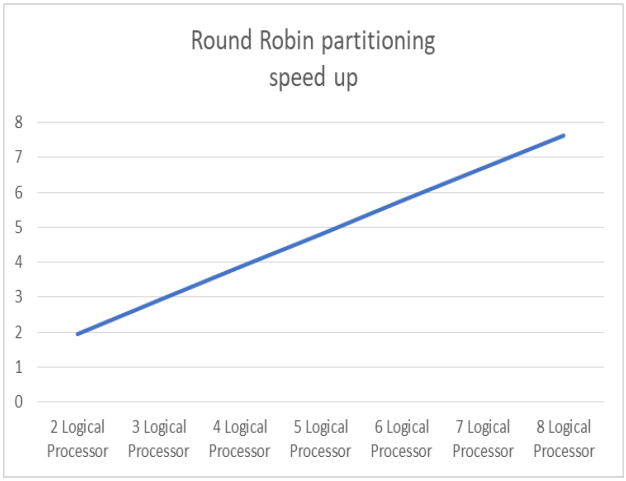

Fig. 10 Round robin segmentation time Line graph

VI CONCLUSIONS

This document discuss the parallelization of Mandelbrot set computation with Message Passing Interface implementing 3 partition schemes, row, tile and round robin. The analysis and comparisons are done presenting theoretical and actual speed up values and it discovered that round robin segment partition scheme indicated higher performance than other two schemes though tile segmentation is still better than row segmentation. The main difference between round robin and other schemes is whether it can distribute the segments well or not as the root node should not take all calculations caused by the extra portions of the remainders. In conclusion, the parallelized code dramatically improved its performance as compared to the serial code, however, it is essential to focus on how to process the remainder parts to have speed up stable throughout any processors.

References

[1] Cho, S., & Melhem, R. (2008). Corollaries to Amdahl's law for energy. *IEEE Computer Architecture Letters*, 7(1), 25-28.

[2] Dewey, D. (1996, 11 02). Introduction to the Mandelbrot Set. Retrieved from http://www.ddewey.net/mandelbrot/

Table A - Row segment partitioning scheme

	Serial Program	Serial Program	Parallel Program						
			MPI						
Row segmentation	Coumputational time	overall time	2 Logical Processor	3 Logical Processor	4 Logical Processor	5 Logical Processor	6 Logical Processor	7 Logical Processor	8 Logical Processor
Run 1	67.682621	67.682851	34.77944	63.967624	33.802681	47.303128	32.110641	36.931758	27.770602
Run 2	68.01266	68.050693	34.764626	64.224585	33.835085	47.392793	32.164981	36.921796	27.717869
Run 3	67.842147	67.891763	34.728517	64.343609	33.770663	47.268442	32.132994	36.942525	27.796473
Run 4	67.661947	67.71256	34.700402	64.044682	33.830799	47.464926	32.123382	36.933267	27.726783
Run 5	68.280702	68.317966	34.87198	63.975776	33.842146	47.283424	32.126891	36.938147	27.762473
Average Time	67.8960154	67.9311666	34.768993	64.1112552	33.8162748	47.3425426	32.1317778	36.9334986	27.75484

Table B – Tile segment partitioning scheme

	Serial Program	Serial Program	Parallel Program						
			MPI						
tile segmentation	Coumputational time	overall time	2 Logical Processor	3 Logical Processor	4 Logical Processor	5 Logical Processor	6 Logical Processor	7 Logical Processor	8 Logical Processor
Run 1	67.682621	67.682851	34.944723	49.730701	25.079723	33.823872	28.349992	24.957463	24.385272
Run 2	68.01266	68.050693	34.916257	49.670089	25.094516	33.795654	28.398647	24.95126	24.362817
Run 3	67.842147	67.891763	34.936036	49.990068	25.109386	33.589812	28.457978	24.949103	24.415307
Run 4	67.661947	67.71256	34.970607	49.962074	25.144722	33.595055	28.48339	24.949478	24.406397
Run 5	68.280702	68.317966	34.818217	49.994488	25.021417	33.573259	28.396588	24.967864	24.381538
Average Time	67.8960154	67.9311666	34.917168	49.869484	25.0899528	33.6755304	28.417319	24.9550336	24.3902662

Table C- Round robin partitioning scheme

	Serial Program	Serial Program	Parallel Program						
			MPI						
Round robbin	Coumputational time	overall time	2 Logical Processor	3 Logical Processor	4 Logical Processor	5 Logical Processor	6 Logical Processor	7 Logical Processor	8 Logical Processor
Run 1	67.682621	67.682851	34.819776	23.414091	17.585439	14.093392	11.769784	10.135357	9.001582
Run 2	68.01266	68.050693	34.90892	23.398597	17.618723	14.116946	11.878614	10.12086	8.910035
Run 3	67.842147	67.891763	34.893562	23.386516	17.598819	14.096983	11.772201	10.169011	8.883332
Run 4	67.661947	67.71256	34.872142	23.392001	17.579904	14.091178	11.791166	10.190211	8.891405
Run 5	68.280702	68.317966	34.7882	23.3887722	17.592146	14.123953	11.771351	10.141859	8.891656
Average Time	67.8960154	67.9311666	34.85652	23.39599544	17.5950062	14.1044904	11.7966232	10.1514596	8.915602