INDIAN INSTITUTE OF TECHNOLOGY GANDHINAGAR

ES 611: Algorithms on Advanced Computer Architectures
Report on Computational Lab Project 1.2
by
Shashank Heda
February 9th, 2013

ABSTRACT

The basic essence of High Performance computing lies in proper work load distribution and code parallelization. This project is based on key findings about the need of high performance computation in the field of scientific research.

OVERVIEW

1. Experimentation with Improved Communication Strategies and Derived Data types and Message Packing
2. Communication Strategies for Message Passing
3. Topologies and Communicators in Parallel Matrix-Vector Multiplication
4. Parallelization of Matrix-Matrix Multiplication using Fox's Algorithm

Experimentation with Improved Communication Strategies and Derived Data types and Message Packing

Introduction

Programs to check the improvement in time taken when passing message using different Communication and Packing strategies and tracking the time taken in all cases with change in size of data sent/received.

Method of Approach

Initially, by subtracting the two equations for the given curves, we find the image of the intersecting curve in x-y plane. The image of the non-planar intersection is found to be an ellipse which may be used to set the boundary limits for integration in x- and y-direction.

Figure 1.1 below shoes the way the 2 curves intersect.

The image of their intersection in x-y plane is an ellipse, whose equation is $2\,x^2 + \,y^2 = 1$

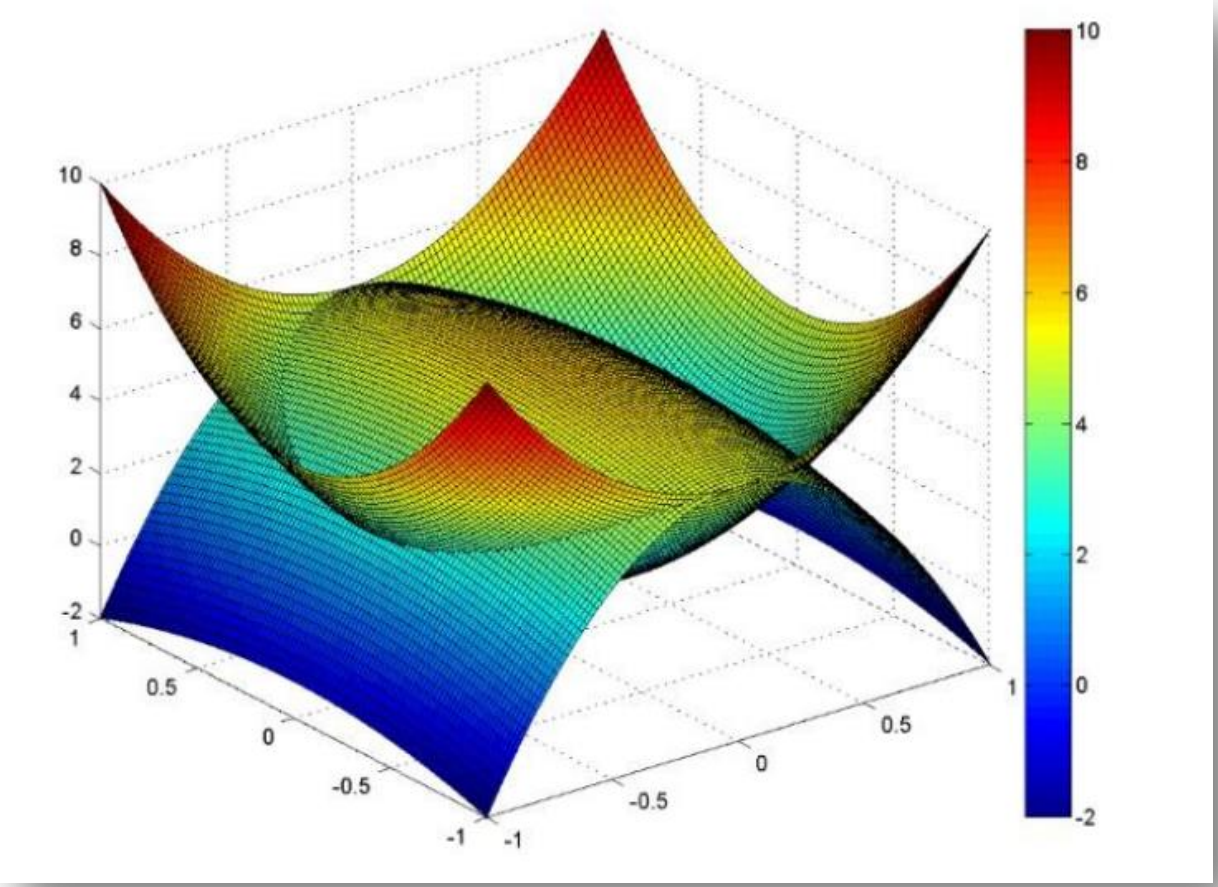

Fig 1.1 MATLAB Plot depicting the intersection of 2 parabolas

Now, the region for which the volume is to be calculated, is divided into many parts. The x-axis is divided into nx fine strips while the y-axis is divided into ny fine strips. The values of nx and ny may be kept equal depending on the precision required in the calculation.

The finely divided x-limit are given to various processes, the region of integration being $-\frac{1}{\sqrt{2}}\ to\ \frac{1}{\sqrt{2}}$.

The y- limits of integration depend on the x-limits and are $-\sqrt{1-2x^2}\ \ to\ \sqrt{1-2x^2}$

Taking every data-point in x-direction at an interval of hx, we hereby get a set of data- points of y at an interval hy. The area of the small rectangular element is hence hx*hy. For volume of the cuboid formed between the elemental points (x[i], y[i], z1[i]) and (x[i], y[i], z2[i]), the area is multiplied to the functional value at corresponding points and then subtracted to get the common volume inside the figure.

Summing this entire sum obtained in above over x from index 1 to nx, yields us the volume of the figure required. The accuracy and precision of the value depends upon the number of grid points one specifies and also the degree of parallelization specified for the code.

Results and Discussion

- As the number of grid points, nx and ny are increased, at initial runs, the calculated answer approaches closer to the analytical answer at a higher rate of convergence than that compared to the rate when the number of grid points is increased even after obtaining sufficiently accurate answer. This phenomenon is depicted through figure 1.2.

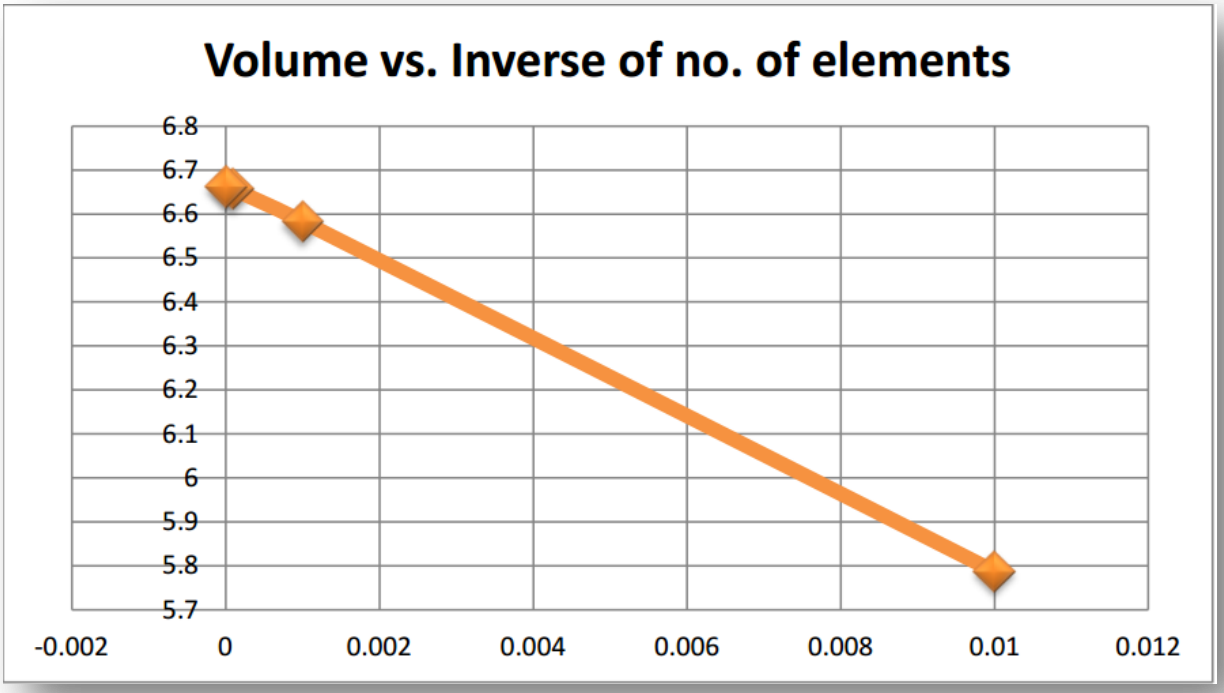

Figure 1.2 Graph depicting variation in the volume between 2 paraboloids with number of elements

(Volume is on y-axis while No. of elements is on x-axis)

- From figure 1.2, it may be inferred that the error in calculating the volume at each stage is dependent upon the error occurred in previous stage because the marginal decrement in the accuracy is the key to the above observation.

- This also implies that for obtaining a fairly accurate answer, increasing the number of grid points after a certain range isn't an efficient way for a better yield since that merely increases the computation and communication time while the resultant change in accuracy decreases marginally.

- Thus we should have an optimum number of processes working towards solving this problem rather than increasing the cores beyond needed.

- Observing the statistics of performance through get_data3.c and get_data4.c files, figure 1.3 for get_data3.c is included below which shows the results of Speedup vs. Number of Processors used:

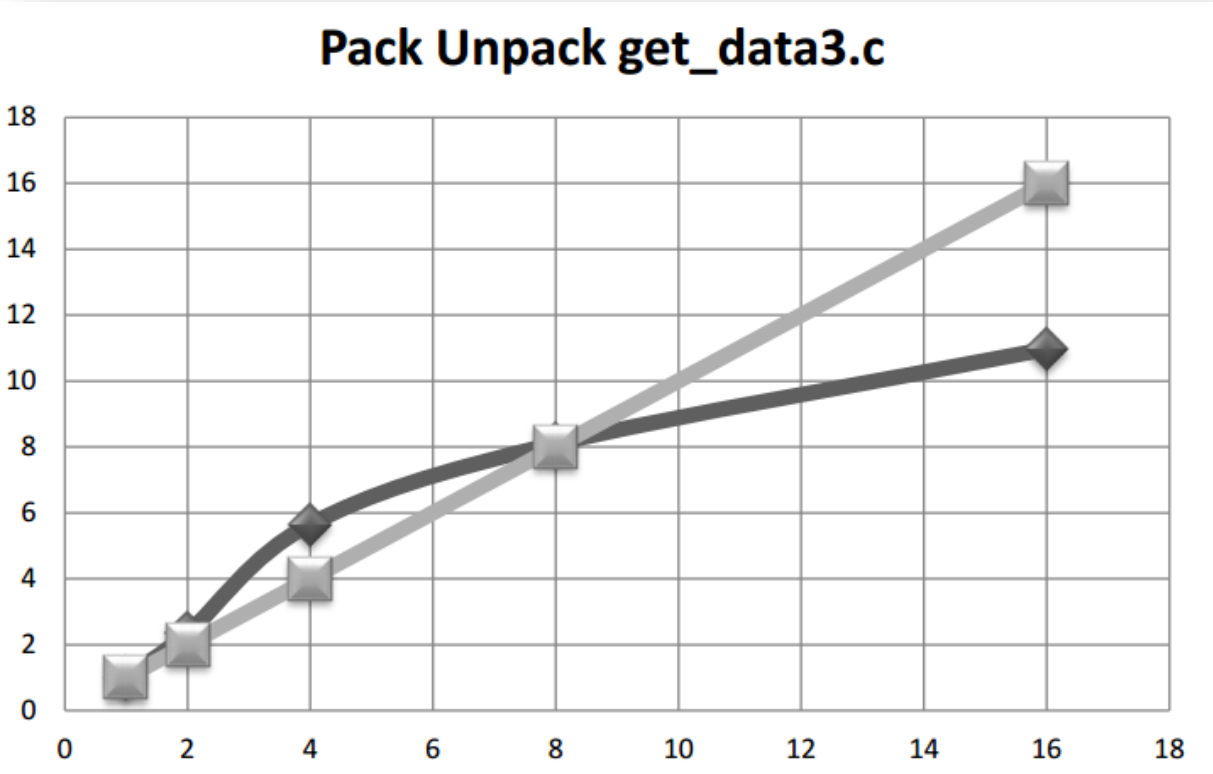

Fig 1.3 Variation of Speedup (Y-axis) with number of processes

- Even though the speedup increases as expected while increasing the number of processors, the extent by which it increases is decreasing with increasing the number of processors.

- For number of processors = 1, 2 and 4, the speedup obtained is quite impressive but from number of processors = 8 onwards the speedup achieved starts decreasing than expected and then finally, if the curve is extrapolated, it seems to saturate after a certain number of processors.

- One of the possible inferences that can be inferred from figure 1.3 is the Validation of Amdahl's law: the crucial parameter he ignored and is visible here is the problem size. Depending upon the problem size, there exists a trade-off between the communication

time and the computation time which at their minima gives a fairly correct estimate of number of processors that should be used to effectively handle the balance of speed, accuracy and resources.

- In order to have a rough idea about the number of processors advisable to use for this problem size, figure 1.4 indicates Increment in Speedup (on Y-axis) vs. Corresponding increment in number of processors

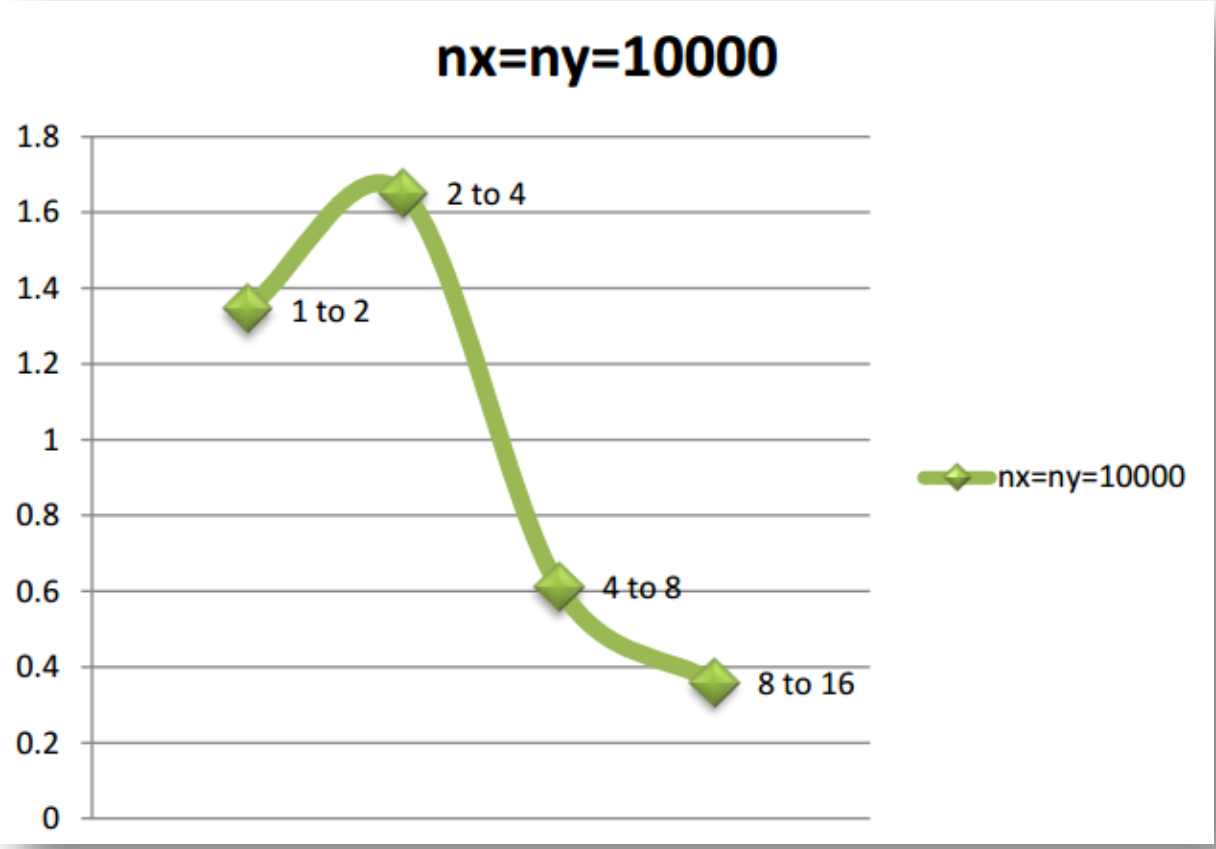

Fig 1.4 Change in Speedup with change in number of processes for nx = ny = 10000

- From figure 1.4 it shows that for a switch of number of processors from 1 to 2 and from 2 to 4 the increment in speedup is quite well but, a switch from 4 processors to 8 processors and then from 8 to 16 leads to a high decrement in speedup. This shows that the optimum number of processors advisable to use on this problem size can be considered to be 4.

- Analysis of the code using get_data4.c file, figure 1.5 is shown below:

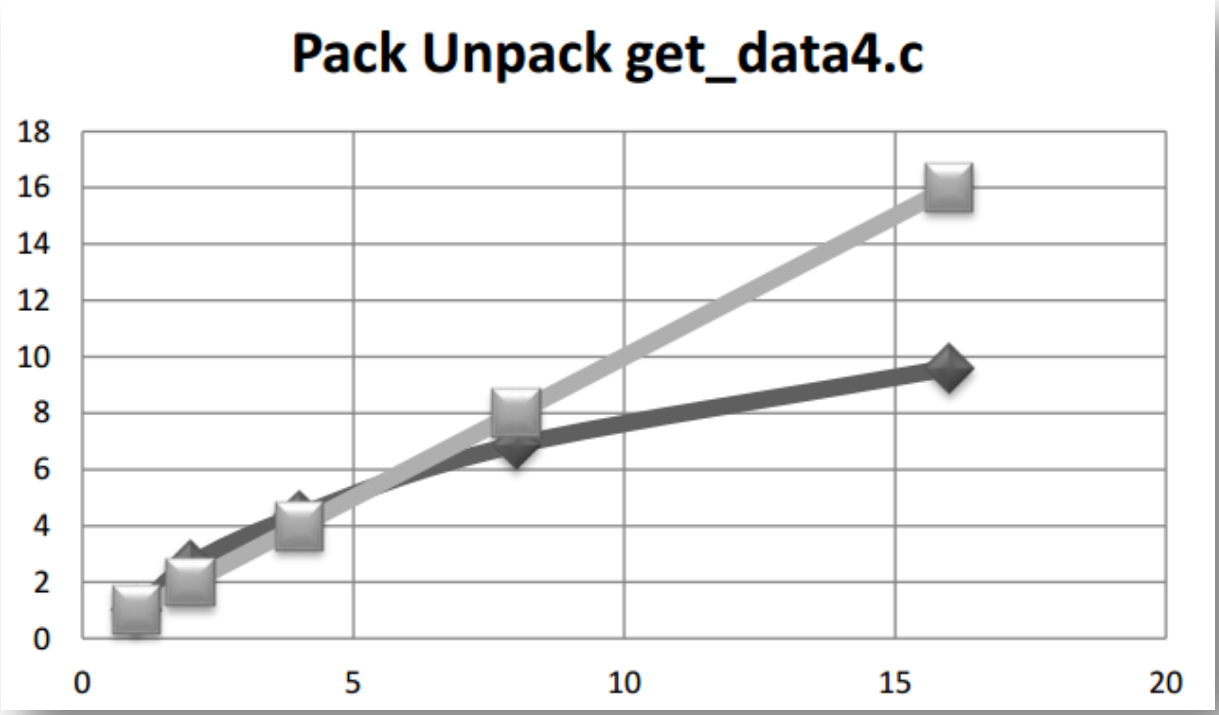

Fig 1.5: Change in Speedup with change in number of processes

- As shown in figure 1.5, it follows more or less a similar trend as that in get_data3.c but one of the major differences in figure 1.4 and figure 1.5 is that as the number of processors increases in the former one, the maximum speedup reached lies somewhere above 10, but this is not the case with the later one: here the maximum speedup attained is lesser than 10.

- One of the probable reasons behind this can be that in the former one, the data is sent as a single message and is broadcasted to all the other processors, hence the execution time for this function consumes a lesser time than creating a new datatype using MPI_Pack followed by broadcasting it and then unpacking it.

- However, it cannot be considered a valid, generalized explanation to say that broadcasting is always a better way to communicate as compared to the Pack- Broadcast-Unpack option; the reason being that as described by Gustafson's observation, the size of the problem is also one of the decisive parameters for defining the efficiency of parallel computation. Here, only a communication among four processors is considered and hence, the time elapsed for broadcasting all values isn't quite significant to the other way of Pack-Broadcast-Unpack. Albeit, it is possible in case of higher order of communication, that the later described way

could be efficient than the former one since there, the communication is for a large chunk of messages while in the later strategy, it broadcasts the required messages *one-shot.*

- Figure 1.6shows marginal increment in speedup with respect to corresponding increment in number of processors:

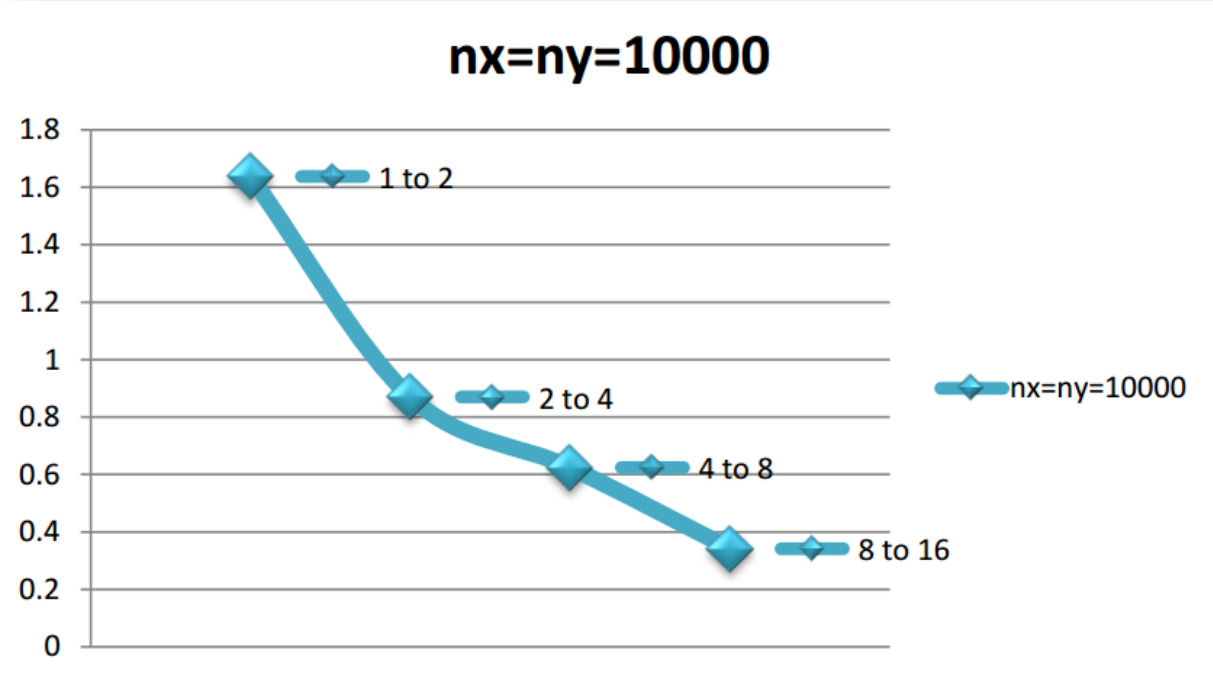

Fig 1.6 Change in consecutive speedup vs. every two-fold increase in number of processes

- Analysing the effect of continuously decreasing speedup per every two-fold increment in number of processors, it shows that the number of processors optimal for implementing the get_data4.c is 2 since a switch from number of processors =1 to number = 2 gives the maximum possible increment in speedup for the given parameters of performance. Hence, the explanation in case of above point seems logically plausible in a sense that since the message size to be communicated here is quite small and also the number of messages to be clubbed together is negligible enough.

Communication Strategies for Message Passing

Introduction:

Communication between various processes for parallel computation is the heart of MPI which decides the maximum efficiency that can be attained by a program. This problem presents various types of message passing strategies.

Method of Approach:

As described in the problem statement, each message is passed on through successive processors via ring-pass scheme exploring various modes of communication strategies including blocking and non-blocking send-receive constructs, collective communication constructs like broadcasting the message, packing-broadcasting-unpacking scheme vs. the individual broadcasting scheme and its analysis.

Results and Discussion

- Figure 1.7 describes the jumpshot profile of the ring pass messaging through blocking and non-blocking send-receive constructs

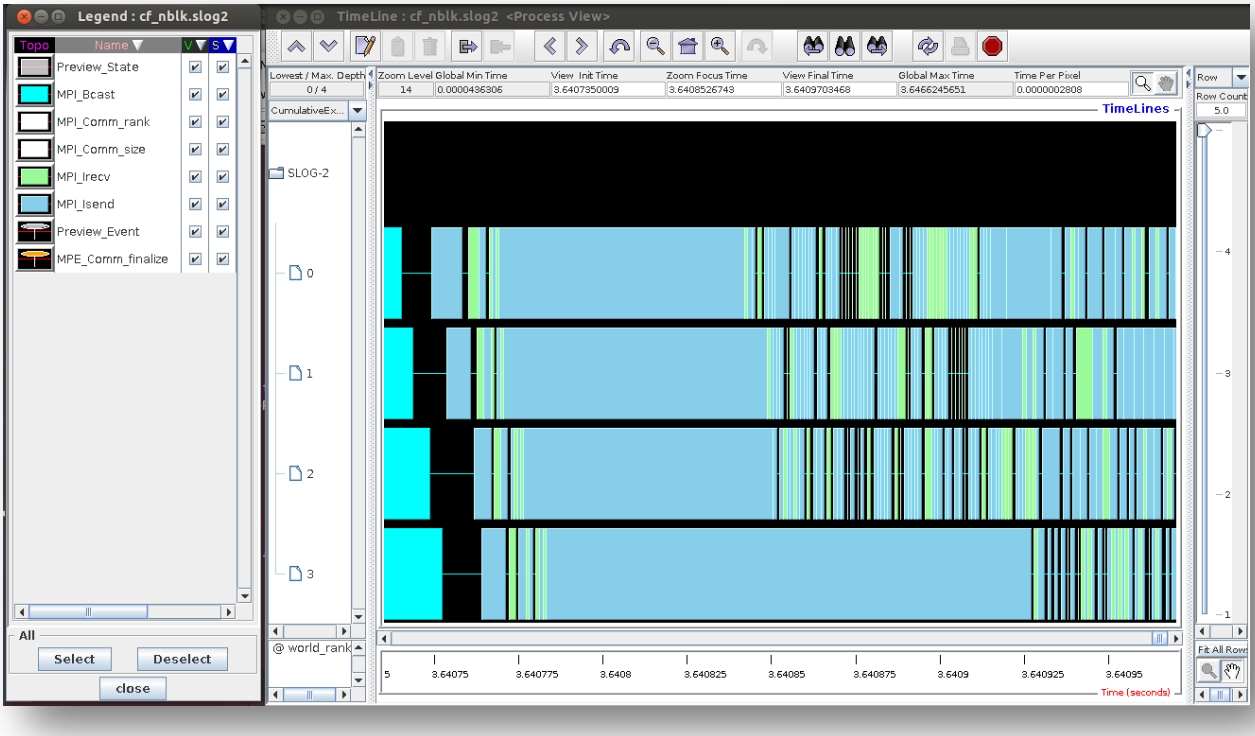

Figure 1.7 Jumpshot Profile for Ring Passing through MPI_Isend and MPI_Irecv constructs

- One of the important points to be observed in this profile is that the MPI_Bcast is quite an expensive function in terms of the time it takes to broadcast the message. The initial light blue region for all the four processors is filled with the time elapsed by the broadcast command. Hence, it should be taken care of while using the broadcast function that the marginal utility of it does not cost the user unnecessary time delay.

- Also, due to several blocking send-receive constructs, the time taken by the process is quite more in a sense that it is forced to wait until the corresponding processor receives the message and gives the confirmation. However, this is quite useful in terms of ensuring security of the message. Also, blocking send-receive constructs help in effective manual debugging of the code.

- Followed by the blocking send and receive construct, figure 1.2(b) shows the jumpshot profile for non-blocking send-receive routine, the one which is generally used in a wide variety of codes.

- Figure 1.8 was for the number of flips = 10000. Hence, in order to realize the effect of increasing the number of flips among various processors, below is figure 1.2(c) describing the jumpshot profile of the same

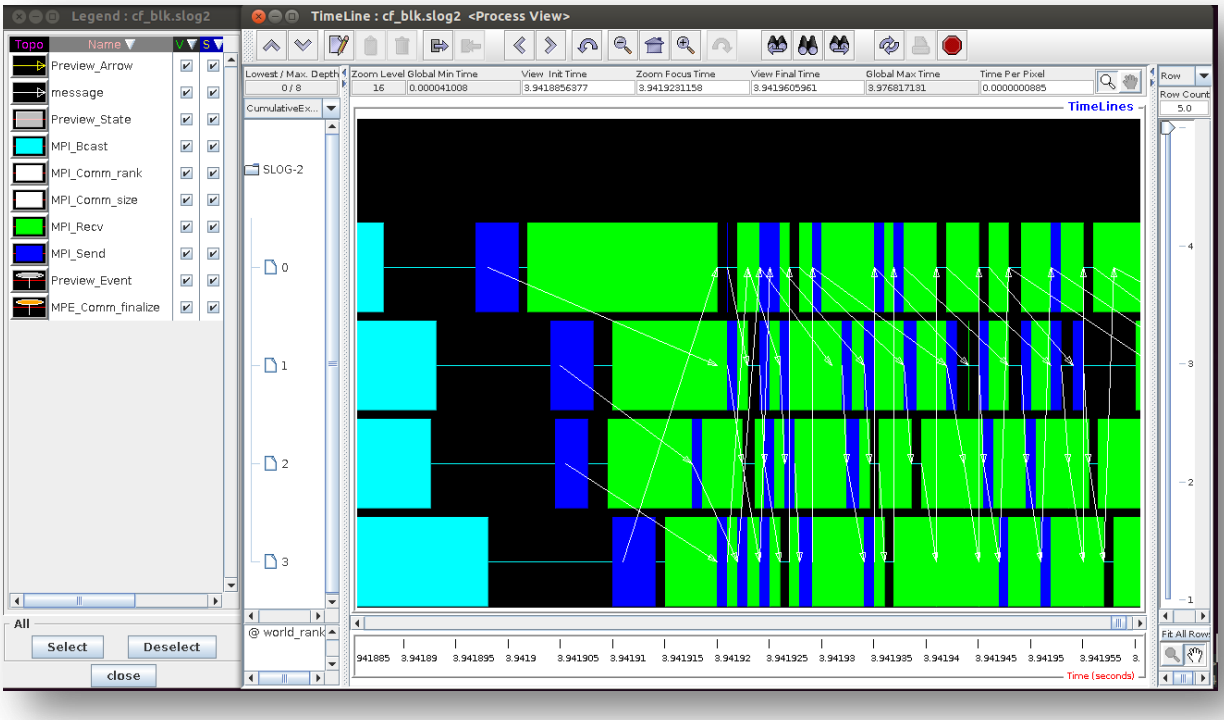

Figure 1.8 Jumpshot Profile for non-blocking send-receive routine

- Even in this case, the MPI_Bcast takes enormous time but due to the non-blocking point-to-point communications among the processors, the time elapsed in that case is quite a bit reduced as compared to the profile in figure 1.7. As far as the data-communication analysis is concerned, the only inference that can be strongly made is that for the given size of the communication in given problem, there is not a need of implementing a blocking send-receive construct since here, every processor is able to communicate with each other efficiently without actually needing those specialized constructs. However, that never says that the blocking send-receive constructs are useless. Maybe in several other applications, there might be a need of those functions where they fit appropriately.

- As it can be seen form figure 1.9, the time taken by the increment in number of flips is fairly higher than that taken for the case of figure 1.8. However, the increase in accuracy and precision in the value of the percentage turning out to be head increases significantly. Hence, this is the situation where we need to make a trade-off between the time taken against accuracy and precision of the problem

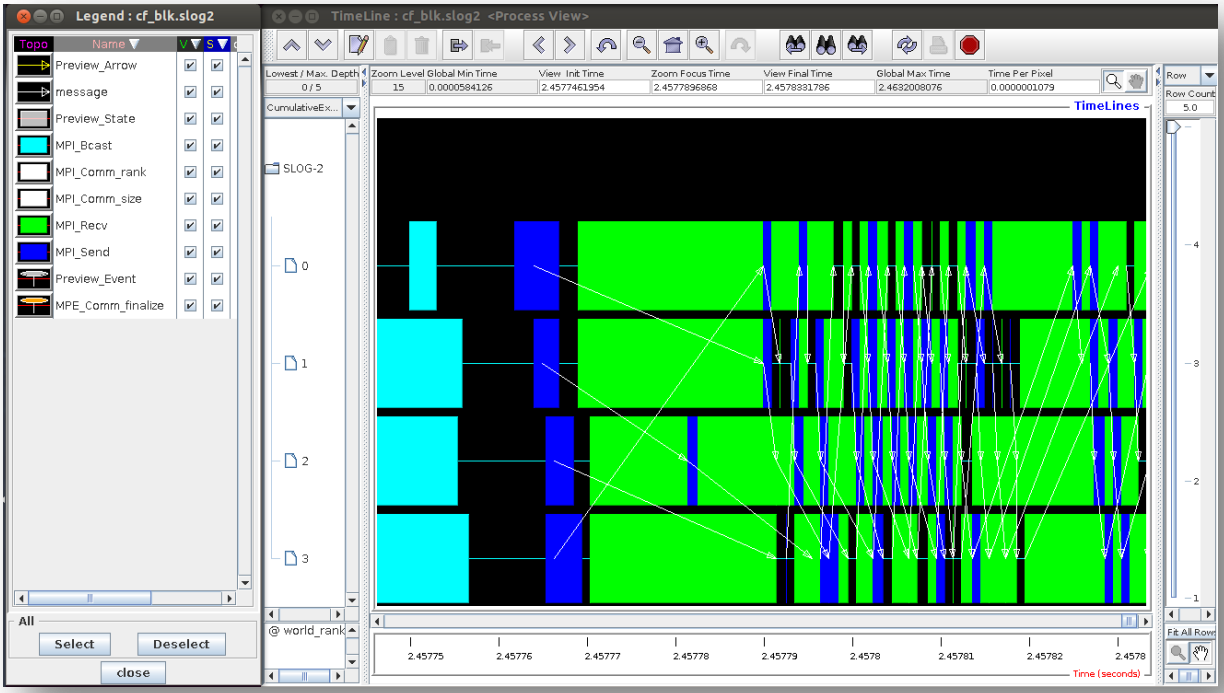

Fig 1.9 Jumpshot Profile for non-blocking send-receive routine for 1000000 flips

- Various types and modes of blocking and non-blocking send-receive constructs need to be analyzed appropriately before utilizing them in the code: the reason being sometimes, through the name it looks as if the function is needed in the code but knowing well the intricacies related to it as discussed above, one can get a better insight of what all parameters to be eyed at before using them.

Parallelizing Matrix-Matrix Multiplication using Fox's Algorithm

Introduction:

Matrix-matrix multiplication has many real life applications and prevails in almost all the areas affiliated to scientific computation and research. This problem includes an analysis of matrix-matrix multiplication through the Fox's Algorithm technique used for parallelizing the operation via the MPI interfacing.

Method of Approach:

- The multiplication process is performed according to the standard Fox's Algorithm in order to be able to parallelize the operations. These operations include generation of two random matrices, storing them, and performing step-by-step procedure parallelized among the specified number of processors.

- As per the Fox's algorithm, for multiplication of A[n][n]* B[n][n], in the first step, the principal diagonal entries of A are broadcasted to all the processors. Simultaneously, the diagonal entries broadcasted are multiplied to each corresponding row element individually in the B matrix and the new values are stored in the C matrix (the resultant matrix). Here, each cluster of (n/p) rows of the B matrix is stored in one processor and hence, the step 1 of multiplication is completed.

- This step is followed by the chronologically next set of diagonal elements of A matrix being broadcasted to all the processors and also, a row shifting in B matrix is carried out in a manner that nth row is shifted to the (n-1)th row, (n-1)th row to (n-2)th row and so on. This is followed by multiplication again as carried out in step 1 and the process continues till n cycles, each time adding the new result obtained to the previously stored result in the C matrix. Hence, after n cycles, we get the resultant matrix C as the final answer of the multiplication.

- The parallelization of the code is carried through the Grid-Communication topology. This topology introduces a new communicator in the form of a virtual grid of processors setup according the dimension and optimization parameters specified by the user.

- The grid dimension in this case would be 2 since the processors are to communicate among themselves the various components of matrices as described above. There are several functions in the fox.c code which forms the basis of constructing a virtual grid of processors.

Results and Discussion:

The serial_mat_mult.c code provided works well for matrices upto the order 512 and after, it starts taking a huge time, due to problems with memory allocation. Figure 1.10 shows graphical result for the time taken when two square matrices are multiplied.

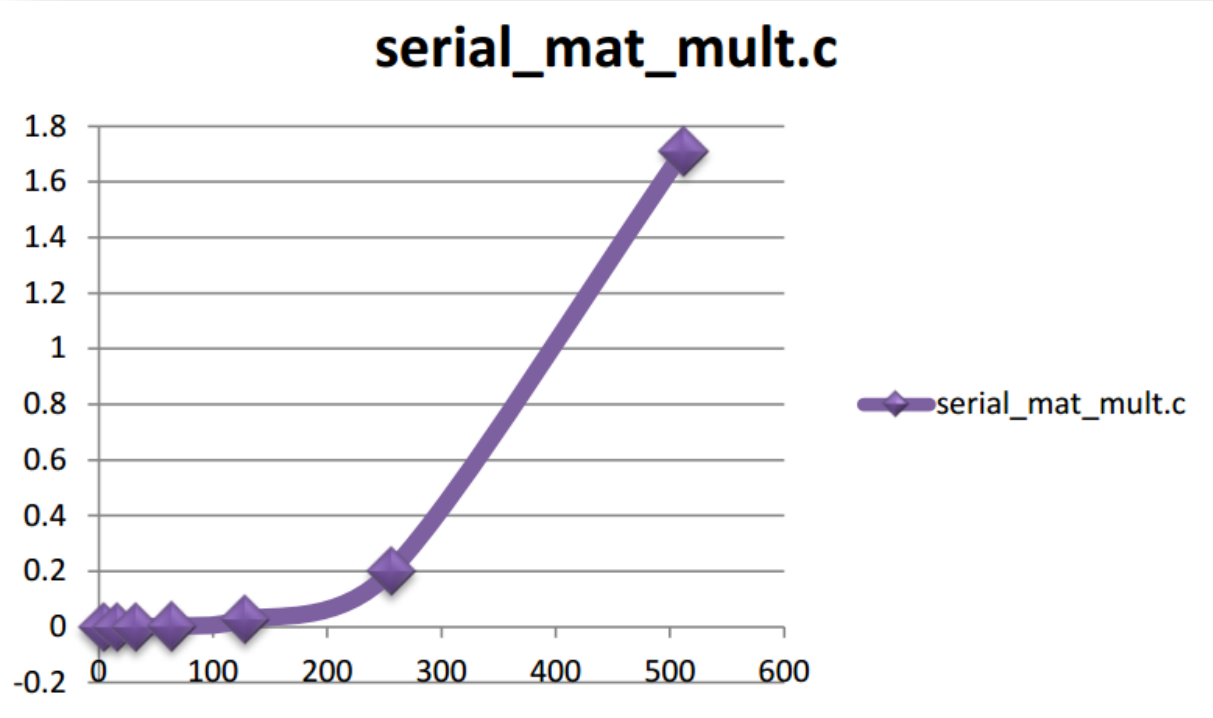

Fig 1.10 Time taken by serial Matrix Multiplication vs Order of Matrix

- The probable exponential increment in the time against varying order of matrices is a result of high memory storage requirements along with the increase in execution process to multiply dense square matrices.

- These timings are compared with that of running the parallelized fox.c code through one processor.

- Figure 1.11 depicts comparison of timings taken by serial matrix multiplication code (Ts) and the parallelized fox.c code running on 1 processor (T1).

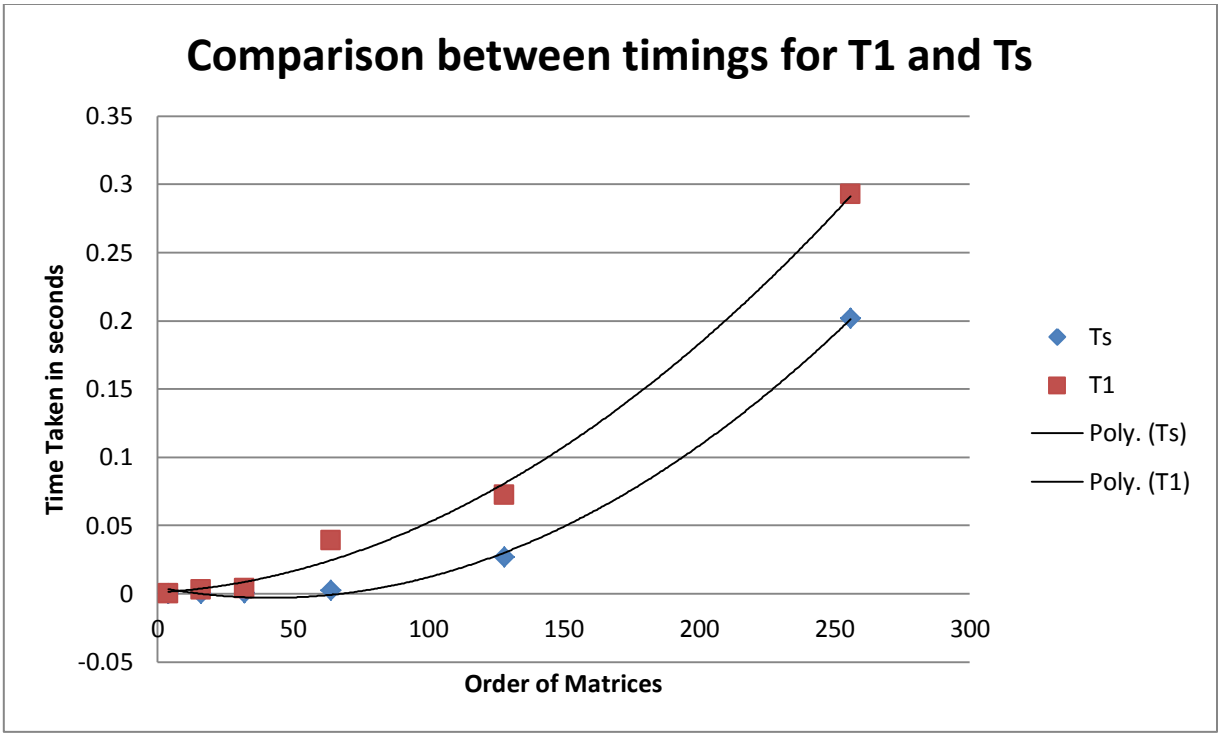

Fig 1.11 Change in Time taken by serial code vs. that taken by a parallel code for one processor with varying order of matrix (to be multiplied)

- Here, the time taken by parallelized code goes significantly higher as the order of the matrices increases. One of the reasons responsible behind this result is that the parallelized code needs to create an entire virtual grid of processors as a primary communicator and then it starts its calculations whereas there is no such new communicator to be setup for the serial code case and hence, a serial code saves that much time by starting the calculations immediately the time from which the program is executed. Hence, the time T1 starts moving higher than that of Ts.

- Figure 1.12(a) and 1.12(b) depicts the plot for time taken by the parallel fox.c code for matrices of certain orders of matrices.

- Further, the separate plot of the same parameters as in figure 1.12(a) but for the order of matrices = 64 is shown in figure 1.12(b). The main reason for inclusion of this graph separately is that time consumed by the resources is almost ten-fold higher than the initial orders.

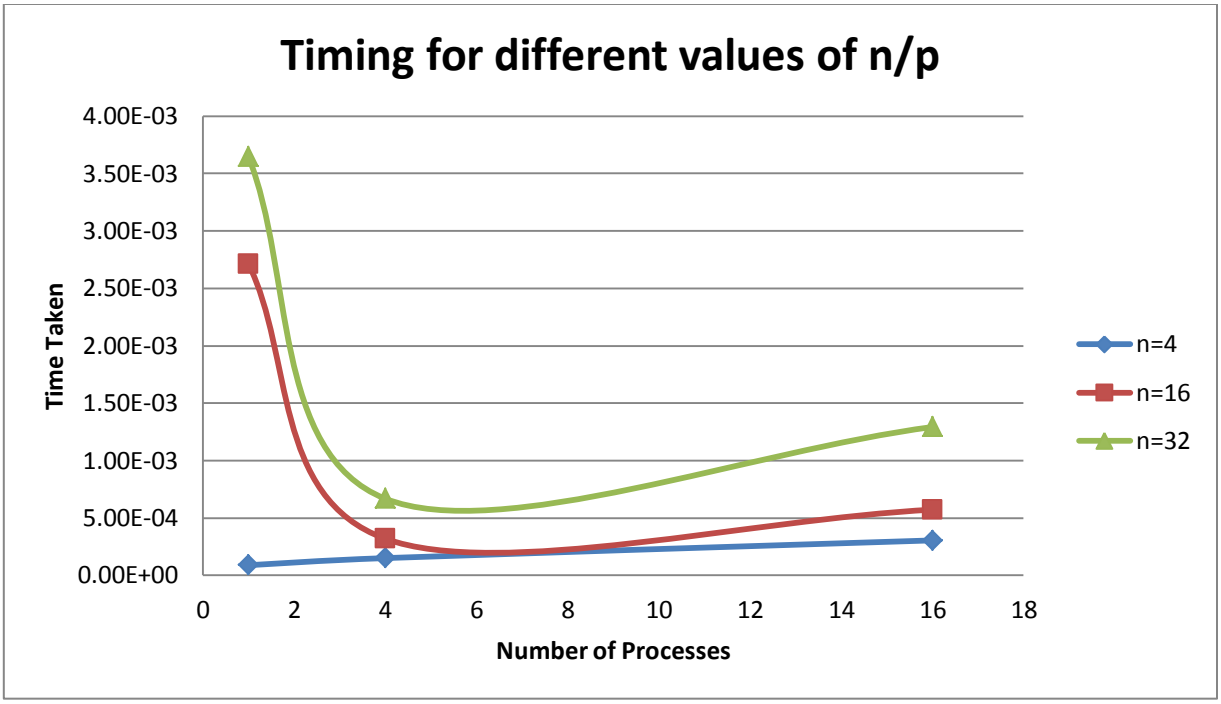

Fig 1.12 (a) Time taken vs. number of processors for a particular value of n

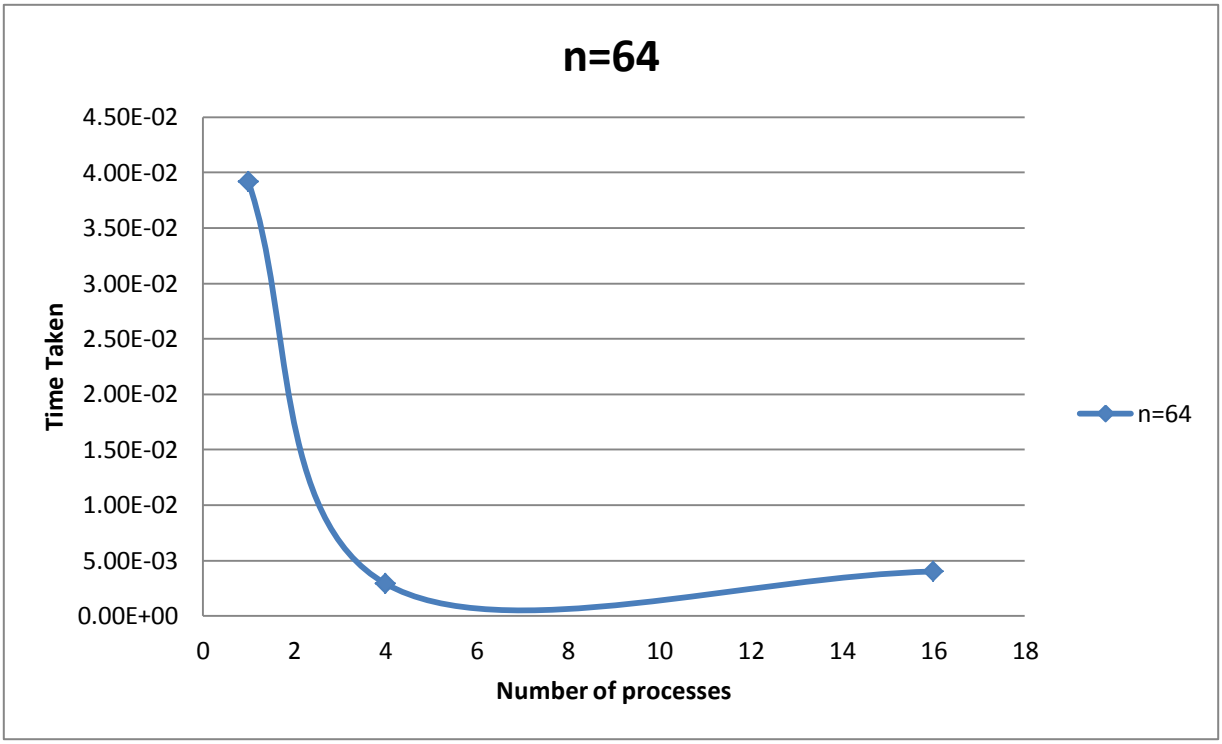

Fig 1.12 (b) Time taken vs. number of processors for a particular value of n

- As discussed in initial points above in this case, it is also visible in figure 1.12 (a) and (b) that the time keeps on decreasing for a switch from 2 to 4 processors but then, again starts increasing from 8 processors onwards, whatever the order of the matrix be.

- A slight off track behaviour can be seen for fourth ordered matrices and the probable reason for this can be that for such a small size of the problem, the efforts invested in parallelizing the computation are quite more than the actual computation needed to be carried out. However, this combination justifies the efforts vs. overall gain for using 4 processors and starts its non-ideal behaviour from then onwards. Hence, this plot supports our initial observation of the optimum number of to be 4 in this case.

- The cause for the shown behaviour is that the increase in the size of the matrix causes a significant issue of high memory storage requirements and frequent access of memory for multiple processors that contribute to increase in time taken by the code to execute.

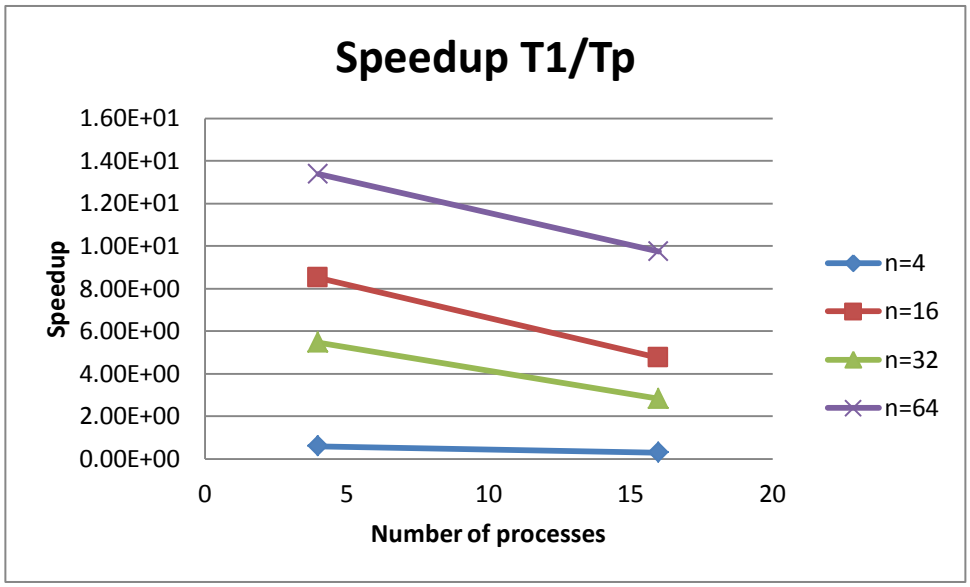

Fig 1.13(a) Speedup vs Number of Processors for various values of order of matrices where T1 is the time taken by parallelized code when run on 1 processor

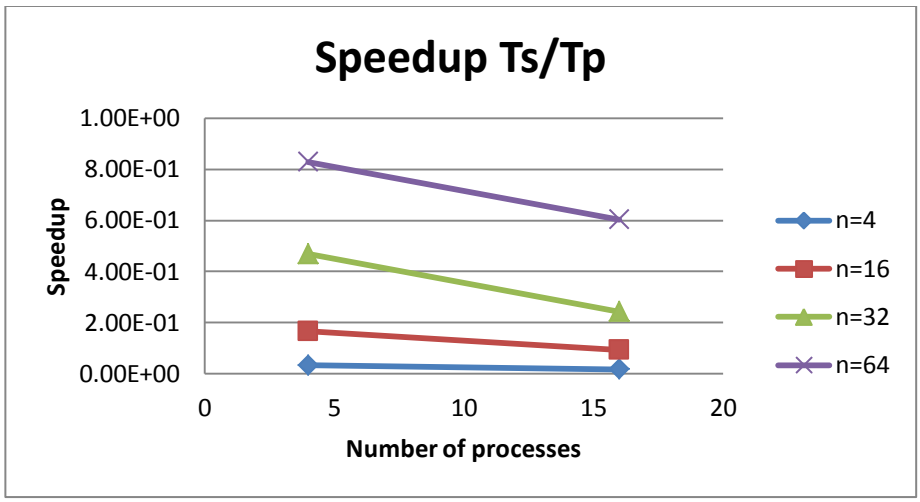

Fig 1.13(b) Speedup vs Number of Processors for various values of order of matrices where Ts is time taken by serial code

- As indicated in the figure 1.13(a) and 1.13(b), the speedup decreases with increasing number of processors irrespective of the order of matrices. Hence, the analysis done for case: 1 of volume of integral matches with the results stated by figure 1.13(a) and 1.13(b).
- The jumpshot profile for the parallelized fox.c code is included in figure 1.14

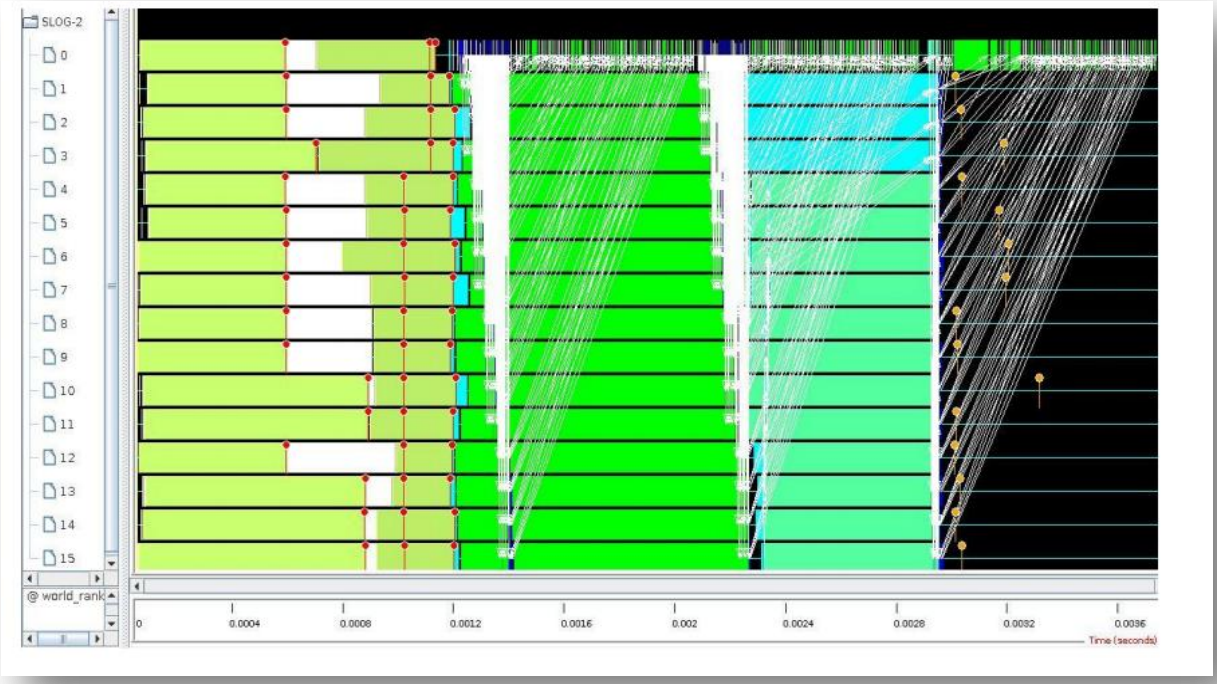

Fig 1.14 Jumpshot Profile for parallel fox.c (made parallel using MPI)

- As shown in figure 1.14, the major time of the code is invested in creating the Grid-Communicator topology for the overall communication tasks to handle. Hence, this explains the reason why this step played a crucial role in consuming more time for small sized problems and ineffective communication delays.

Acknowledgement

I have taken efforts in this project. However, it would not have been possible without the kind support and help I received from Prof. Murali Damodaran. I would like to extend my sincere thanks to him. I am 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 fellow team mates: Ravi, Nishant and Parth in developing the project codes and people who have willingly helped me out with their abilities.