Performance Results of MPI over Various Network Parameters

Ayse Betul Buyukkurt
abuyukku@cs.ucr.edu

Ram Sewak Sharma
rssharma@cs.ucr.edu

Abstract

Parallel Programming on networks/clusters of workstations has gained much attention during the past decade due to their being very easy to build, having large memory capacities, readily available cheap clock cycles and due to the performance and quick response time needs of many different types of emerging scientific applications. Message passing interface paradigm emerged as a solution to how to program the COWs and NOWs. This paper analyzes the performance of MPI over a network of workstations through running test programs with different network parameters. Our parameters are tuned such that the computation vs. communication overhead boundaries is determined.

1. Introduction

Parallel programming refers to the concept of using multiple computers or computers with independent internal processors, to solve a problem at a greater computational speed than using a single computer [1]. The effective speedup obtained is constrained by the sequential portion of the program. Even if one assumes that the communication overhead among the processors is negligible, the effective speed is inverse of (s+(1-s)/n) where s is the sequential portion of the program and the rest is amenable to parallelization. The speedup is further constrained by the overhead of inter-processor communications. However, it has been observed that computationally intensive programs, especially scientific applications, are parallelizable and it is possible to exploit the parallel programming concepts and techniques.

Then comes the issue of organizing the processors to work in parallel. One approach is to connect a number of processors through some common communication mechanism in a single box. The other approach is simply to connect a number of computers. The objective of this paper is to discuss the performance issues of network of workstations (NOW). A NOW can be a heterogeneous environment formed of computers of different types (PCs, Unix, etc…) and configurations. Computers are connected to each other over a network, where hosts communicate with each other through exchanging messages. This overall constitutes the message passing programming model.

The rest of the paper is organized as follows: We explain the message-passing interface in the next section followed by our project description in section 3. We provide a brief comparison between shared memory and the message-passing programming models in section 4. Section 5 explains our implementation process and section 6 provides brief explanation on our preliminary selection of test programs that we chose. Finally, section 7 discusses our results and section 8 concludes the paper.

2. What is MPI?

MPI stands for Message Passing Interface. Initially, various computer vendors came up with their own implementations of the message-passing paradigm. However, the efforts to develop a standard and portable model were felt as early as 1992. A group comprising of most of the manufacturers and researchers was formed in 1992 to develop an MPI standard. During the period 1992-1995, there were extensive meetings and workshops and an MPI document version 1.0 was brought out. It was further extended during the coming two years when version 2 was brought out in April 1997. The current MPI standard is sufficiently portable, efficient, and flexible standard for message passing.

MPI is a library-based approach, where all the MPI related constructs, data structures and functions are implemented in a library. The main advantage is it's ease-of-use. It comes with its own compiler and commands that can execute one program on multiple machines, check the status of them, kill processes if necessary, etc… All the high level routines in MPI are built upon lower level message passing routines, which abstracts the programming details of low-level tedious network communication from the programmer and replaces such pieces of code with its simple one line well-tested, efficient MPI communication constructs. MPI is also portable, which helped it became a message passing standard. The definition of this standard later provided vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases provide hardware support for to enhance scalability [2].

3. Project Objective

The main goal of this project is to make a series of time measurements on MPI processes distributed over a network of workstations to see what are the computation-communication boundaries for a series of applications. The measurements will be collected for a range of network and problem sizes.

We believe that in parallel applications, where inter processor communication is unavoidable, the design of the algorithm plays a crucial rule on the performance of the systems. For message passing environments the communication time is the bottleneck. If time spent on communication grows larger

than the time spent on computation, one can even easily observe a slow down instead of a speed-up. Furthermore, since granularity of the exploited portion of the available parallelism in a program determines both the amount and the frequency of inter process communication, the way the partitioning of the tasks is done also affects the performance of the algorithm.

We hope that this study gives more insight to the reader about varying effects of parallelism approaches on the performance of parallel algorithms. This measurement analysis paper looks into the following questions in particular: How to get an idea of effective resources in a parallel multi-user environment? How to better parallelise a given program for a NOW, when there exists the computation/communication tradeoff? In how fine granularity should one divide and assign a problem to a set of processors?

4. Shared Memory vs. the Message Passing Paradigm

When a task is divided among a number of processors, there are two ways in which the processors can communicate among themselves. The first one is through writing to a shared memory area, where the processors access through a kind of shared bus architecture, be it a shared bus, crossbar or a mesh. The communication between the processors is thus implicit and transparent to the programmer who does not have to make any efforts to code the inter-processor communications. These systems are called shared memory systems.

The alternative to using shared memory is through maintaining explicit communication among processors. If a processor P1 want to communicate with another processor P2, P1 has to form a packet enclosing the data and then send this packet explicitly through whatever the physical medium of communication among the processors exists. Such systems are named message-passing architectures.

From the viewpoint of architectural complexity, the shared memory systems leverage conventional architecture better, since existing processors can be added to the shared bus easily. However, message-passing systems are easier to build because the system is equivalent to just connecting a few computers together and there are fewer assumptions on the model. However, this requires the program to be re-written due to explicit interface to the communication. On the

other hand, ensuring correct communication is entirely the task of the programmer in the message passing systems. Implementing communication protocols for such systems can be very complex, error-prone and also expensive and slow.

From the viewpoint of scalability, the shared medium of communication among various memory locations (normally the shared bus or mesh) becomes an important bottleneck if the communications increase among the processes running on different processors, as the number of processors increase. Hence the shared memory system is not scalable. On the other hand, the message passing systems suffer from communication delays, which are much larger than those in the shared memory models due to the explicit message passing.

5. Project Implementation

The implementation of our project got started by installing the MPI on couple host machines. The installation took a great portion of our limited time due to the fact that the program did not want run at first due to network security settings which got resolved later after having the systems administrators open one of the labs for MPI use.

We have collected our measurements over twenty machines with the following configurations: All of them were over 1GHz in speed. Unfortunately, the amount of memory they contain is not as high as desired, however they are all 32Megs and above. All were PCs running Linux, emulating Windows through the vmware software.

The version of MPI we use is the latest available version, namely LAM version 6.5.6. LAM stands for Local Area Multi-Computer. It is an MPI programming environment and features a full implementation of the MPI communication standard. It allows working with heterogeneous computers on a network. With LAM, a dedicated cluster or an existing network computing infrastructure can act as one parallel computer solving one problem. LAM is also a development environment for MPI; it features extensive debugging support in the application development cycle and peak performance for production applications [4].

During this study we also observed the following important sides of MPI that we think needs to be mentioned in couple lines. First of all, MPI does its thread distribution statically. It expects the user to specify a list of hosts and to specify the number of processors that the user wants to run his/her code on at each run. The latter number is a logical number, whereas the first one being the list of the physical boxes connected to the NOW. MPI has the ability to get the count of actual processors in each of the boxes.

When an algorithm is executed over MPI, MPI first assigns the first job to the first processor in the hosts list, second job to the second processor, third one to the next and so on… If there are no processors left in the host list, MPI walks through the same list again spawning a separate thread at each processor till the user's number of logical processors is reached.

6. Test Programs

Our initial aim was to perform our measurements using one of the publicly available parallel benchmarks, however in response to our intensive searches we found most of the links to such sites being down. We then chose to work on two of our own MPI programs from one of our past studies. Those programs are explained in detail in the coming two subsections. The following two programs are downloaded from a scientific computing teaching site and again explained in the subsequent two subsections.

6.1 Ping-Pong

Ping-Pong is a well known entirely communication intensive test/benchmark program for parallel systems, either it be testing of parallel programming models or parallel architectures. In this application each machine sends a dummy packet to its neighbor(s), and receives another such packet for each packet it sends. This application is mainly useful for testing whether the logical communication link between any two processes/processors is up and running. Ping-Pong type applications are also helpful in verification of correctness of communication routines of message-passing models. Testing the routines on a live and running link does the latter.

Our Ping-Pong algorithm works as follows. Application takes two parameters: the message length and message count. The algorithm requires that every process know its id. Id is an integer number that is assigned by LAM/MPI to every single processor during its startup. Inside the application, using MPI specific commands every process learns its id by checking the id of the processor.

Every process with an even id sends a message of size “message length” to their neighbor processor with an odd id. The processors with odd ids, upon receiving the packet, reply back to the sending process with a packet of same length. This communication is done in parallel between all machines in the setting and for a number “message count” of times.

In this study the measurements are done on the network for the following variants: processor pool sizes 2, 4, 8 and 16; message lengths 100, 1000 and 10000 bytes; and message counts 2, 4, 8, 16, 32.

6.2 Gauss-Seidel with Successive Over Relaxation

This is a very computation intensive problem. Another name for the problem is the Heat Diffusion problem. It relates to areas like heat transfer, molecular biology and medicine. Normally, following differential equation governs the rate of heat transfer.

The basic model of a diffusion problem is as follows: This application considers a metal sheet with known constant temperature values on its three edges, where the goal is to find out the temperature values for the interior points of the metal sheet. Implementation views the metal sheet as fine a grid of points. As a result, we implemented the sheet as a two dimensional array. The problem is partitioned into separate tasks by dividing the grid (or the matrix) into horizontal strips. Each processor is then assigned a strip and programmed to perform the following calculations on each point of their portion of the grid sequentially:

For each point in the given 2D data set, a processor p takes the half of its value and the summation of the temperature values of the its four immediate neighbors, then performs the following computation and moves onto the next point in the data set.

$$A[i][j] = A[i][j]/2.0 + (A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1]) / 8.0;$$

The algorithm runs till the sum of differences between the latest values of all the points and the previous iteration's reaches a given precision. Program runs up to couple thousand iterations depending on the precision desired.

Inside the program the main communication that takes place is when the processors exchange the data values of their border rows with their neighboring cells prior to computing the new values of the interior data points of their own stripes. The second communication that exists in the program is where all processors send their difference values to the master where the total difference is compared with the desired total precision. The former is done per iteration whereas the latter once per ten iterations.

6.3 Interacting Particles

Interacting particles is another communication intensive MPI program. It is an interesting and quite fast algorithm for evaluating interactions between many particles. The list of particles is visible to all the processors through an array, where each particle is represented with a data structure having x, y and z coordinates and mass of the particle as its members. The length of this array is fixed and each process is responsible of finding out the number of particles itself is going to look after. Each particle has also a velocity, captured by another velocity structure having x, y and z components of the velocity as its members.

Firstly, each process sends the number of particles it is going to look after. This information is kept in an array of integers (counts) whose length is equal to the number of processes participating in the operation. This information is shared among all the processes. In other words, each process knows as to how many particles other processes are going to look after. From this, each process finds out its own offset in the original array of particles. This information is kept in an array called displacements. Also, each process knows the total number of particles being handled by all the processes.

Another array (offsets) is also maintained, so that each process communicates its own offset to the master. This is derived from the previous step of calculating the offsets. After this step, each process initializes the position and weight of all the particles it is going to work on from the original array particles, whose copy is with every process.

Then some computation takes place. Each process first initializes the velocity of each of the particles in its charge to zero. Then it computes forces, which all other particles in the system exert on it and the effect, which these forces will have on the velocity of the particle in question according to the following formula.

$$\begin{aligned} \boldsymbol{F}_{ij} &= \frac{m_i m_j}{(\boldsymbol{r}_i - \boldsymbol{r}_j)^2} \frac{\boldsymbol{r}_i - \boldsymbol{r}_j}{|\boldsymbol{r}_i - \boldsymbol{r}_j|} \\ \boldsymbol{v}_i &\leftarrow \boldsymbol{v}_i + \boldsymbol{F}_{ij} \end{aligned}$$

The order of magnitude of this computation is O(N*m) where N is the total number of particles and m is the number of particles in charge of the processor under consideration. In our example, we equally divide the particles among all the processors. Hence if n is the number of processors then we have m = N/n. Therefore, each processor's computation is in the order of $O(N^2/n)$. Thus for a given N, the computation amount in this phase decreases linearly to the number of processors. Though the formula for the calculation of forces on one particle by all others is given by the usual Newtonian formula, its correctness is not the objective. The only objective is to demonstrate the operation of MPI in computation and/or communication intensive situations. Each process takes its own time in computing the velocities of its own particles and we use the synchronization device MPI_Barrier that returns only when all the processes have completed their job. The time elapsed will be the time taken by the slowest process among all the processes.

Once the computation completes, the new updated states for all particles are exchanged amongst all participating processes. This action is done by the MPI function MPI_AllGather where each process sends its own contribution and it is collected and received by all the processes. It is like, each process putting its contribution in a slot of some structure and at the end of each process having contributed, the final result is received by all the processes.

6.4 Matrix Multiplication

Our final program is a matrix multiplication application. This program's significance is in that it illustrates an important parallel programming technique which is often referred to as a job queue or a bank queue paradigm. The basic idea behind the program is as follows: For a number of jobs which are independent of each other, a queue of pending jobs is maintained at the master's end. The master allocates the jobs to slaves who start working on the jobs as soon as they get the job and return the result to the master. The master, upon receiving a result, sends a new job to the slave. The processes terminate when the job queue becomes empty. In the intervening period, the master does not contribute to the computation.

There are a number of interesting observations we have made on this program. The first is that if the granularity of the task being assigned to the slave processes is very small, it not only results in more number of communications but also makes the master as a bottleneck. If the number of slaves are very large and the tasks do not take long, they are going to trouble the master too often and they may have to wait because the master has to communicate with so many slaves sending them jobs.

The code for the program involves dividing the matrix multiplication process among the slaves. The objective is to multiply two matrices A and B. The master process first broadcasts matrix B to all the slaves. Then the master begins at the first row of the matrix A and sends it to all the slaves, one row to each slave. Each slave works on the job of multiplying the given row to B so as to produce the corresponding row of the result matrix. Once it has computed the result, it sends the result back to the master. The master, while receiving the result checks whether there are some more rows to go. If there are, it sends one more row to the slave.

6.5 The Diffusion Problem Again

We used the same problem described in section 6.2 here again, however the code used to test code is different. This time our goal is to bring up a very important issue of choosing an optimal

topology for dividing the problem, given a fixed number of processors. We expected that this change would affect the communication time substantially.

To explain the idea with an example: Suppose the metal sheet that the problem is considering is square in shape, whose dimensions are 16x16 units. Supposing we have 16 processors, there are a number of ways in which the problem can be divided. If we assume that each processor has an equal computing capacity and can handle an equal amount of work, some of the possible divisions are: 4x4, 8x2 and 16x1. Below the sample logical topologies are illustrated.

First layout:

4x4 (1,1,)	4x4 (2, 1)	4x4 (2,1)	4x4 (1,1)
4x4 (1,2)	4x4 (2,2)	4x4 (2,2)	4x4 (1,2)
4x4 (1,2)	4x4 (2,2)	4x4 (2,2)	4x4 (1,2)
4x4 (1,1)	4x4 (2,2)	4x4 (2,2)	4x4 (1,1)

Second Layout:

2x8 (1,1)	2x8 (1,1)
2x8 (1,2)	2x8 (1,2)
2x8 (1,2)	2x8 (1,2)
2x8 (1,2)	2x8 (1,2)
2x8 (1,2)	2x8 (1,2)
2x8 (1,2)	2x8 (1,2)
2x8 (1,2)	2x8 (1,2)
2x8 (1,1)	2x8 (1,1)

Third layout:

1x16 (0,1)
1x16 (0,2)
1x16 (0,2)
1x16 (0,2)
1x16 (0,2)
1x16 (0,2)
1x16 (0,2)
1x16 (0,2)
1x16 (0,2)
1x16 (0,2)
1x16 (0,2)
1x16 (0,2)
1x16 (0,2)
1x16 (0,2)
1x16 (0,2)
1x16 (0,2)

In each of these possible layouts, each processor is going to look after the same number of elements, however the inter-process communication pattern is going to be different in each case. In this application processors exchange with their neighbors their border values. For example each processor in the first row has only one downward neighbor. Similarly, each processor in the bottom row also has only one upward neighbor. The rest of the processors will have both upper and lower neighbors. Similarly, each first/last column processor will have to

exchange its column data only with one right/left neighbor and the rest of the processors will perform the exchange operation both with their left and the right neighbors. The figures in brackets indicate the number of exchanges that needs to take place per iteration. Therefore, for layout one, the total number of row and column exchange operations performed in a single iteration has to be 48. For the second layout, the same number drops to 44, i.e. 28 row and 16 columns exchanges. Finally, for layout 3 the total number of exchanges is only 30 all being row exchanges, no column ones. The difference in the amount of data transferred in each of the layouts is considered not to be so significant.

7. Results

We collected a series of measurements for all of the above listed programs at various network and program sizes. Below presented are the measured data and our analysis.

7.1 Ping-Pong

In this program the machines in the network got paired up with their neighbor machines and exchanged fixed size message for a fixed number of times. The same process is repeated in parallel for various network sizes, message lengths and message counts. The results are listed in Table-1. The table demonstrated that as the message counts increase the total communication time differences for the same message counts for different processor pool sizes disappear. This made sense to us, since the initial pairing ups helped the communication get distributed over the network evenly, not generating hot spots, where processors would have backed off due to many message crashes.

Secondly, as message counts increased, an increase in the total communication time is observed. This was well matching to our initial expectations. Finally, we related the few irregularities in our results, such as the peak communication time value when the pool size is 4, message size is 1000 and message count is 2, due to the hidden load on the network by other simultaneous users.

Processor Pool Size	Message Size (In bytes)	Communication Time (sec)	Bandwidth (bytes/sec)
Message Count = 2			
2	100	0.000579	690846.875000 b/s
4	100	0.000738	541987.250000 b/s
8	100	0.001055	379146.125000 b/s
16	100	0.001200	333344.250000 b/s
Message Count = 8			
2	100	0.001786	895859.875000 b/s
4	100	0.001960	816310.250000 b/s
8	100	0.001866	857402.125000 b/s
16	100	0.002588	618230.000000 b/s
Message Count = 16			
2	100	0.003404	940064.625000 b/s
4	100	0.003515	910382.750000 b/s
8	100	0.003430	932942.187500 b/s
16	100	0.003790	844349.062500 b/s
Message Count = 32			
2	100	0.006643	963411.875000 b/s
4	100	0.006787	942970.625000 b/s
8	100	0.006899	927670.812500 b/s
16	100	0.006667	959966.562500 b/s
Message Count = 2			
2	1000	0.001265	3161933.000000 b/s
4	1000	0.005348	747947.750000 b/s
8	1000	0.001540	2597494.250000 b/s
16	1000	0.002013	1987115.500000 b/s
Message Count = 8			
2	1000	0.004476	3574659.250000 b/s
4	1000	0.004636	3451303.250000 b/s
8	1000	0.004563	3506393.500000 b/s
16	1000	0.006524	2452495.500000 b/s
Message Count = 16			
2	1000	0.008692	3681535.250000 b/s
4	1000	0.008903	3594283.250000 b/s
8	1000	0.008818	3628928.250000 b/s
16	1000	0.009110	3512633.500000 b/s
Message Count = 32			
2	1000	0.017070	3749255.750000 b/s
4	1000	0.017968	3561899.750000 b/s
8	1000	0.017173	3726795.000000 b/s
16	1000	0.017571	3642371.000000 b/s

Message Count = 2			
2	10000	0.004802	8329882.500000 b/s
4	10000	0.006608	6053260.000000 b/s
8	10000	0.006774	5904871.500000 b/s
16	10000	0.005813	6881125.500000 b/s
Message Count = 8			
2	10000	0.017832	8972673.000000 b/s
4	10000	0.021239	7533296.500000 b/s
8	10000	0.021338	7498364.500000 b/s
16	10000	0.019359	8264893.000000 b/s
Message Count = 16			
2	10000	0.035202	9090386.000000 b/s
4	10000	0.048275	6628690.500000 b/s
8	10000	0.048543	6592097.000000 b/s
16	10000	0.039597	8081413.500000 b/s
Message Count = 32			
2	10000	0.069876	9159088.000000 b/s
4	10000	0.07562	8463363.000000 b/s
8	10000	0.077726	8234052.500000 b/s
16	10000	0.073334	8727196.000000 b/s
Message Count = 64			
2	10000	0.139279	9190186.000000 b/s
4	10000	0.152192	8410429.000000 b/s
8	10000	0.157589	8122396.500000 b/s
16	10000	0.15535	8239461.000000 b/s

Table-1: Results for the Ping-Pong application

7.2 Gauss-Seidel with Successive Over Relaxation

This program happened to be our most computation intensive application. In order for it to run the code should allocate two quiet large floating-point arrays in the memory and perform lots of number crunching. Moreover, the algorithm has to run for at least a couple thousand times before it converges to a result so that it stops. For higher data sizes the algorithm took many hours before it finished. Therefore, for cases where long execution time was needed, we ran the programs overnight so that we do not suffer from other students booting up the machines in the middle of our test executions.

Unfortunately, we suffered from resource limitations in running this particular application. Our testing lab computers had 32MB of memory, which disallowed the processes from allocating large chunks from their respective local memories. Moreover, the lab machines were simultaneously being used by other students, which even limited our scarce resources more. Therefore our programs had to content for the memory and the processor clock cycles with other user's applications.

Total Time		**Data Sizes**				
		16	64	256	1024	4096
Processor Sizes	2	0.120402				
	4	0.074528	1.79461	116.502159	2865.681885	38141.48047
	8	0.054961	1.007641	39.770424	2041.858032	18694.53906
	16	0.021628	0.536837	19.13518	1025.312744	

Table-2: Gauss-Seidel Over Relaxation Results

Above table demonstrates the data we were able to collect. Blank cells indicate the cases where we were unable to take any measurements. We observed that only for the cases where either the processor pool size was too small to handle the data or the data size was too large to handle for some of the machines in the processor pool, we could not obtained any results. Later we concluded that both could be due to the hidden load in those particular machines.

Regarding the interpretation of the results, we constantly observed that the more the processor pool we increased, the faster the program became. This observation was in fact what we expected from this program, since for a computation intensive program an increase in the processor pool size means more computation power and therefore a steady decrease in the total execution times.

7.3 Interacting Particles

After presenting our results for almost one pure communication and one pure computation intensive applications, here are our results for the mixed cases. Below table demonstrates the results obtained for this problem. As it will be noticed, due to mixed nature of the application, we had the opportunity to measure the impact of communication and computation times separately.

We have collected the times taken by the program for a varying the number of processor and particle sizes. Initially when the number of particles is small such as 100, we observed that the total time of the process increases as we increase the number of processors for the simple reason that the computation is comparatively small while the inter-process communication increases as the number of processors is increased. However, when we increase the number of particles, the computation increased in the order of $O(N^2/n)$ whereas, the communication increased only linearly or even at a lesser order of magnitude. Therefore we observed that for this application for large data sizes, increasing the number of processors has a very positive influence on the total processing time. This reinforces a very important issue regarding the suitability of MPI paradigm to those situations where the ratio of computation to communication is large.

Processor Pool Size	Communication Time	Computation Time	Total Time
Particles : 100			
2	0.00037	0.00161	0.00198
5	0.00028	0.00118	0.00146
10	0.00045	0.00145	0.00190
20	0.00105	0.00247	0.00352
25	0.00137	0.00315	0.00452
Particles :1000			
2	0.00184	0.14746	0.14930
5	0.00258	0.06002	0.06260
10	0.00316	0.03069	0.03385
20	0.003	0.01667	0.01967
25	0.00179	0.01724	0.01903
Particles : 10000			
2	0.02461	15.44354	15.46815
5	0.02608	6.45975	6.48583
10	0.03053	3.12509	3.15562
20	0.03246	1.61395	1.64641
25	0.03267	1.23923	1.27190
Particles : 100000			
2	0.2981	2133.28258	2133.58068
5	0.8932	788.49873	789.39193
10	1.99668	393.20267	395.19935
20	4.40143	191.20233	195.60376
25	5.46514	303.9252	309.39034

Table-3: Interacting Particles Results

7.4 Matrix Multiplication using Job Queues

Through this test case, we aim to explain the principle of granularity of job assignment and its effect on the program execution times in message passing models. As it may be recalled, this application runs in a master slave setting, where the master process assigns processors new batches once they are finished. We modified the granularity of the task assignment by increasing the size of the job size per processor, while keeping the size of the problem fixed.

We initially tested the execution times on matrices of sizes 100, 500 and 1000. We noticed that the size of the computation of each row increased by $O(N^3)$, where N is the number of rows/columns in each matrix. We collected the below measurements.

Processor Pool Size	Computation Time	Total Time
100X100 Matrix		
2	0.000130	0.024999
5	0.000031	0.008975
10	0.000013	0.047736
20	0.000006	0.009494
25	0.000007	0.010054
500X500 Matrix		
2	0.003138	0.254336
5	0.000787	0.147377
10	0.000289	0.175565
20	0.000184	0.090167
25	0.000200	0.105486
1000X1000 Matrix		
2	0.011803	0.724094
5	0.003134	0.541909
10	0.001544	0.549290
20	0.000769	0.447617
25	0.000609	0.457178

Table-4: Results of the Matrix Multiplication application

As expected, the computation time decreases in direct proportion to the number of processors for any given problem size. However, the same behavior is not exhibited by the total computation time. If the results relating to the 1000X1000-matrix case are observed, for the 2 processors case there is only one master and one slave. For the processor pool size of 5, there are as many as 4 slaves. Assuming there was no bottleneck at the master's level and that all the slaves were working in parallel, then the total time should be inversely proportional to the number of processors. What we instead observed here is that the master starts getting bottlenecked by the message exchanges even at earlier stages when there are only 5 processors. That is the reason we have no improvement in the performance even when we have 10, 20 or 25 processors, though the computation times decrease. Therefore, the marginal improvement is only due to the reduction in the computation time at each machine.

Then we decided to increase the size of each job per processor by a factor of two and five by sending 2 and 5 rows at each batch to the slaves. The master still does the initialization of the matrix. We clocked the computation time for each of the slaves when it starts the computation after getting the job and summed it up. This allowed us compute the average computation time for each slave. On the other hand, we started the clock on the master just before the entire process starts, precisely right before it broadcasted the matrix B to all slaves and ended the clock, when it received the results from all the slaves after the queue length became zero.

Job Size per Processor	**Computation Time**	**Total Time**
5 Processors : 100X100 Matrix		
1	0.000038	0.009266
2	0.000036	0.005488
5	0.000051	0.004141
10	0.000038	0.004208
10 Processors : 100X100 Matrix		
1	0.000021	0.008939
2	0.000016	0.006625
5	0.000033	0.004332
10	0.000039	0.004200

Table-5: Matrix Multiplication Results: Impact of changing the granularity

Above are the results we obtained for the varying granularities. It is observed that the total time reduces when the number of rows per processors is raised to 2 and 5. However, no reduction is observed for batch sizes beyond 5 rows. The reason is that initially, there is acceleration due to large batch sizes, which utilized the processors better and allowed the master get less bottlenecked. However, once there was no bottleneck at the master's end, the increase in the job size ceases to have any impact on the total execution time.

7.5 The Diffusion Problem Again

The analysis given in section 6.5 indicates that the way you divide the jobs among processors makes a difference. We demonstrated this effect by running the diffusion program on a grid of 16x16, where we deployed 16 processors in all the three different topologies. Each processor only exchanges data with its neighbors and goes onto the next iteration. In our case, the processor is not doing any significant computation, since the main point, which we want to bring out is the impact of topology on the number of times exchanges and therefore the overall communication time. To serve as the stopping condition, we carried a fixed number of iterations (1000) for all three topologies. Below are our results.

Logical Topology	Total execution time (in sec)
4x4	6.6973
8x2	5.0231
16x1	2.8654

Table-6: Processor topology vs. total communication times

The results came as expected, since the 4x4 topology required the largest number of message exchanges, the time it took to finish the 1000 iterations came out the largest. Similarly, 16x1 processor topology required the les many exchanges, therefore the lowest overall time we measured verified the correctness of our expectations.

8. Conclusion

In this study, the performance of MPI for a NOW is measured and analyzed. The programs that we executed to test the performance of MPI, helped us to better analyze the impact of the following sets of metrics over MPI:

- Impact of communication for varying sizes of message size, message count and processor sizes in NOW environments
- Impact of MPI over computation only applications
- The performance boundaries of computation/communication mixed applications in message passing environments
- Impact of task granularity on processing time in an MPI environment
- Impact of changing the algorithm for the same problem in an MPI environment

In conclusion, throughout this study we tried to test and cover all aspects of MPI that we thought of and tried to measure the impact of message passing environments for all of them. We hope to have provided a complete coverage of the parameters that would impact the performance of MPI on a network of workstations.

9. References

[1] B. WILKINSON & M. ALLEN, *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice Hall, Upper Saddle River, 1999, ISBN 0-13-671710-1

[2] http://www-unix.mcs.anl.gov/mpi/

[3] http://beige.ucs.indiana.edu/B673/

[4] http://www.lam-mpi.org/

[5] Various LAM/MPI newsgroups for information on porting tricks