INDIAN INSTITUTE OF TECHNOLOGY GANDHINAGAR

ES 611: ALGORTITHMS ON ADVANCED COMPUTER ARCHITECTURE

Report on Computational Project 4: Interfacing MPI and CUDA along with concurrent parallel constructs including OpenMP and Pthreads for Image Processing, Linear Algebra.

By Group 4:
Nishant Rao (11110059)
Parth Gudhka (11110062)
Ravi Kumar (11110081)
Shashank Heda (11110096)

ABSTRACT

High performance computing is based upon the principles of work load distribution and code parallelization. The first case of the CP 1.2 assignment deals with a similar issue, explaining the need of high performance computation in the field of scientific research.

OVERVIEW

1. Implementation of Julia Set images through OpenMP construct
2. Communication Strategies for Message Passing
3. Using Topologies and Communicators in Matrix-Vector Multiplication
4. Parallelizing the Matrix-Matrix multiplication using the Fox's Algorithm

Case 1: Implementation of Julia Set images through OpenMP Construct

INTRODUCTION

Julia set is an image based graphical approach to visualize a mathematical function, inequality or an identity, developed by Mandel Brot. Based upon the calculation of an arbitrary point whose locus is of the prime interest, each pixel is assigned a color value which is stored in the form of a matrix and is displayed through specific X11 programming. Here is a brief analysis of a parallelized Julia Set implemented using the OpenMP constructs. This report gives a wide glance of the intricacies associated with the shared memory architecture and the parallelization of mutually exclusive pixel relationship in case of Julia Set.

METHOD OF APPROACH

- For the construction of the prescribed image using Julia Set, the present case is such that the calculation and plot of each pixel is independent of the neighboring pixels as well as any other pixel in the entire image.

- This offers a scope of massive parallelization as quite a less amount of code constitutes a critical section, especially in case of the shared memory construct.

- The ‘for’ loop responsible for the computation of the color of each pixel is parallelized among varying number of threads depending with an aim to arrive at the optimum degree of parallelization for the given problem.

- Due to no control over the individual threads as well as on their behavior, it is perhaps not advisable to parallelize the X11 coding and the window display region in case of shared memory architecture. Possibility of deadlock and feasibility of erroneous outcomes should be foreseen before parallelizing the display commands among the slave threads.

- Also, the presence of entire X11 programming which involves several steps of creating the display window and plotting the two dimensional region makes the corresponding code parallelization in the MPI construct slightly complex in terms of distributing the display window parameters among various processors.

RESULTS AND DISCUSSIONS

- Initially, the image is parallelized among varying number of threads keeping the values of imax and complex variable c constant. Figure 4.1(a) shows the graph of comparison of total time, the image plotting time and the computation time taken by the process.

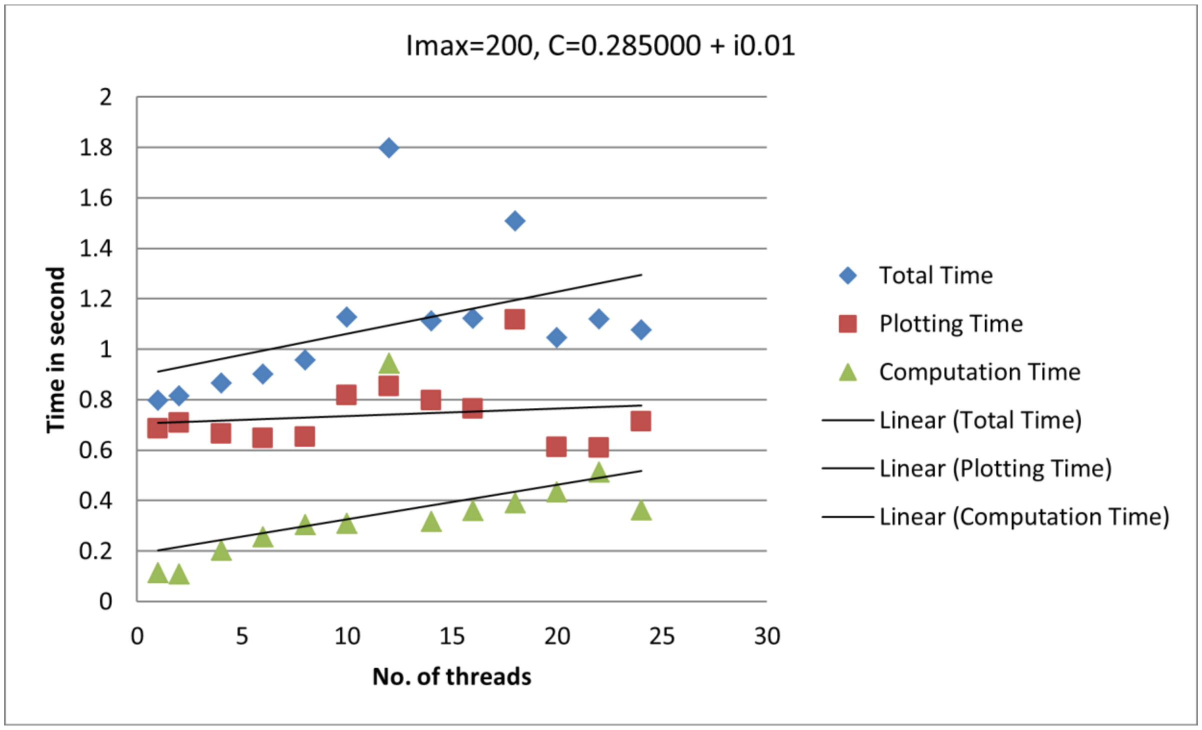


Figure 4.1(a): Time taken vs. Number of threads for mentioned value of Imax, c.

- As observed from figure 4.1(a), the plotting time is dominant for almost all the data points as compared to its computation time. This precisely indicates the effect of parallelizing the computation, keeping the control of the plotting process just under the master thread.
- Hence, the Thread-0 (master thread) takes its time to display the entire image in addition to contributing in the computation process.
- However, an important phenomenon visible here is that the total time is increasing with increase in number of threads for the problem. Most probably, it seems to happen so because of the display unit being entirely executed after the pixel value is calculated by each thread.
- The plotting process of the pixel region was parallelized at one stage but the problem it generated was of the multiple threads modifying the loop variables (i, j) as and when they reach their local value: that is suppose Thread-1 initially executes for i=1 and j=1, the process during which some other thread, say Thread-2 executes and modifies the i=3, j=4, in such case, the subsequent value that the Thread-1 will be left with some spurious value that follows high probability of generating a non-desired outcome.
- Hence, parallelizing the X11 programming is perhaps not desirable and advisable for this case. It results in a waiting condition for all the threads: all the threads compute their code within a short time but need to wait for the master thread to complete its job and then finally display the image. Also, printing the image is dependent highly upon the internet connection speed 0.156 between the local client and the HPC server.
- Hence, the plotting time and therefore the total time keeps on increasing with the increase in number of threads.
- Figure 4.1(b) shows the time taken for mentioned Imax, c.

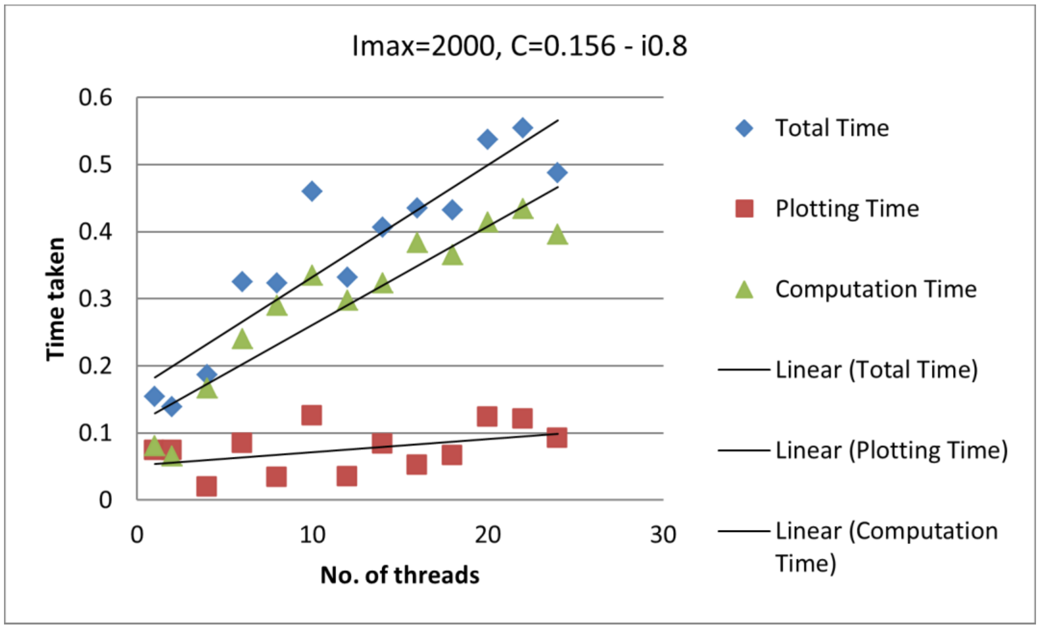


Figure 4.1(b): Time taken for execution varying the number of the threads

- Figure 4.1(b) indicates the trend of all the three time intervals associated with the given value of Imax, c. It should be noted that in this case, magnitude of c is almost 10 times greater than that in figure 4.1(a); due to this, the loop to be executed inside the calculation of "TheK" variable (that stores the color value of a pixel) is executed less number of times.
- This results in a fair speed due to reduced number of loop iterations because higher the magnitude of c, faster the iterative method will converge to a value in time lesser than the previous case. Here, maximum time taken in seconds goes upto 0.6 seconds whereas that in the previous case is almost three times higher than the later one.
- Figure 4.1(c) is a plot of time taken by the code against the number of threads executing the code.

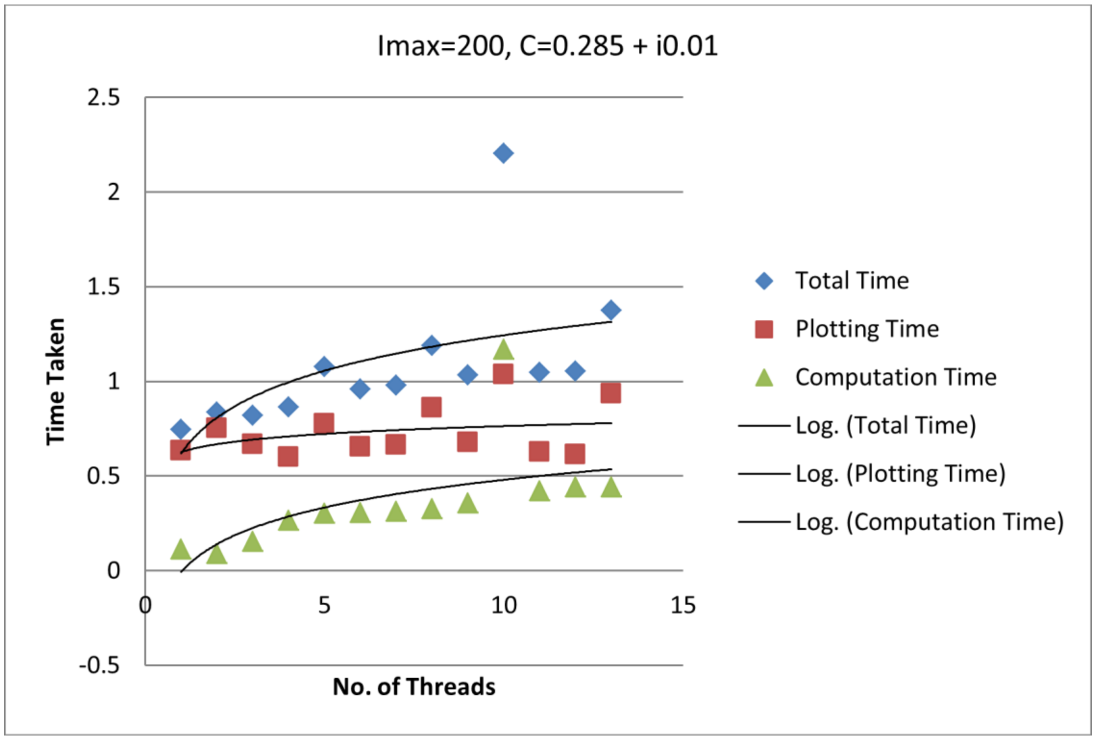


Figure 4.1(c): Time taken vs. no. of threads executing the

- The chief objective that figure 4.1(c) stands for is the linear variation of timings including the total, plotting and the computation timings with respect to the number of threads, for a given value of imax and c. Moreover, the graph also contains the kinky nature of the data points at some instants which is most probably a result of out-lying points.
- Especially, in case of 12 threads, the out-lying behavior is clearly visible. This seems to happen for 12 threads because the division by 12 makes the calculation improper for accurate load division and hence, the timing changes abruptly.
- However, the trend line depicts the computation time to become saturated or constant after certain optimum number of threads whereas, the plotting time too exhibits the saturation or perhaps a decrement after certain number of threads which at present stage cannot be afforded by the current HPC cluster. It also claims a possibility of an overall decrement in time taken by the code on increasing

the number of threads provided the number of threads is increased to the maximum limit of the HPC provision.

- Besides, based upon parallel inputs from the other groups, it should be noted that the OpenMP construct generally gives abnormal results while operating upon the image processing problem.
- A couple of basic queries and in fact, an area of future exploration include:
    - Why is the time in every graph appearing to increase with an increment in the number of threads for the given values of imax and c?
    - There has been almost an equal number of plots amongst whom some of them show the plotting time to be dominating over the computation time while some others seems to depict the converse phenomena. How does it justify the parallelism?
- Figure 4.1(d) is about the variation in time with respect to the magnitude of c, for a given fixed value of imax.

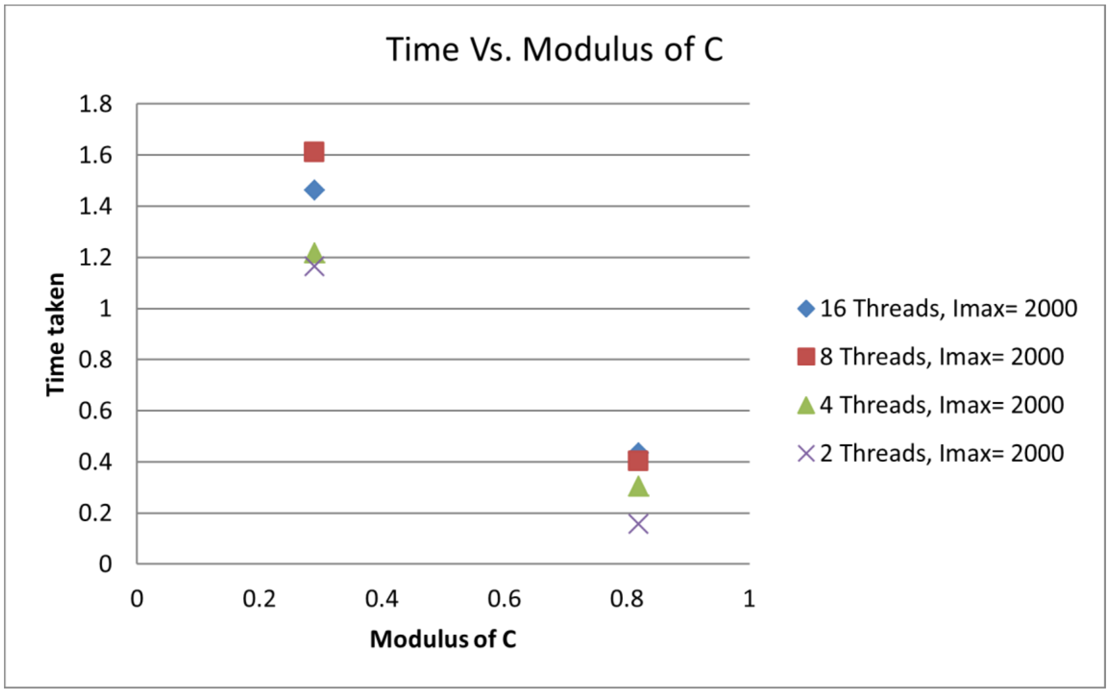


Figure 4.1(d): Time taken vs. modulus of c for a given number of threads and

- Here, the time taken by the code decreases with increment in the value of c, but is increasing for an increment in number of threads.
- It can be seen that the decrement in time with respect to the value of c is quite expected since this value contributes to the number of iterations that the code is bound to execute to arrive at the desired z-value.
- Looking at the other side, even though the time taken by increasing number of threads is increasing, however, the gap between this time-interval seems to decrease as we increase the threads. One of the probable reasons or the execution being slowed by incrementing threads can be the amount of overheads which are dependent upon the creation of threads if causes more time of generation as compared to the overall computation or overall plotting time, this result may be obtained.

- Provided more number of threads, it can be actually tested and verified whether the timings seem to increase with increment in number of threads or does it start decreasing after certain number of threads, or does it move towards saturation for constant increment in number of threads.

MODIFICATION TO ZOOM AND SHIFT THE IMAGE

It is quite simple to zoom in and zoom out the Julia Set image by modifying the value of z. The increment in value of z-imaginary part zooms in the image while a change in the real part of the z value shifts the image accordingly.

Figure 4.2(a), 4.2(b) and 4.2(c) explains this feature:

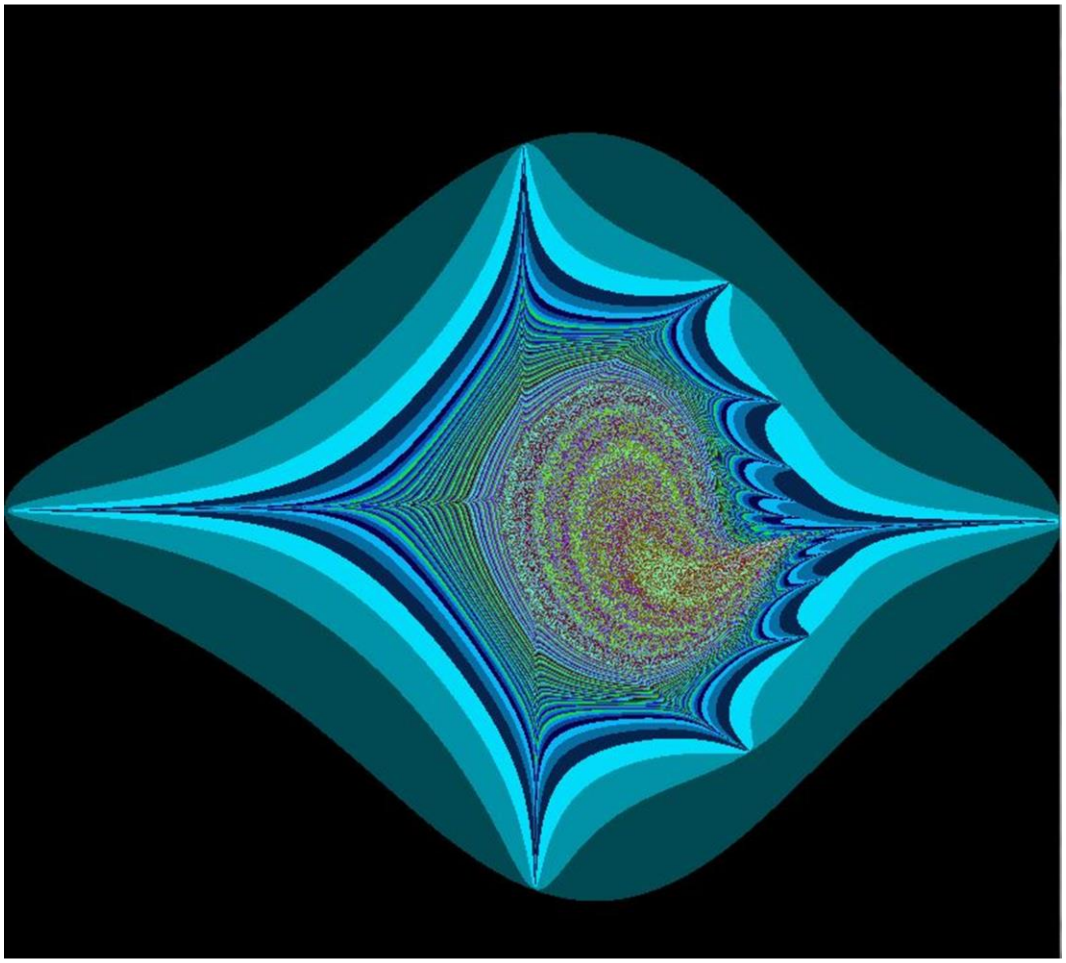


Figure 4.2(a): Original image

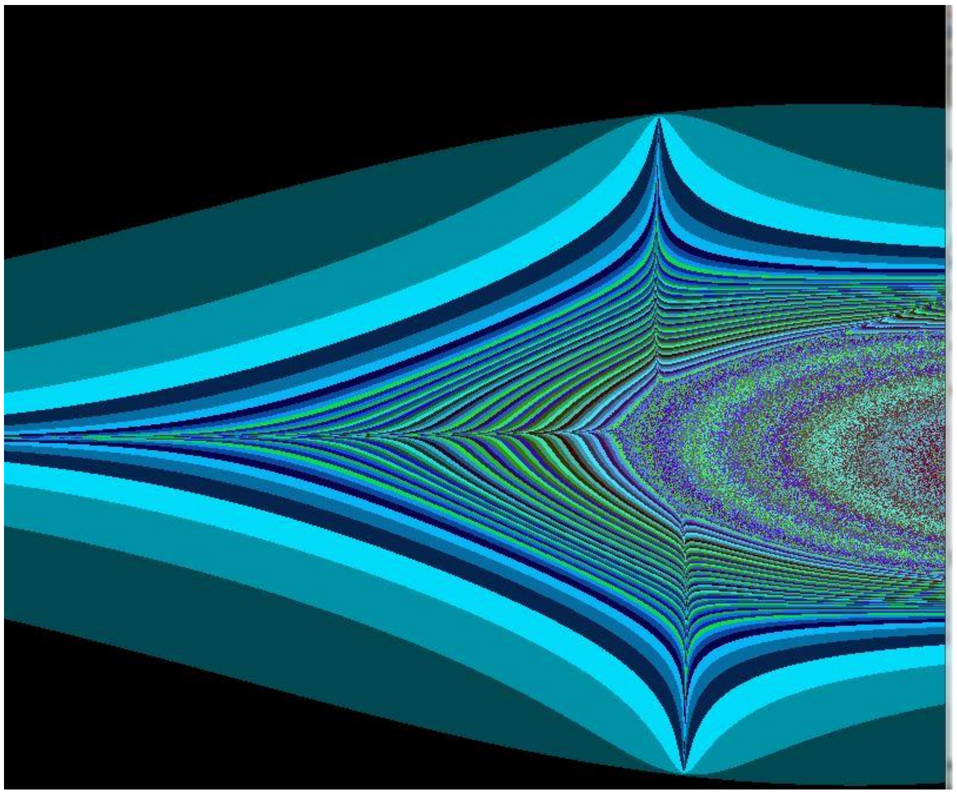


Figure 4.2(b): Zoomed Image

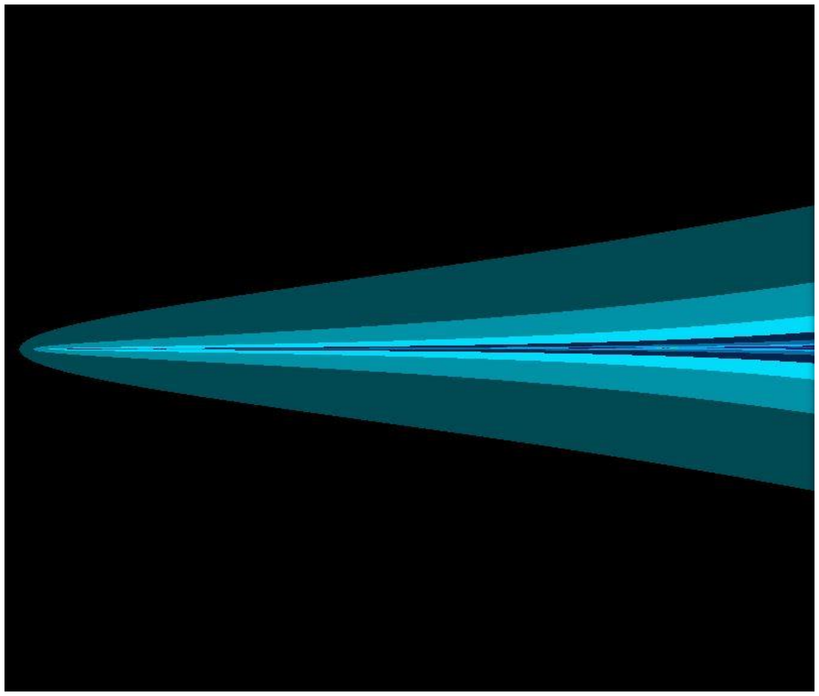


Figure 4.2(c): Shifted + Zoomed Image

CONCLUSION

Based on all the above graphs and results discussed in this report, it can be said that the OpenMP construct perhaps isn’t an appropriate choice for parallelizing the Julia set image formation. It also involves the serial functioning of X11 code which cannot be parallelized in case of OpenMP construct.

2. Solution of Sparse Linear systems of the form A x =B using Direct Methods

Introduction

A physical process, governed by a continuous equation of the form, Lx = f, where L is an operator, can be solved using different methods for finding the value of unknown function x using computational methods based on *Thomas Algorithm*, *Cyclic Reduction (Odd-even Reduction)* and parallelising such available computational tools would result in enhancement of the computational performance.

Method of Approach

The given Thomas algorithm was studied and executed for different values of m and methods by which it can be parallelized using OpenMPI, OpenMP, Pthreads on multi-core computers and CUDA were discussed. A more efficient algorithm, Cyclic Reduction Method, which has intrinsic parallelism is then discussed and parallelised using MPI and OpenMP. The results obtained were checked with the theoretical formulae given in part (b) and part (c).

Results and Discussion

The serial code that used Thomas Algorithm for solving the given set of equations with unknowns for a tri-diagonal matrix was executed for different values of m, and the results about the variation of time taken with the order of matrix, time proportion spent in Memory allocation, Initialisation, Computation, Back Substitution and Freeing the occupied Memory are discussed.

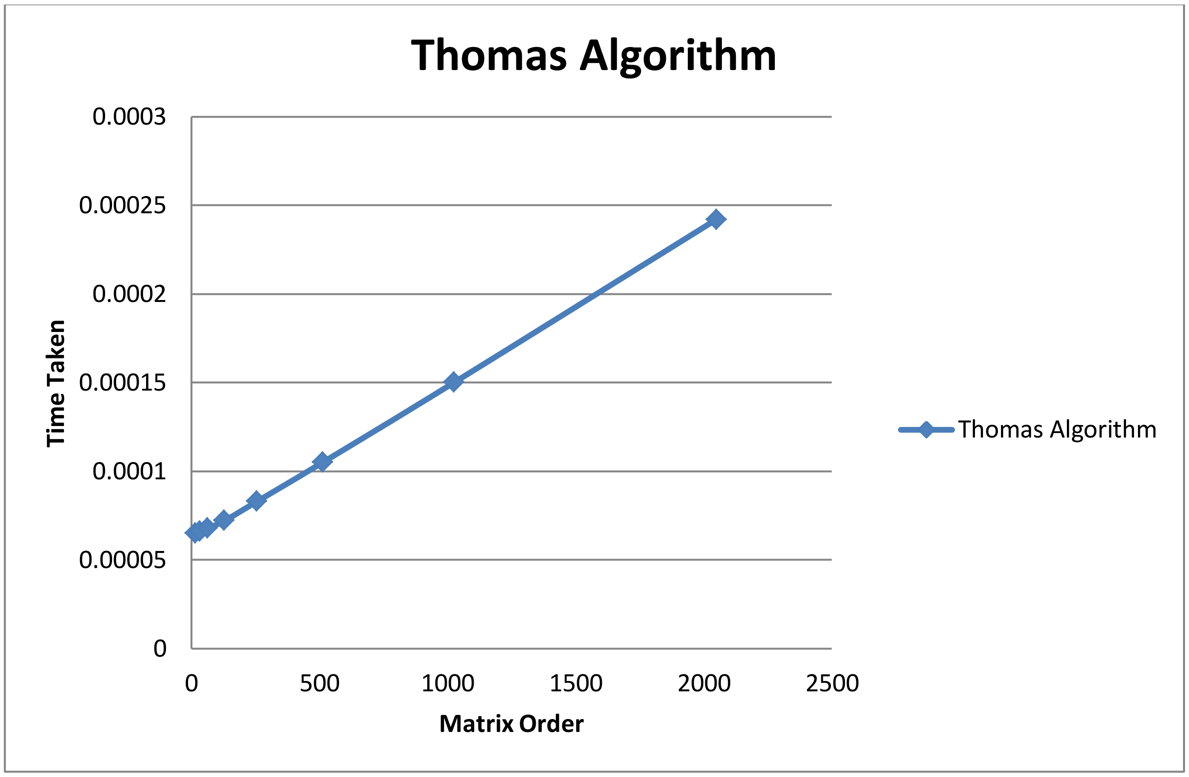


Fig 2.1 Variation of Time taken with change in matrix order for Serial Thomas Algorithm

- It may be noticed from Fig. 2.1 that as the matrix order m is increased, i.e., as number of unknowns in the matrix m x m is increased, the time taken for solving a serial code based on solving a Tri-diagonal matrix (using Thomas Algorithm) increased linearly. It is because as the order of matrix increases, the steps needed to find the exact solution increase linearly as can be seen in above graph.

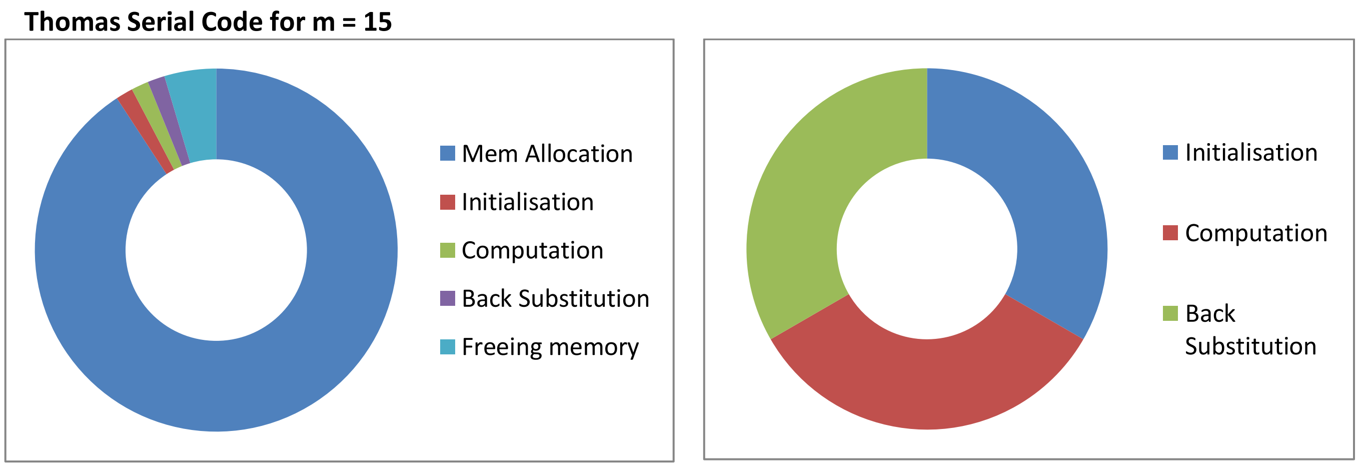


Fig 2.2 (a) and (b) depicting the time distribution for Thomas Algorithm

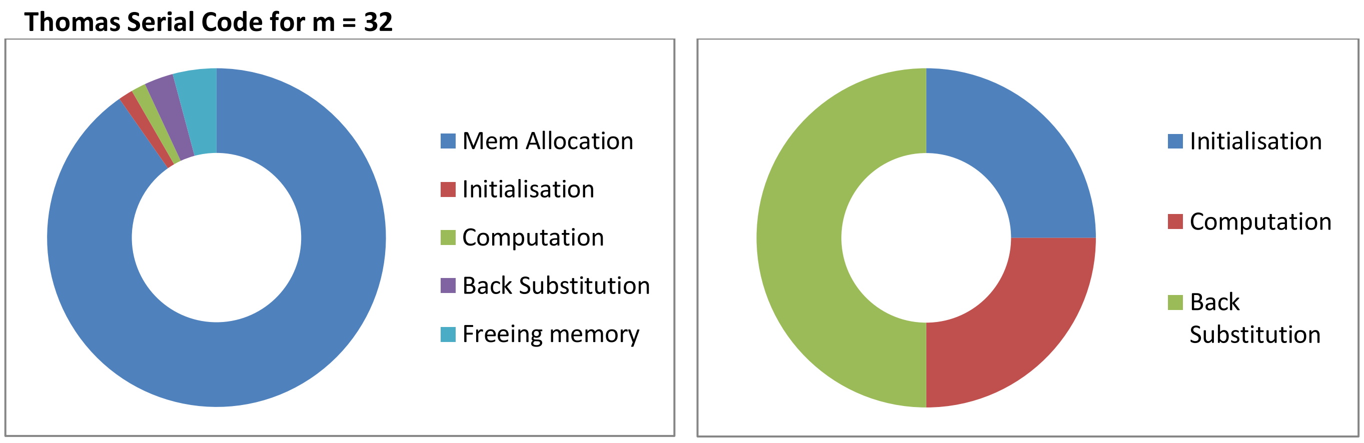


Fig 2.3 (a) and (b) depicting the time distribution for Thomas Algorithm

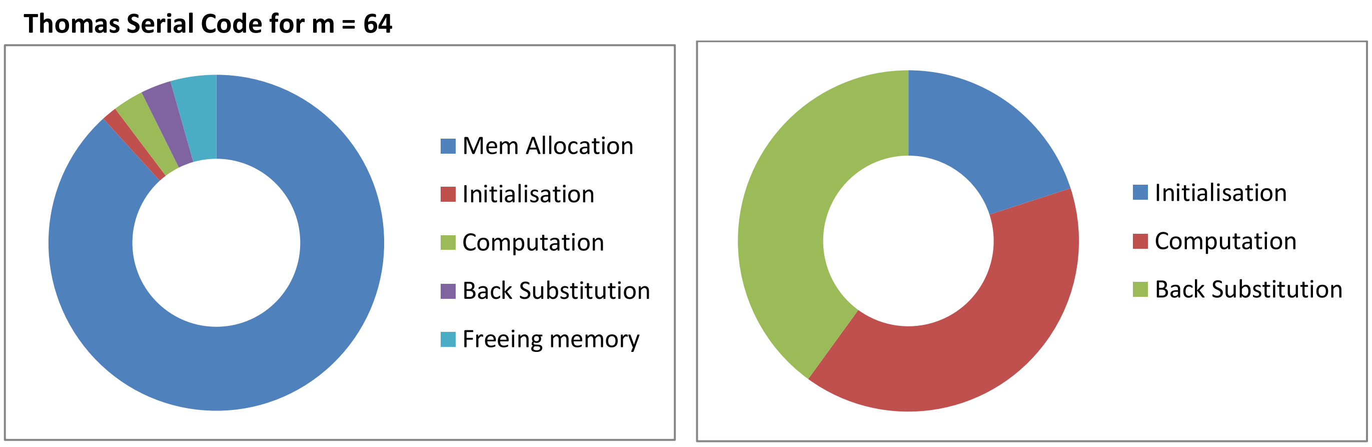


Fig 2.4 (a) and (b) depicting the time distribution for Thomas Algorithm

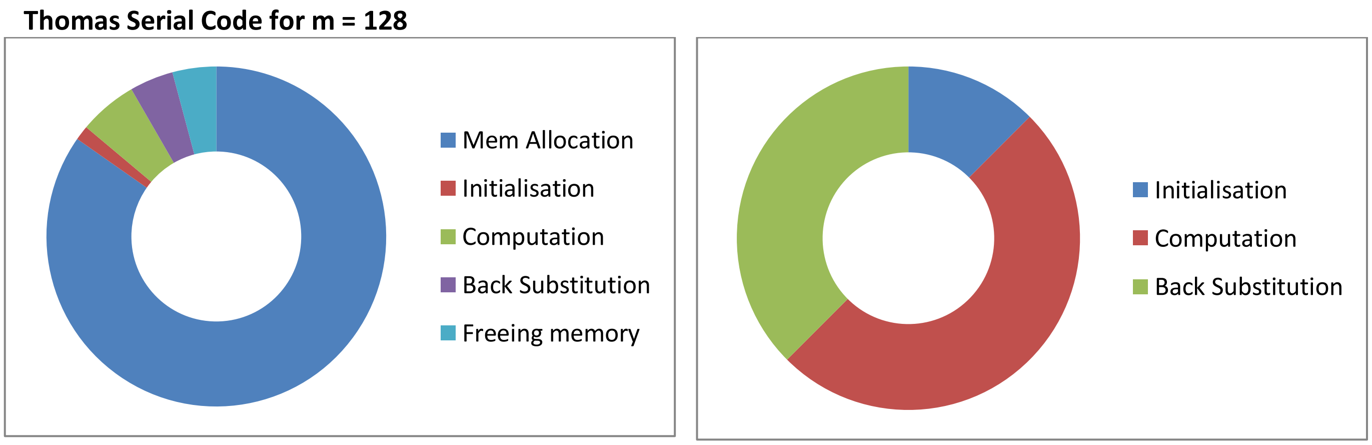


Fig 2.5 (a) and (b) depicting the time distribution for Thomas Algorithm

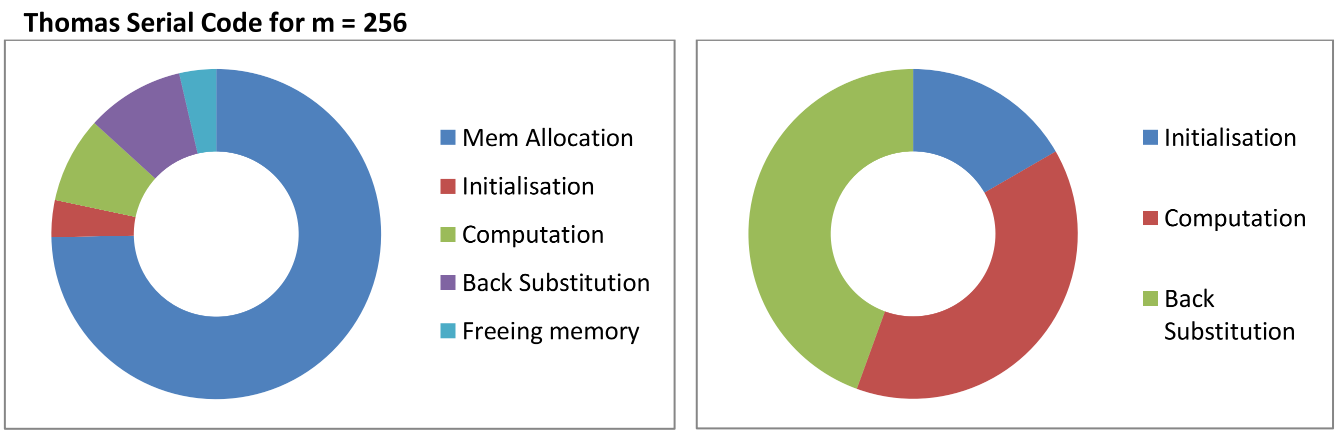


Fig 2.6 (a) and (b) depicting the time distribution for Thomas Algorithm

Thomas Serial Code for m = 512

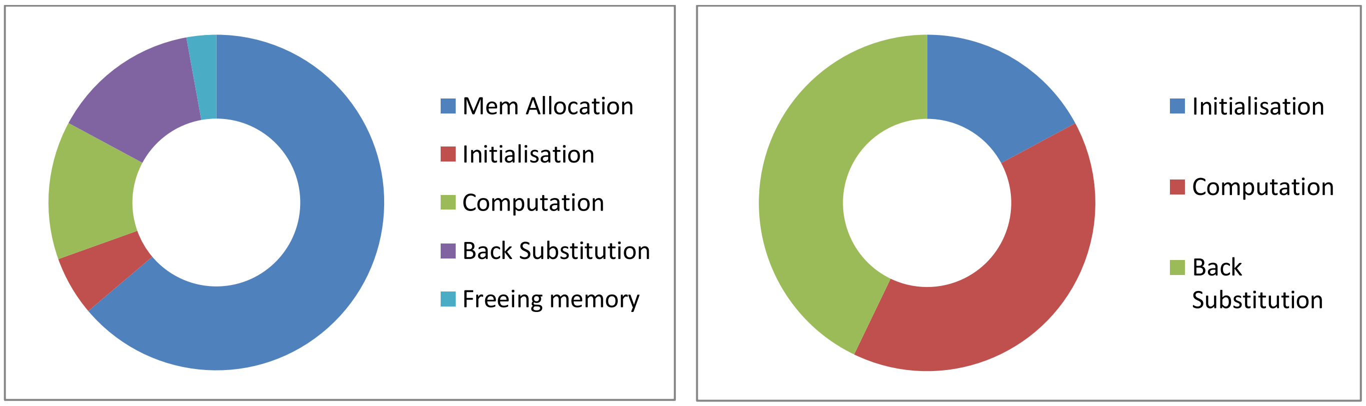


Fig 2.7 (a) and (b) depicting the time distribution for Thomas Algorithm

Thomas Serial Code for m = 1024

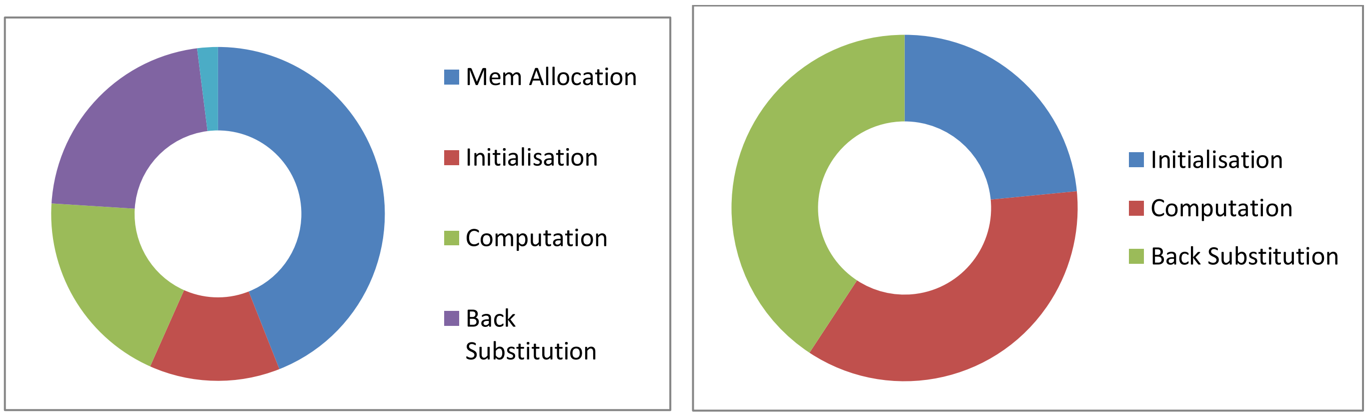


Fig 2.8 (a) and (b) depicting the time distribution for Thomas Algorithm

Thomas Serial Code for m = 2048

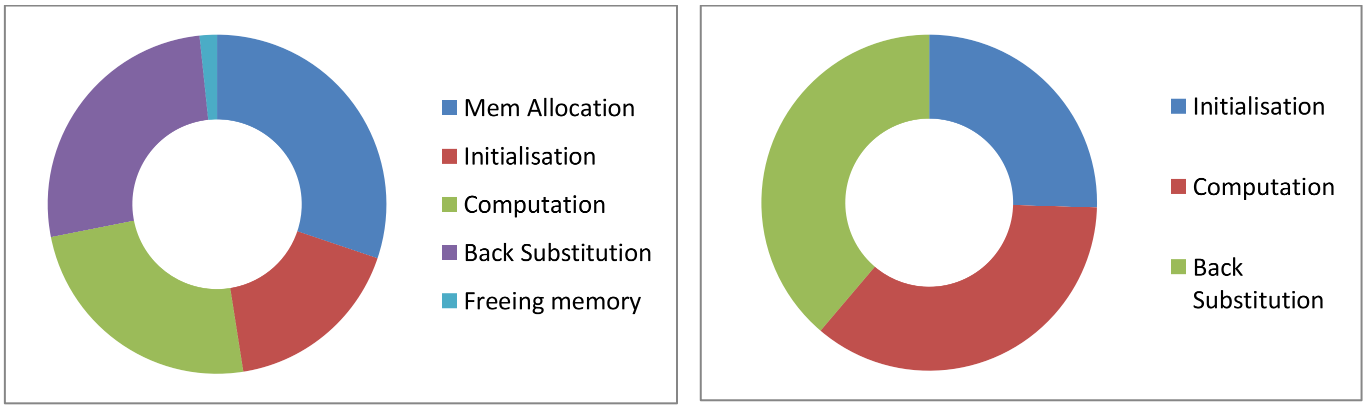


Fig 2.9 (a) and (b) depicting the time distribution for Thomas Algorithm

- The above doughnut figures (Pie Charts) indicate that the Thomas algorithm is inefficient for lower values of matrix order m. It is because it may be seen from Fig. 2.2 – Fig. 2.7, memory allocation takes the major amount of time (percentage wise), while in Fig. 2.8, 2.9, it may be noticed that the computation and back substitution emerge as high time consumers resulting into high (computation time/total time) ratio is high which is desired.
- If memory allocation and Memory freeing time is excluded from the picture and if we analyze only time distribution for initialisation, computation and back substitution (the main Part of our program), we may see that Back Substitution emerges as the major time consumer for lower order of matrix, but the time is distributed equally among the three parameters for higher value of m.
- The idea behind Thomas algorithm to find solution of a tri-diagonal matrix, is based on pivoting the 'n' th element in each row. For example, the first element of first row is made the pivot for the first row, 2^{nd} element of 2^{nd} row acts is made pivot for that row and so on, after which the matrix is reduced (using computation) so that all elements before each pivot become 0 (in each row) and final solution is found using Back Substitution.
- Discussion for parallelizing using MPI, OpenMP, Pthreads:
    - The computation and back-substitution for each row of Thomas Algorithm depend on the previous rows and thus, if it is parallelized using MPI (by giving different rows to different processes) or Pthreads (by giving different rows to different threads), it results into higher time than a serial program. It is because due to interdependency of rows while computation and back-substitution, each process/thread has to wait for values from other processes/threads before completing the computation and thus making MPI highly inefficient for parallelisation.
    - The code may be parallelised using OpenMP on Shared memory architecture. Thus, the whole matrix would be available to all the threads for computation/Back substitution. But the major problem in using OpenMP for parallelisation is that the threads need to be synchronised, i.e., the order of execution of different threads needs to be defined. It is because the threads computing the lower part of matrix need inputs from the upper part of the matrix (because computation for each row is based on the upper row pivot) and thus, even if the matrix lies on shared memory, it cannot be used (upper part needs to pivoted before lower part).

      The shared memory implementation proves to be highly efficient for Back substitution, as the shared matrix can be used as such by all threads (no inter dependent computation problems as in computation part)

Cyclic Reduction:

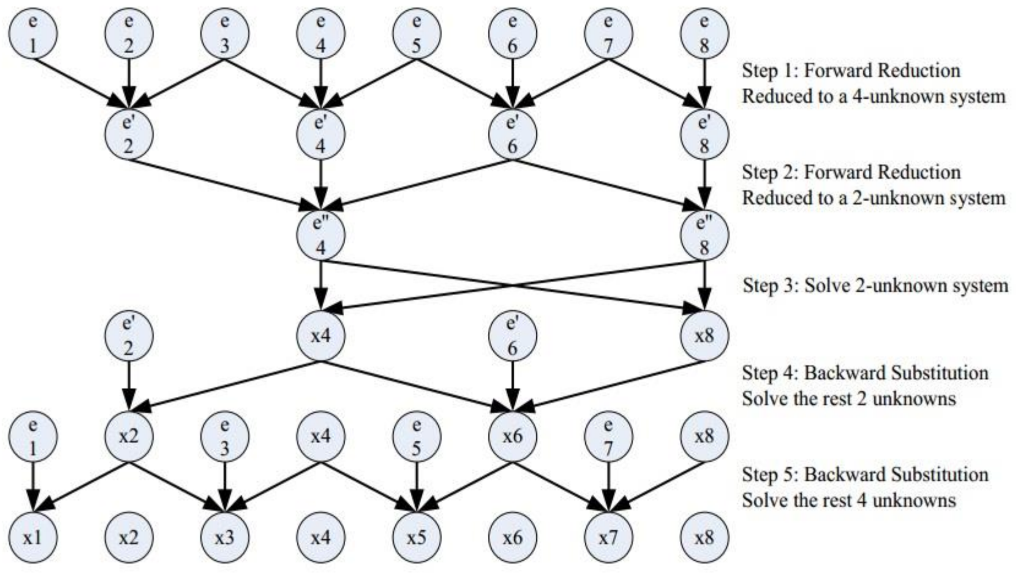

Step 1: Forward Reduction
Reduced to a 4-unknown system
Step 2: Forward Reduction
Reduced to a 2-unknown system
Step 3: Solve 2-unknown system
Step 4: Backward Substitution
Solve the rest 2 unknowns
Step 5: Backward Substitution
Solve the rest 4 unknowns

Figure 2.10 Method (Steps involved) of Cyclic Reduction

To show that for a m x m tri-diagonal matrix, q-1 reductions are required to reduce the original set of equations to a single equation with one unknown, where

$$\boldsymbol{q-1 = \log_2(\frac{m+1}{2}) \quad \Leftrightarrow \quad \frac{m+1}{2} = 2^{(q-1)}}$$

Proof by induction:

Testing the given hypotheses for m=1.

Solving the given equation with m=1, we find that

$$\frac{1+1}{2} = 2^{q-1}$$

$$q = 1$$

Thus it may be seen that given hypotheses is true for m=1 as q-1 = 0 steps are needed for reduction in case of a matrix of order 1.

Considering that the formula holds for m=$\boldsymbol{2^k - 1}$, to prove it true for m=$\boldsymbol{2^{k+1} - 1}$

We have $\frac{2^k - 1 + 1}{2} = 2^{q-1}$

To Prove: $\frac{2^{k+1} - 1 + 1}{2} = 2^q$, i.e., if q-1 reductions are needed for m x m matrix, we have to prove that q reductions would be needed for m' x m' matrix, where m = $2^k - 1$ and m' = $2^{k+1} - 1$

Now, we have $\frac{2^k - 1 + 1}{2} = 2^{q-1} \quad \Leftrightarrow \quad \frac{2^k}{2} = 2^{q-1} \quad \Leftrightarrow \quad \frac{2^k}{2} x\, 2 = 2^{q-1}\, x\, 2 \quad \Leftrightarrow \quad \frac{2^{k+1}}{2} = 2^q$

⇨ $\frac{2^{k+1}-1+1}{2} = 2^q$ => Hence, q reductions are needed for $2^{k+1}-1$ order matrix.

The serial code that used Cyclic Reduction Algorithm for solving the given set of equations with unknowns for a tri-diagonal matrix was executed for different values of m, and the results about the variation of time taken with the order of matrix, time proportion spent in Memory allocation, Initialisation, Computation, Back Substitution and Freeing the occupied Memory are discussed.

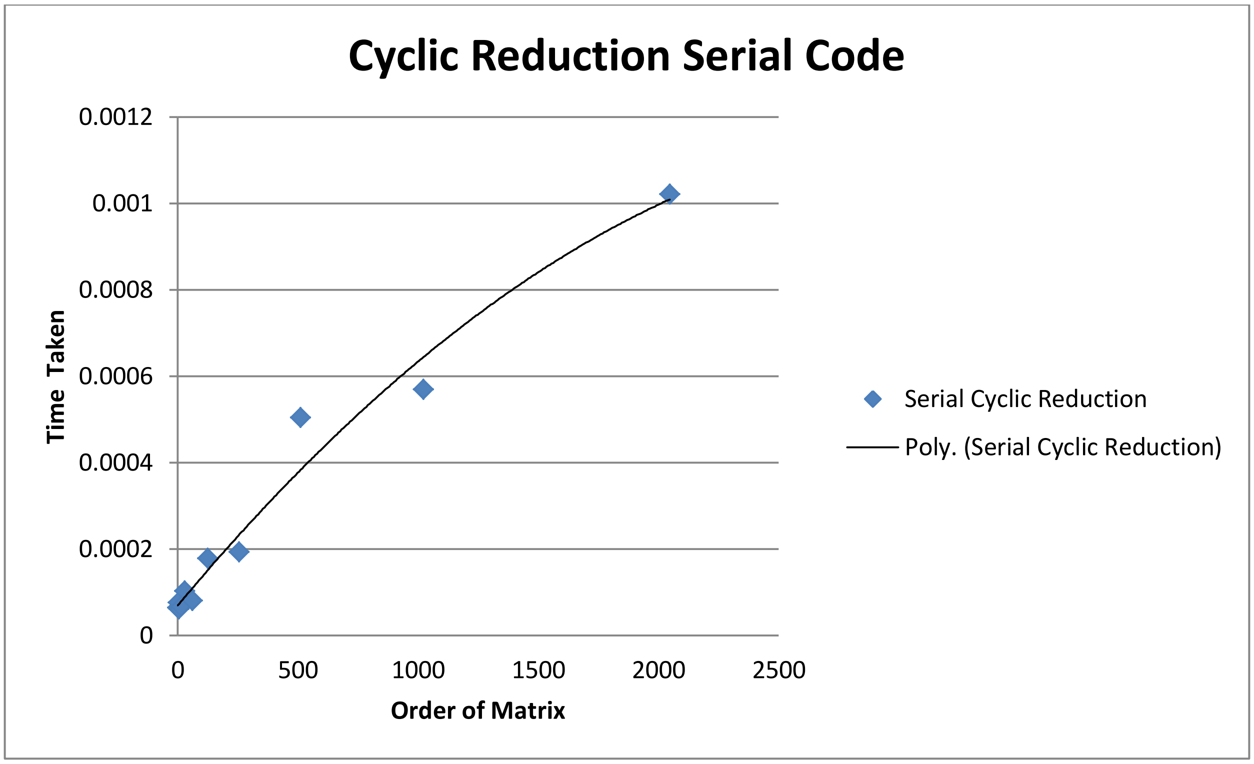
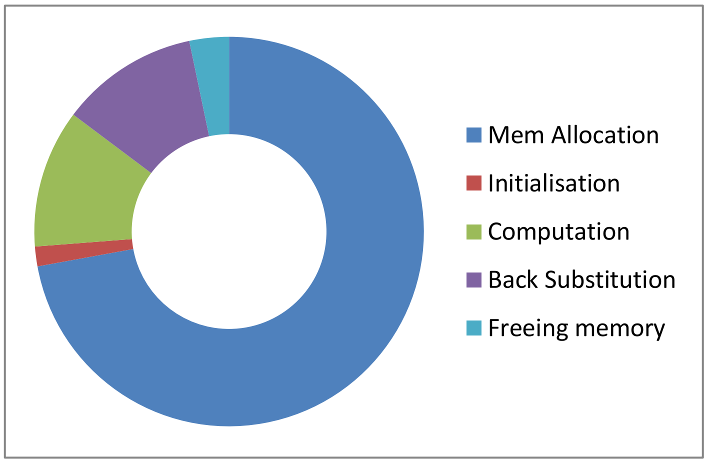


Fig 2.11 Variation of Time taken with change in matrix order for Serial Cyclic Reduction Algorithm

- It may be noticed from Fig. 2.11 that as the matrix order m is increased, i.e., as number of unknowns in the matrix of size m x m is increased, the time taken for solving a serial code based on solving a Tri-diagonal matrix (using Cyclic Reduction Algorithm) increases linearly (approximation). It is because as the order of matrix increases, the steps needed reduce the increased number of equations increase linearly as can be seen in above graph.

Cyclic Reduction Serial Code for m = 7



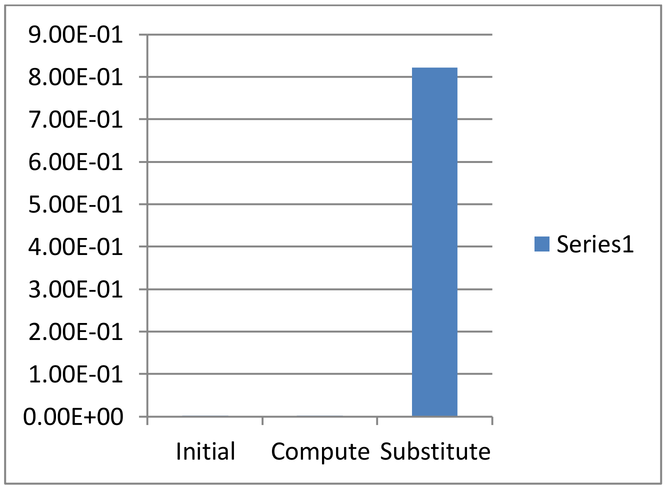
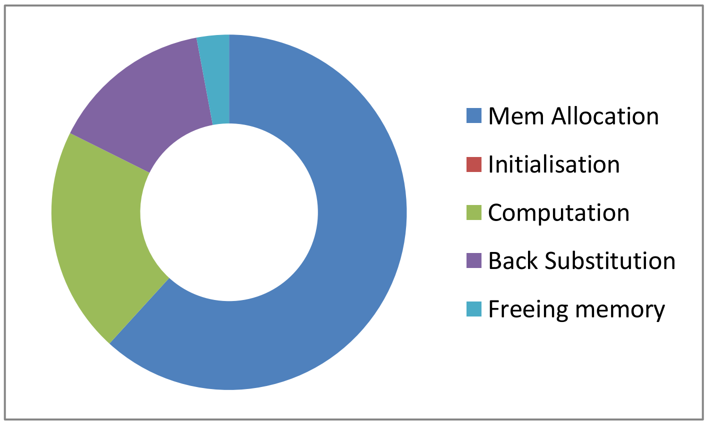


Fig 2.12 (a) and (b) depicting the time distribution for Thomas Algorithm

Cyclic Reduction Serial Code for m = 31



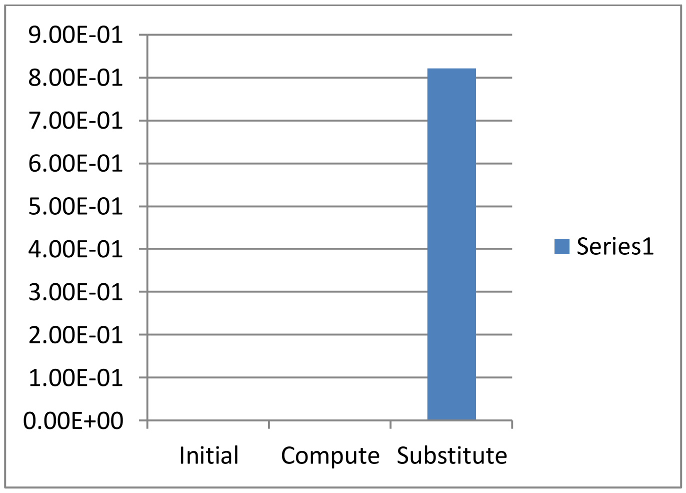
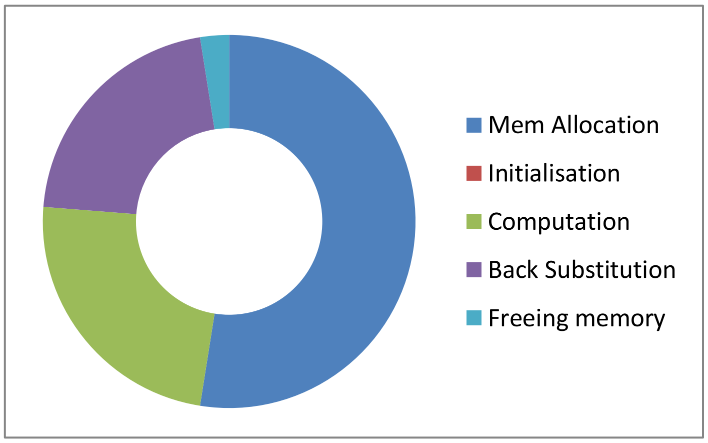


Fig 2.13 (a) and (b) depicting the time distribution for Thomas Algorithm

Cyclic Reduction Serial Code for m = 63



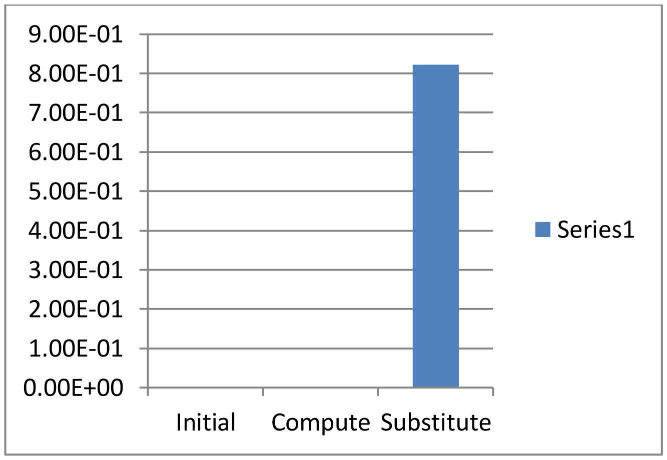
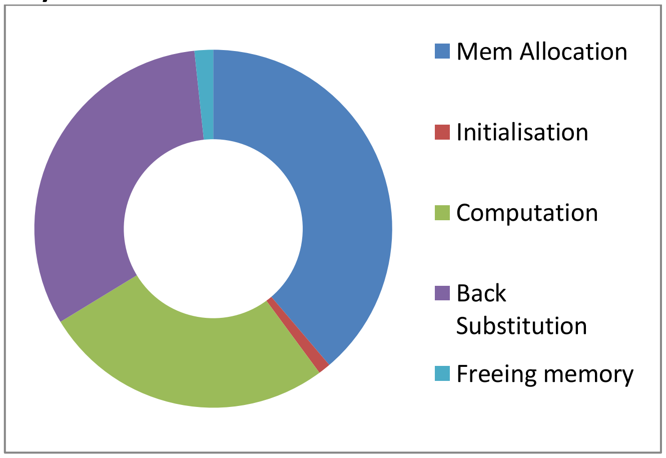


Fig 2.14 (a) and (b) depicting the time distribution for Thomas Algorithm

Cyclic Reduction Serial Code for m = 127



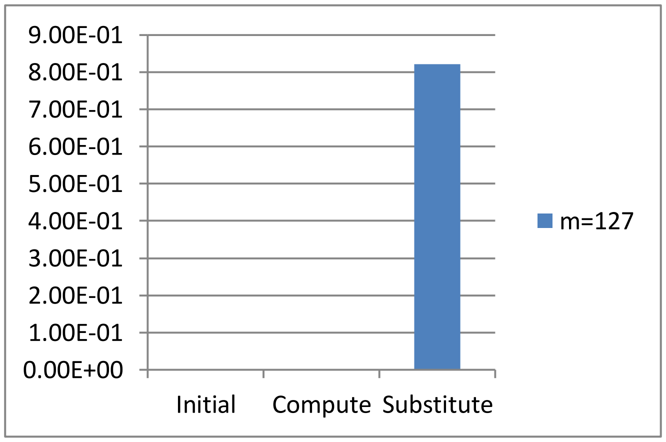
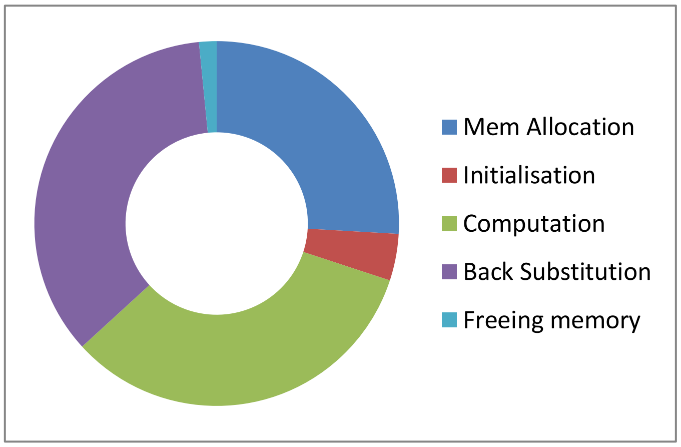


Fig 2.15 (a) and (b) depicting the time distribution for Thomas Algorithm

Cyclic Reduction Serial Code for m = 255



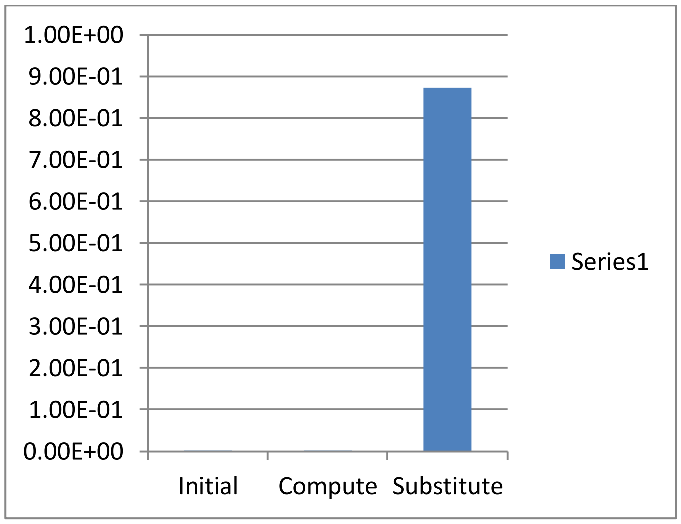
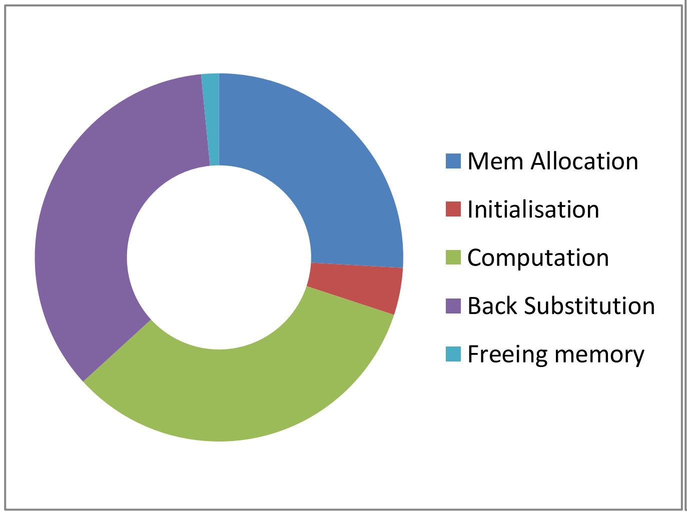


Fig 2.16 (a) and (b) depicting the time distribution for Thomas Algorithm

Cyclic Reduction Serial Code for m = 511



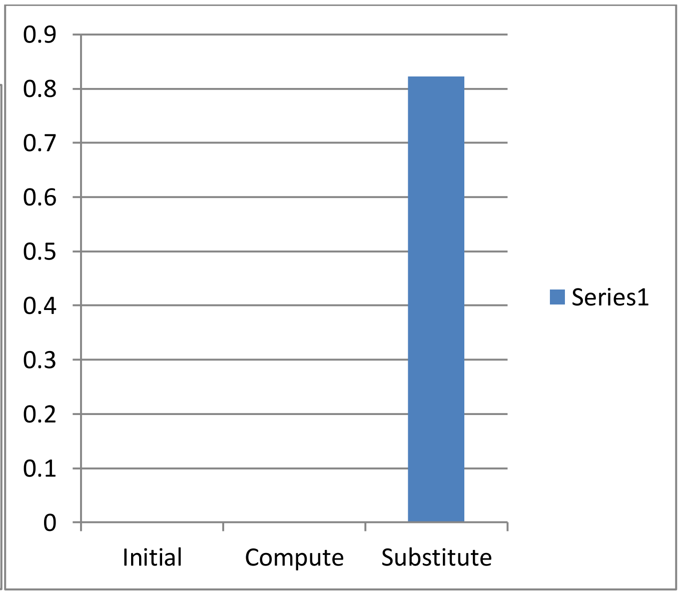
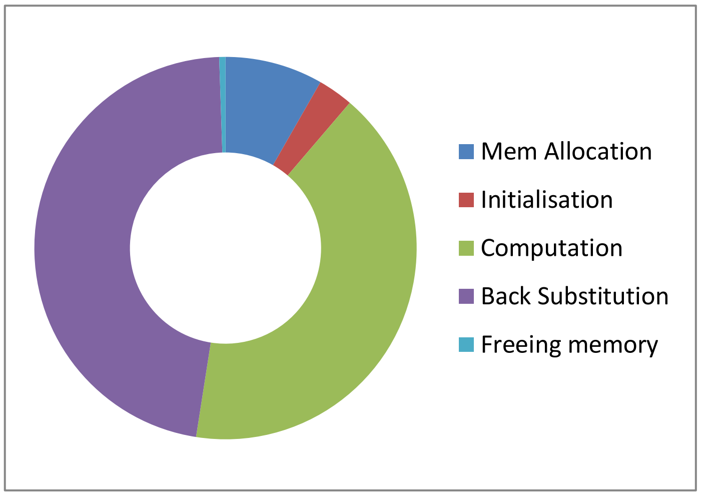


Fig 2.17 (a) and (b) depicting the time distribution for Thomas Algorithm

Cyclic Reduction Serial Code for m = 1023



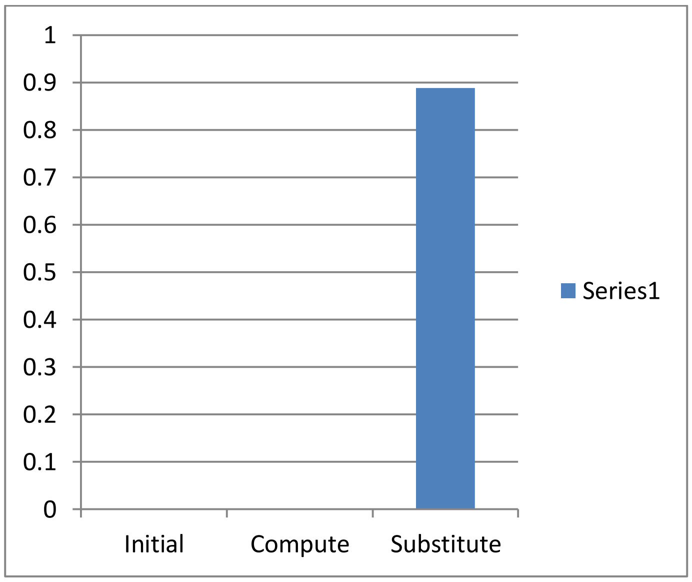
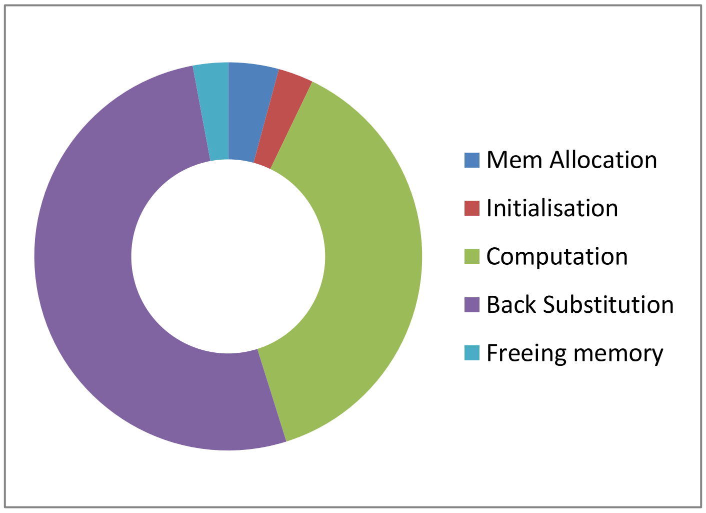


Fig 2.18 (a) and (b) depicting the time distribution for Thomas Algorithm

Cyclic Reduction Serial Code for m = 2047



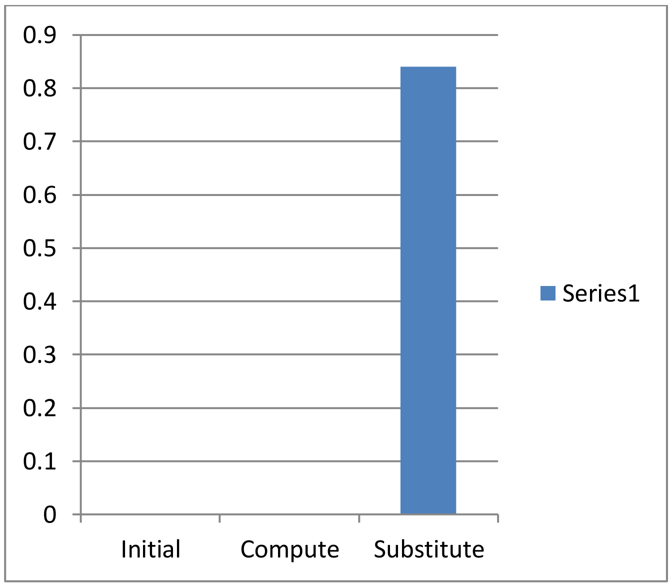


Fig 2.19 (a) and (b) depicting the time distribution for Thomas Algorithm

- The above doughnut figures (Pie Charts) indicate that the Serial Cyclic Reduction algorithm, which has intrinsic parallelism is inefficient for lower values of matrix order m. It is because it may be seen from Fig. 2.12 – Fig. 2.14, memory allocation takes the major amount of time (percentage wise), while in Fig. 2.15-2.19, it may be noticed that the computation and subsequently as the order increases, back substitution emerge as high time consumers resulting into high (computation time/total time) ratio is high which is desired.

- If memory allocation and Memory freeing time is excluded from the picture and if we analyze only time distribution for initialisation, computation and back substitution (the main Part of our program), we may see that Back Substitution emerges takes the maximum amount of time for the given Serial Cyclic Reduction Code. The overall time taken drops when compared to Thomas Algorithm, proving Cyclic Reduction to be more efficient but it may be seen that back substitution consumes highest time as back substitution of a row is highly dependent on other rows.
- The idea behind Cyclic Reduction Algorithm is as mentioned in Fig. 2.10 where an equation with n-unknowns is reduced step-by-step to an equation with single unknown and then the real solutions are calculated.
- The serial code for Cyclic Reduction gives correct value and is confirmed by the formula which gives the analytical solution: $x_k = \frac{kN}{2} - \frac{k(k-1)}{2}$

Parallel cyclic Reduction

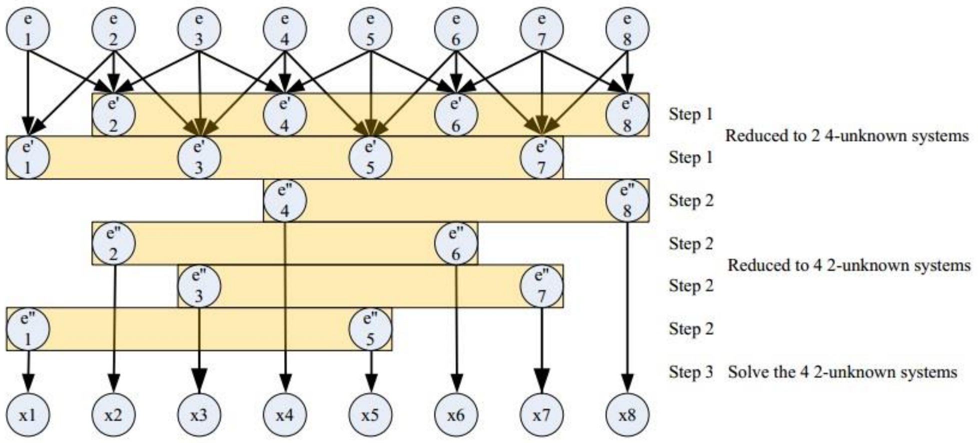


Figure 2.11 Method (Steps involved) of Parallel Cyclic Reduction

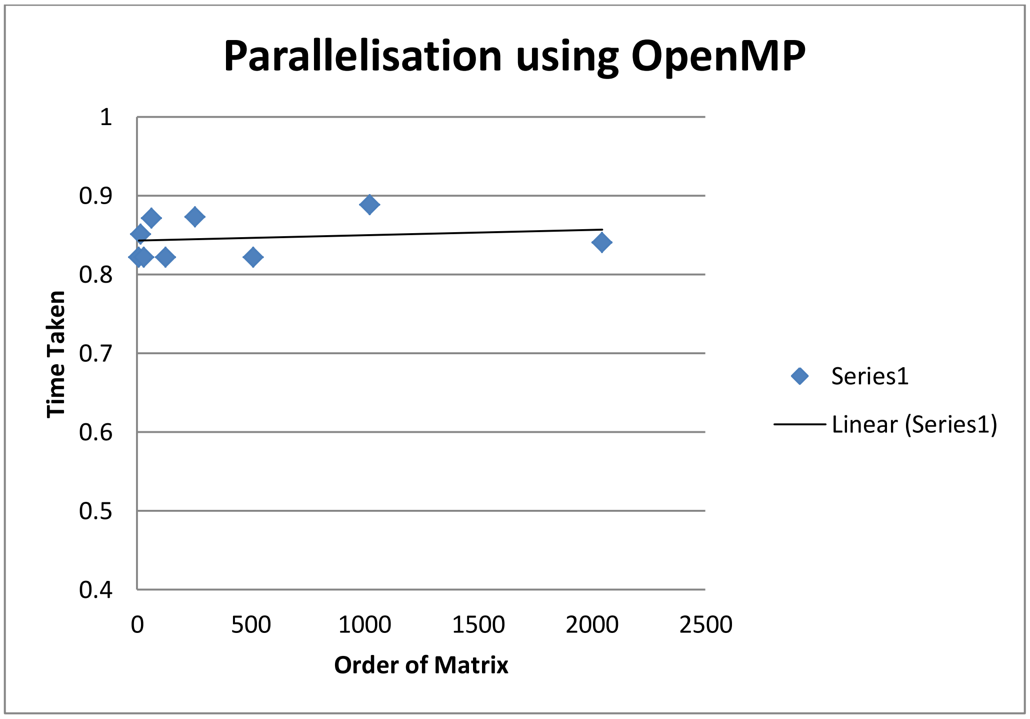


Fig 2.20 Variation in time taken with order of matrix when parallelised using OpenMP

- It is seen that the cyclic reduction problem is easily parallelised on a platform which uses shared memory implementation such as openMP which takes lesser time than Serial Cyclic Reduction
- The code, when written using CUDA, had problems in back-substitution and GPU Thread synchronization which made us to keep that problem for sorting out in a later time.

Conclusion:
Based on the graphs and results discussed above, it may be said that the Thomas algorithm is a non-parallelizable algorithm. Cyclic Reduction Method proves to be more efficient than Thomas Algorithm and parallelising it offers huge advantage over Serial Program. CUDA can only be used for computation and initialisation part in a convenient way and poses problems if the Back Substitution Part is also calculated using CUDA threads.

Acknowledgement

We, members of Group4, have taken efforts in this project. However, it would not have been possible without the kind support and help we received from Prof. Murali Damodaran. We would like to extend my sincere thanks to him. We are highly indebted to his guidance and constant supervision as well as for providing necessary information regarding the project & also for his support in completing the project.
My thanks and appreciations also go to my friends of other groups with whom we constantly interacted and countered the problems we had in developing the project problems and people who have willingly helped us out with their abilities.