Chirag Soni: 665930262
Sanamdeep Singh Anand: 677787570

PARALLEL PROCESSING: LAB3

GOAL: The goal is to first perform the matrix multiplication and to compute $|X^k|$ in parallel, where
• X is a n × n matrix
• k is an input parameter
After performing the matrix multiplication, we need to calculate the determinant of the resultant matrix.

MPI Functions used in the ring and hypercube algorithm:

1. **MPI_INIT**: Initiate an MPI computation. (Called only once during the execution of a program)
2. **MPI_COMM_SIZE**: Determine number of processes.
3. **MPI_COMM_RANK**: Determine my process identifier.
4. **MPI_SEND**: Send a message.
5. **MPI_RECV**: Receive a message.
6. **MPI_Get_processor_name (processor name, &name_len)**: Returns the name of the processor on which it was called now of the call.
7. **MPI_Wtime:** Returns the floating-point number of seconds, representing elapsed wall-clock time since some time in the past.
8. **MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[], const int periods[], int reorder, MPI_Comm *comm_cart)** : This command is used to create a new communicator to which the communication information has been attached. The ndims specifies the number of dimensions in our cartesian grid, dims tell us the number of processors in each dimension and periods is a logical array that specifies whether the grid is periodic or not.

Chirag Soni: 665930262
Sanamdeep Singh Anand: 677787570

9. **MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[]):** This method is used to specify the cartesian coordinates of the specified processor within the implemented cartesian topology.
10. **MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)** : This command is used to specify the rank pf the given processor in the network.

The program use two different mapping: **ring and 2D mesh**; with data decomposition.

The determinant is computed using three different algorithms:

1. **Traditional m^3:-**

The traditional algorithm works right to left and does explosions ("carries") as one goes along. On paper it is swift and compact and this might be why it has been the favored way of doing long addition for centuries.

2. **Cannon's method: -**

Cannon's algorithm is a distributed algorithm for matrix multiplication for two-dimensional meshes It is especially suitable for computers laid out in an *N* × *N* mesh. While Cannon's algorithm works well in homogeneous 2D grids, extending it to heterogeneous 2D grids has been shown to be difficult. The main advantage of the algorithm is that its storage requirements remain constant and are independent of the number of processors. The Algorithm for this works as follows:

We need to select k in every iteration for every Processor Element (PE) so that processors don't access the same data for computing aik * bki. Therefore, processors in the same row / column must begin summation with different indexes. If for example PE (0, 0) calculates a00 * boo in the first step, PE (0,1) chooses a01 * b11 first. The selection of k: = (i + j) mod n for PE (i, j) satisfies this constraint for the first step. In the first step we distribute the input matrices between the processors based on the previous rule. In the next iterations we choose a new k': = (k + 1) mod n for every processor. This way every processor will continue accessing different values of the matrices. The needed data is then always at the neighbor processors. A PE (i, j) needs then the a from PE (i, (j + 1) mod n) and the b from PE ((i + 1) mod n, j) for the next step. This means that a must be passed cyclically to the left and b cyclically upwards. The results of the multiplications are summed up as usual. After n steps, each processor has calculated all aik * bki once and its sum is thus the searched cij.

The Run time of the algorithm is

$$T(n,p) = T_{coll}(n/N,p) + N * T_{seq}(n/N) + 2(N-1)(T_{start} + T_{byte}(n/N)^2)$$

where Tcoll is the time of the initial distribution of the matrices in the first step, Tseq is the calculation of the intermediate results and Tstart and Tbyte stands for the time it takes to establish a connection and transmission of byte respectively.

3. Chain matrix multiplication:

Matrix chain multiplication (or Matrix Chain Ordering Problem, MCOP) is an optimization problem that can be solved using dynamic programming. Given a sequence of matrices, the goal is to find the most efficient way to multiply these matrices. The problem is not actually to *perform* the multiplications, but merely to decide the sequence of the matrix multiplications involved.

There are many options because matrix multiplication is associative. In other words, no matter how the product is parenthesized, the result obtained will remain the same. For example, for four matrices A, B, C, and D, we would have:

$$((AB)C)D = (A(BC))D = (AB)(CD) = A((BC)D) = A(B(CD)).$$

However, the order in which the product is parenthesized affects the number of simple arithmetic operations needed to compute the product, that is the computational complexity. For example, if A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix, then computing $(AB)C$ needs (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations, while computing $A(BC)$ needs (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.
Checking each possible parenthesizing (brute force) would require a run-time that is exponential in the number of matrices, which is very slow and impractical for large n. A quicker solution to this problem can be achieved by breaking up the problem into a set of related subproblems. By solving subproblems once and reusing the solutions, the required run-time can be drastically reduced. This concept is known as dynamic programming.

Working of the code:

A matrix raised to the power k is basically multiplying a matrix with itself (k-1) times. So, in our code we have multiplied the given matrix by itself k-1 times.
First, we try to multiply two matrices. The syntax of the two matrices is given below:

A =	a00	a01	a02	B =	b00	b01	b02
	a10	a11	a12		b10	b11	b12
	a20	a21	a22		b20	b21	b22

The multiplication of the two matrices is shown below:

A x B =	a00*b00 + a01*b10+a02*b20	a00*b01 + a01*b11 + a02*b21	a00*b02 + a01*b12 + a02*b22
	a10*b00 + a11*b10 + a12*b20	a10*b01 + a11*b11 + a12*b21	a10*b02 + a11*b12 + a12*b22
	a20*b00 + a21*b10 + a22*b20	a20*b01 + a21*b11 + a22*b21	a20*b02 + a21*b12 + a22*b22

If we notice carefully, the first column of A is multiplied only with the first row in B. The bold characters highlight this characteristic.

A x B =	**a00*b00** + a01*b10+a02*b20	**a00*b01** + a01*b11 + a02*b21	**a00*b02** + a01*b12 + a02*b22
	a10*b00 + a11*b10 + a12*b20	**a10*b01** + a11*b11 + a12*b21	**a10*b02** + a11*b12 + a12*b22
	a20*b00 + a21*b10 + a22*b20	**a20*b01** + a21*b11 + a22*b21	**a20*b02** + a21*b12 + a22*b22

Similarly, we see that the 2nd Column in A is multiplied only with the 2nd row in B and the 3rd column in A is multiplied only with 3rd row in B. So, each processor gets 1 row and 1 column.
Once each processor has the particular data, both the column and row vectors are multiplied to get a 3x3 matrix.
After this step, each processor has a 3x3 matrix. Now, all the processors send their data to the processor with rank 0. The processor with rank 0 receives the data and adds them. This addition of all the data from the processors gives us the required matrix multiplication.
If the value of k is greater than 2, this process is repeated k-1 times to get the final result.
Once, we have calculated the final matrix at the processor with rank = 0, we calculate its determinant using an algorithm that uses cofactor process.

2D mesh

We can see from the figure that a node at row *i and column j* is connected to all its neighbors in the four directions i.e. it is connected to the nodes at the location (*i-1,j*), (*i+1,j*), (*i,j+1*) and (*i,j-1*). The nodes at the edges of the topology have either 2 or 3 neighbors only. The 2-D mesh network helps in parallelizing the transfer of data between the processors. Instead of sending data one by one to all the processors, the data is sent in the following way: Consider a 4 x 4 2-D mesh network as shown in the above figure. Let the 0th processor contain the data. In the first step, the 0th processor sends to the 2nd processor in the horizontal direction. Now, the 2nd processor also has the data. In the next step, the 0th and the 2nd processor sends the data to the 1st and the 3rd processors parallely. After this step the first row has the data. In the next step, all the 4 processors from the 1st row send the data to all the 4 processors in the 3rd row. And in the last step the processors from the 1st and the 3rd rows send the data to the 2nd and the 4th rows parallely.
Data Transfer in a 2D mesh

Chirag Soni: 665930262
Sanamdeep Singh Anand: 677787570

RESULTS

For Parameter Ranges:-
Matrix size (n) when the power of matrix (k) is 4:

- 2X2
- 4X6
- 6X6
- 8X8

And
When the power of matrix (k) is 8:

- 10X10
- 12X12
- 14X14
- 16X16

Where number of processors(p) used are

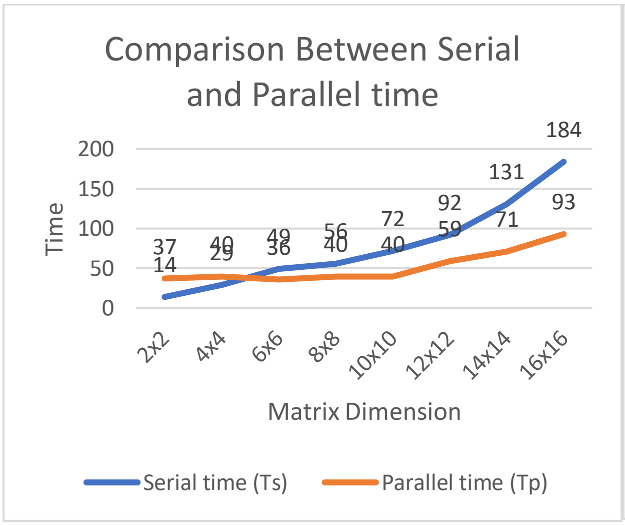
- 1
- 4
- 8
- 16

Analysis

Matrix Dimension (n)	*Input Parameter (k)*	*Serial time (Ts)*	*Parallel time (Tp)*	*No of Processors (P)*	*Speed Up (S) = Ts/Tp*	*Efficiency =S/P*
2x2	*4*	*14*	*37*	*2*	*0.378378378*	*0.189189189*
4x4	*4*	*29*	*40*	*4*	*0.725*	*0.18125*
6x6	*4*	*49*	*36*	*6*	*1.361111111*	*0.226851852*
8x8	*4*	*56*	*40*	*8*	*1.4*	*0.175*
10x10	*8*	*72*	*40*	*10*	*1.8*	*0.18*

12x12	*8*	*92*	*59*	*12*	*1.559322034*	*0.129943503*
14x14	*8*	*131*	*71*	*14*	*1.845070423*	*0.131790744*
16x16	*8*	*184*	*93*	*16*	*1.978494624*	*0.123655914*

Figure 1



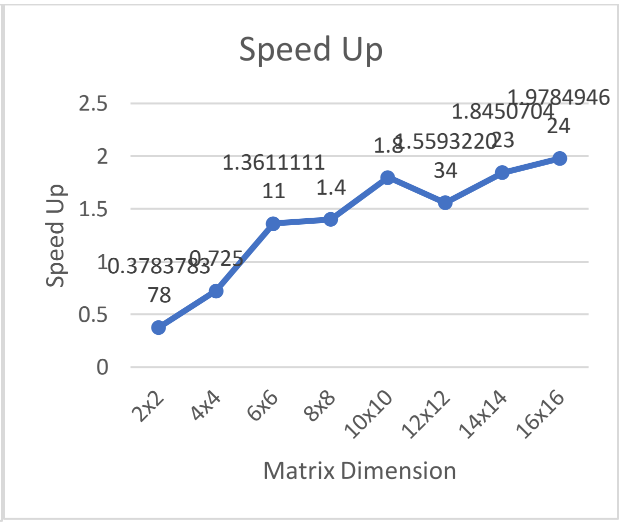


Figure 2

Figure 3

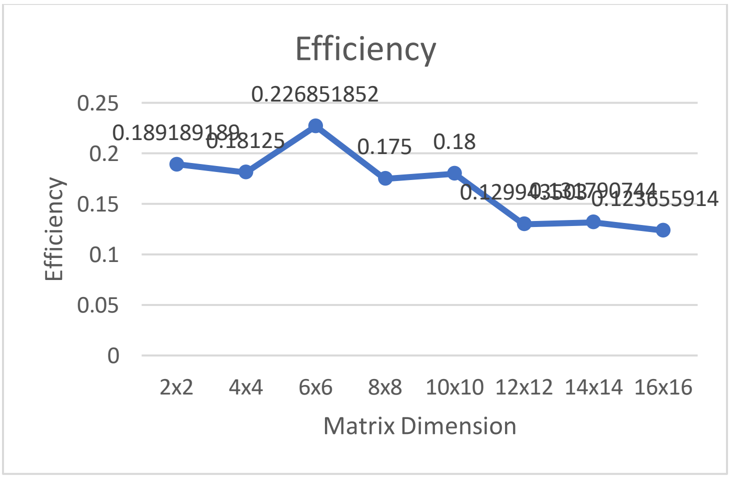


Figure 4

- Figure 1 shows the serial run time, parallel run time, speed up, and the efficiency of the system based on the varying parameters matrix dimension and number of processors.
- Figure 2 shows the speed up comparison between serial and parallel execution. It can be observed that serial program runs faster than the parallel program up to the matrix dimension 4x4. The reason behind this observation is that the communication time between the processors for parallel execution nullifies the advantage of the parallel system. This overhead time increases the overall time.
- From the Figure 2, it can also be observed that if the matrix dimension increases over 4x4, the parallel program takes less time as compared to the serial system. The reason behind this observation is that the parallel system divides the workload among all the processors. All of these processors works in parallel and save the time.
- Figure 3 shows that the speed up increases when the matrix dimension increases. It is because of the fact that the speed up is directly proportional to the serial time and the serial time is increasing with the increase in matrix dimension.
- Figure 4 shows that the efficiency of the system decreases with the increase in the matrix dimension. We know that the efficiency of the system is directly proportional to the speed up and inversely proportional to the number of processing elements. Here the speed up is not increasing in equal of greater proportion as compared to the increase in the proportion of the number of processing elements. Hence, the efficiency decreases.

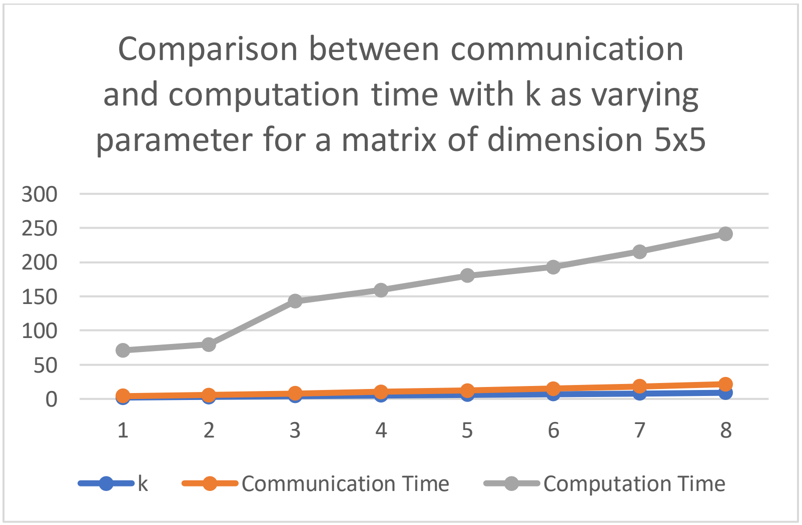


Figure 5

- Figure 5 shows that, if we fix the matrix dimension and increase the value of 'k', both the computation and communication time will increase. This observation is because of the fact that if we increase 'k', we are increasing the workload on each and every processor. Hence, the time increases.

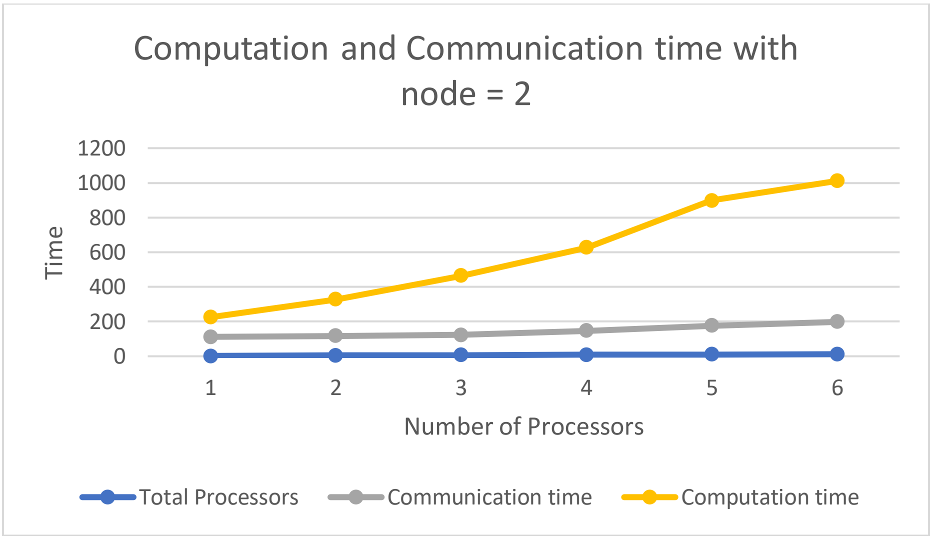
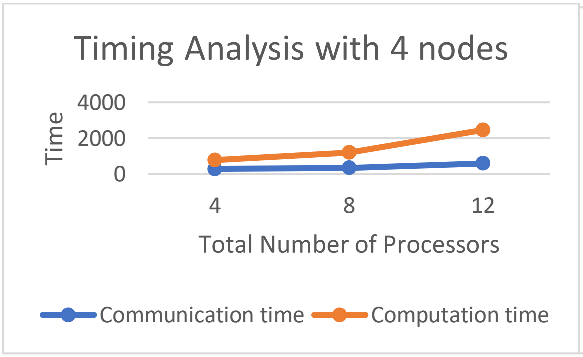


Figure 6

- Figure 6 shows the communication and computation time when the node is increased to 2. From figure 2 and figure 6 it can be observed that the communication time increases with the increase in number of nodes. This is because of the fact that now, the processor on one node needs to communicate with the processor at the other node.
- For n=2, the computation time also increases as compared to the computation time for n=1. The reason behind this result is that, due to the increase in the communication time for n=2 the computation time also increases.



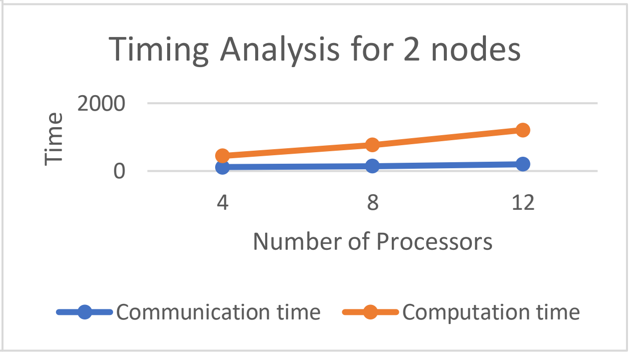


Figure 7 ***Figure 8***

- From figure 7 and figure 8 it can be observed that if we increase the number of nodes (keeping total number of processors as same), the computation time increases because of the increase in the communication time between the processors.

Chirag Soni: 665930262
Sanamdeep Singh Anand: 677787570

LESSONS LEARNT:

- In this coding assignment, we have learned how to map matrix multiplication on different types of topologies in parallel. This helps to solve the problem in less time.
- This assignment helped us to understand how to create virtual topologies and how to use these topologies for communication between the processors.
- From the conclusion section we have learnt that, though, the parallel system decreases the time to run the application, the time increases if we increase the matrix size.
- Time also increase if we increase the number of nodes due to the increase in the communication time.
- By calculating the speed up and the efficiency, we have learnt the best combination of the various parameters such as the matrix size, k, number of processors, and the number of nodes to run the application most efficiently.
- Tradition matrix multiplication is a bad example for parallel systems because the performance of the parallel system is almost same as the serial one with many processors.