TO IMPLEMENT A PARALLEL COMPUTER SYSTEM AND STUDY THE PERFORMANCE GAIN ACHIEVED BY SUCH A SYSTEM UNDER VARIOUS CONFIGURATIONS AND FOR DIFFERENT TYPES OF COMPUTATIONS

Submitted in partial fulfilment of the requirements
for the award of the degree of

Bachelor of Technology
In
Computer Science Engineering

Guide:
Prof. Sunil Kumar Singh
CSE Department

Submitted By:

Aman	Madaan	10411502709
Ankur	Aggarwal	06511502709
Ankur	Dewan	10611502709
Ashwani	Gupta	03011502709

Bharati Vidyapeeth's College Of Engineering
A-4, Paschim Vihar, New Rohtak Road, New Delhi – 110063
Affiliated To
GURU GOBIND SINGH INDRAPRASTHA UNIVERSITY
(2009-2013)

DECLARATION

We hereby declare that the project entitled **“**To Implement a Parallel Computer System and Study the Performance Gain Achieved by Such a System Under Various Configurations and For Different Types of Computations**”** submitted for the B.Tech Degree is our original work and the project has not formed the basis for the award of any degree, associateship, fellowship or any other similar titles.

Signature of the Students:

(AMAN MADAAN)	(ANKUR AGGARWAL)
(ANKUR DEWAN)	(ASHWANI GUPTA)

Place: New Delhi

Date:

CERTIFICATE

This is to certify that the project entitled “To Implement a Parallel Computer System and Study the Performance Gain Achieved by Such a System Under Various Configurations and For Different Types of Computations” is the bonafide work carried out by

Mr. AMAN MADAAN (10411502709)
Mr. ANKUR AGGARWAL (06511502709)
Mr. ANKUR DEWAN (10611502709)
Mr. ASHWANI GUPTA (03011502709)

students of B.Tech (CSE), BVCOE affiliated to GGSIPU, during the year 2012, in partial fulfilment of the requirements for the award of the Degree of Bachelor of Computer Technology and that the project has not formed the basis for the award of any degree, diploma, associate ship, fellowship or any other similar title.

Signature of the Guide
Place: New Delhi
Date:

ACKNOWLEDGEMENT

Something as complicated as the project we have taken is seldom completed without support and guidance of the seniors ,This project was no exception and thus, we would like to thank **Mr. Alok Basu**, Head, Department of CSE and the **members of faculty at Department of CSE** at BVCOE for their encouragement, support and guidance. They instilled in us the confidence for taking up this project, and finishing it successfully.

Prof. Sunil Kumar Singh, helped us a mentor to the fullest. Apart from providing the systems for the project, our mentor gave us many wonderful ideas that helped in taking the project further, helped us in keeping with the schedules, encouraged us to add more and more features to this project and replied patiently to each of the gazillion e-mails that we have flooded his inbox with during the last 3 months. We thank him for that.

We would also like to thank **Mrs. Narina Thakur**. She did a wonderful job in handling the logistics for the mini projects of the third year students of CSE. She kept us informed about each and every deadline, and ensured that everything required for the presentations was in place.

Team Members
Aman Madaan (10411502709)
Ankur Aggarwal (065112709)
Ankur Dewan(10611502709)
Ashwani Gupta (03011502709)

ABSTRACT

To implement a parallel computer system and study the performance gain achieved by such a system under various configurations and for different types of computations. At this point we must distinguish between the multicore computers that we use nowadays and the parallel computer system that we have implemented. The difference is essentially that in former all the processors share the system resources like memory and system clock (tightly coupled), while in latter each machine is an independent entity on its own (loosely coupled). While prima facie it may appear that adding more systems to solve a problem will lead to a faster solution, the notion is only intuitive as was confirmed by our findings. Interesting results were obtained by varying the number of systems that are working together. The algorithms used for distributing the workload were also being of special interest.

The initial steps were implementation of the system and understanding the MPI environment.The total execution time is taken as the decisive criteria. We use functions provided by the MPI library itself to get the total running time.

```
startwtime = MPI_Wtime();

...

endtime=MPI_Wtime();

Total running time= endtime-starttime;
```

The running time was then plotted against iterations for various configurations. The speed up was calculated by dividing the runtimes of single node and other configurations.

- The Cluster Gives tremendous results for the embarrassingly parallel problems
- The "dumped" systems were able to beat the ultra-fast i3 processor
- The shared memories are way too fast as compared with the distributed memories
- The performance hit of the tune of 20 times is compared

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Time * Workers = Constant

One may say that this single equation is the seed over which the now dense and feature rich tree of parallel computing has evolved since the 60's.

It is simply put beautiful to see the systems work in tandem to produce the desired output.
The parallel programmer is the like a music director directing a mellifluous symphony, distributing the work among the nodes and getting the separate systems to achieve the common goal.

With cloud computing, multicore computing, distributed computing as buzzwords, parallel computing is attracting more limelight than it ever did.

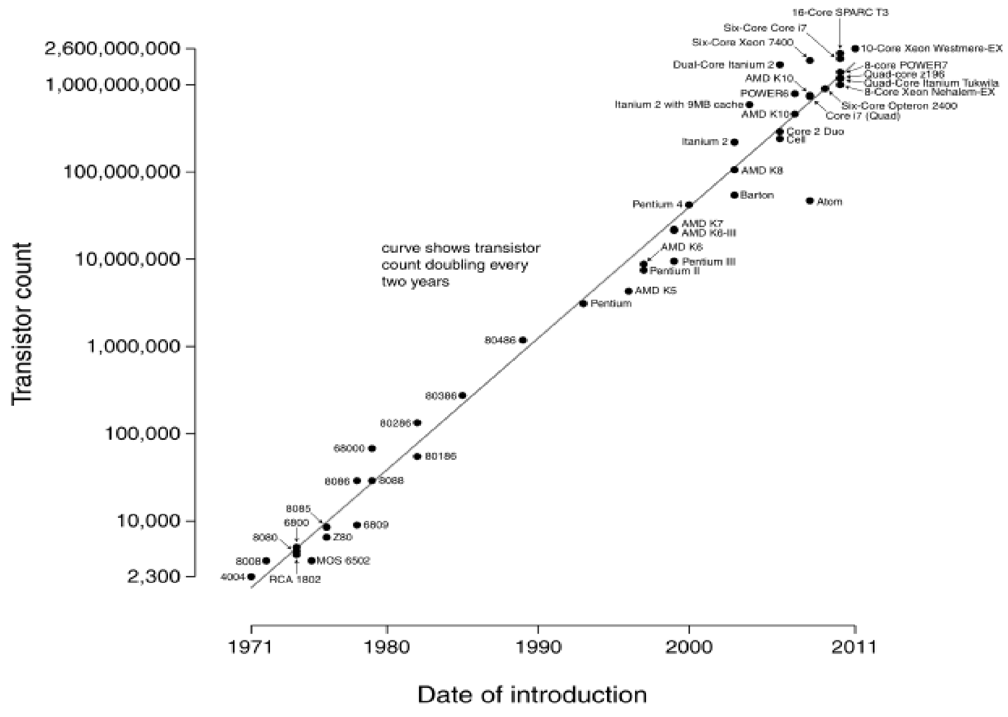


Figure 1.1 Graph for Moore’s Law

In this project, we implemented a distributed computing system, familiarized ourselves with the MPI model and authored some parallel programs. We got some shocking and some quite expected results.

1.1 Parallel Programming

Parallel computing is a form of computation in which a problem is divided into smaller ones, which are solved simultaneously. Broadly, there are 2 categories of parallelism: shared memory and distributed memory parallelism. Shared memory parallelism is exploited using the concept of threads. It is the ability for all processors to access all memory as global address space. Shared memory parallelism is generally used to solve sub tasks with the same task. Distributed Memory Parallelism is implemented by distributing the computations across different nodes (computers).

The computations are distributed by means of a message-passing library. The implementation used in this paper is called mpich2; it is a widely portable and free implementation of the MPI standard. MPI provides portability and performance for complex applications on a variety of architectures.

The approach is very similar to the divide and conquers algorithms. A large problem is broken down into pieces; each node is given a chunk of the piece to solve. The final solution is obtained by combining the individual results. A very common example is performing matrix operations quickly by distributing the parts of the matrix to the nodes to work on.

1.2 Junk Computing

As a branch of the project, we took on what is called junk computing.
The mankind is hushing down one of the biggest problems of the 21st century, the e waste. Millions of computers are manufactured each day, only to get outdated, and abandoned ultimately. The motivation of this research came from a myriad of computers that were lying idle in the labs of our college. We were curious to know whether these machines are as dumb

as they look, or do they have the flare left. Surely, we cannot expect any one of these Pentiums to fight the might Intel i-series generation processors, but as the old saying goes, one and one makes eleven. In this research, we compared the performance of some of these abandoned warriors and the results were surprising.

More and more organizations nowadays prefer implementing their own cluster for several reasons. Very often, there is a divide between buying new hardware and using the existing resources. The results from this research would help the organizations in deciding whether a cluster made from junkyard can satisfy the need, or do they need to invest in hardware.

This project aims to answer some of these questions.

1.3 Difference between Reducing the Complexity and Reducing the Running time

It is worth noticing that the performance gain is obtained because we execute more than one instruction in a single time unit. An algorithm that is O (n^2) will still run 10^4 slower if the input size if increased by a factor of 100. However, intuitive notions direct us to believe that an algorithm with a running time of T (n), when executed by P nodes working together, can be expected to finish in T (n)/P time under ideal conditions. Amdahl's law provide a more formal treatment of this intuitive notion. As expected, the factors that prevent us from getting the ideal speed up are

1. Communication Delays.
2. There may be some portion of the algorithm that is inherently parallel, so there may be a fraction of code that has to be executed serially, thus keeping us apart from the ideal speedup.

CHAPTER 2

PARALLEL COMPUTING MODEL

Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel"). There are several different forms of parallel computing: Bit-level, Instruction level, Data, and Task parallelism.

Parallel computing uses multiple processing elements simultaneously to solve a problem. This is accomplished by breaking the problem into independent parts so that each processing element can execute its part of the algorithm simultaneously with the others. The processing elements can be diverse and include resources such as a single computer with multiple processors i.e. Multicore system, several networked computers, specialized hardware, or any combination of the above.

Parallelism appears in various forms, such as look ahead, pipelining vectorization, concurrency, simultaneity, data parallelism, partitioning, interleaving, overlapping, multiplicity, replication, time sharing, space sharing, multitasking, multiprogramming, multithreading, and distributed computing at different processing levels.

2.1 Flynn’s Classification

Michael Flynn introduced a classification of various computer architectures based on notions of Instruction and Data streams. Conventional sequential machines are called SISD Computers. Vectors are equipped with scalar and vector hardware or appear as SIMD machines. Parallel computers are reserved for MIMD machines. An MISD machines are modelled. The same data stream flows through a linear array of processors executing different instruction streams. This architecture is also known as systolic arrays for pipelined execution of specific algorithms.

1 SISD uniprocessor architecture

2 SIMD architecture (with distributed memory)

3 MIMD architecture (with shared memory)

4 MISD architecture (systolic array)

2.1.1 SISD (Single Instruction Stream Single Data Stream)

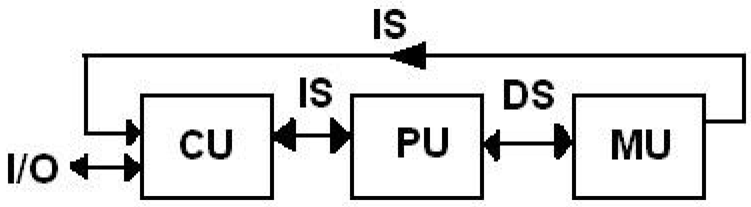


Figure 2.1 SISD

SISD is similar to single core processor, with one processor and one memory. SISD attends/executes only one Instruction at a time and only one data stream to be operated upon. It has only one Control unit, one Processing unit and one Memory unit/Input/Output is done by CU. Same Instruction Stream and Data Stream is shared by all of the elements present in the system.

2.1.2 SIMD (Single Instruction Stream Multiple Data Stream)

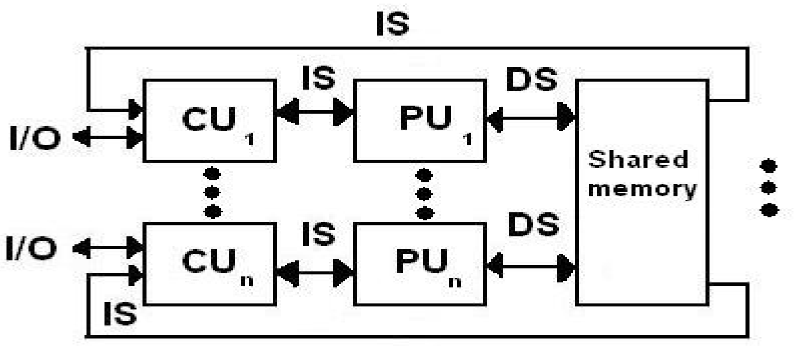


Figure 2.2 SIMD

SIMD has a common shared memory used by all the processing units. The instruction is same for all the processing units but they all operate on different set of data simultaneously. It is very similar to Multi core processor. Also this can be implemented as tightly coupled system where they have different processing units and common memory but each can individually have its part in memory.

2.1.3 MIMD (Multiple Instruction Stream Multiple Data Stream)

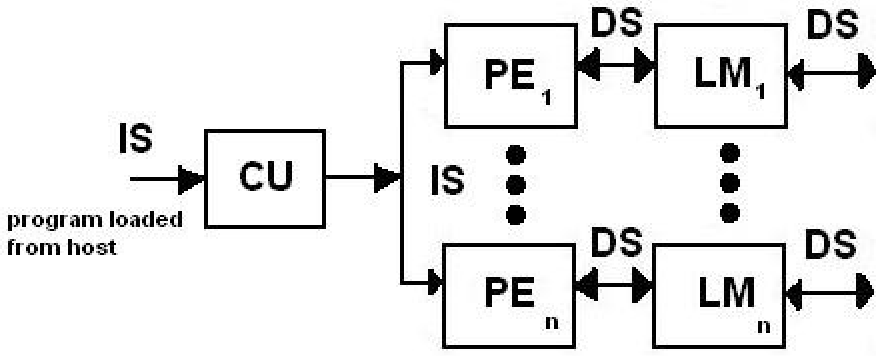


Figure 2.3 MIMD

MIMD is pure Supercomputer. Here multiple instructions are executed simultaneously over different set of data. Each processing unit has its own local memory. It is the fastest and strongest among all other forms of this classification.

2.1.4 MISD (Multiple Instruction Stream Single Data Stream)

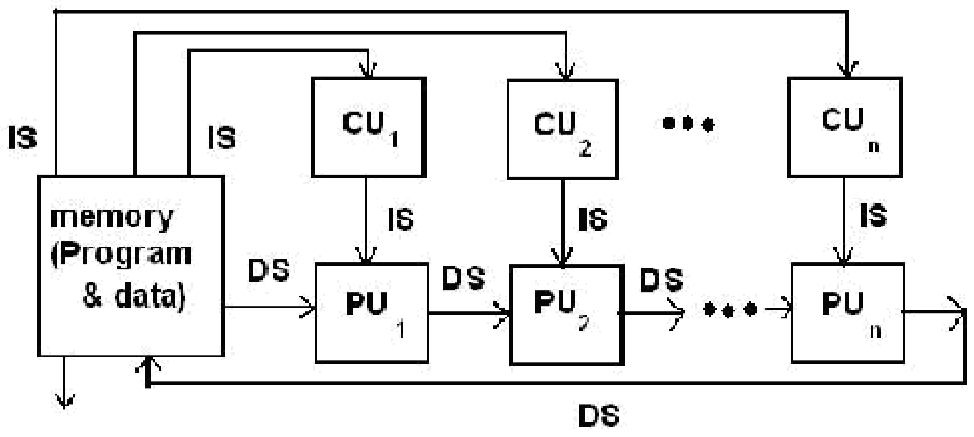


Figure 2.4 MISD

MISD is a theoretical model where we have multiple control units and multiple Processing units. Each processing units works different instruction on the same data stream. It also resembles the normal assembly scheduling in the factories where different machines works differently on the same thing one by one to make it full and complete.

2.2 Multiprocessors and Multicomputer

A multiprocessor is single computer with multiple processors or in other words tightly coupled system which may be multicore processor or multiprocessor on the other hand Multicomputer is collection of various computers. These physical models are distinguished by having a shared common memory or unshared distributed memories.

2.2.1 Shared-Memory Multiprocessors

There are three shared-memory multiprocessor models: the uniform memory access (UMA) model, the non-uniform-memory-access(NUMA) model, and the cache only memory architecture(COMA) model. These models differ in how the memory and peripheral resources are shared or distributed.

2.2.1.1 The UMA Model

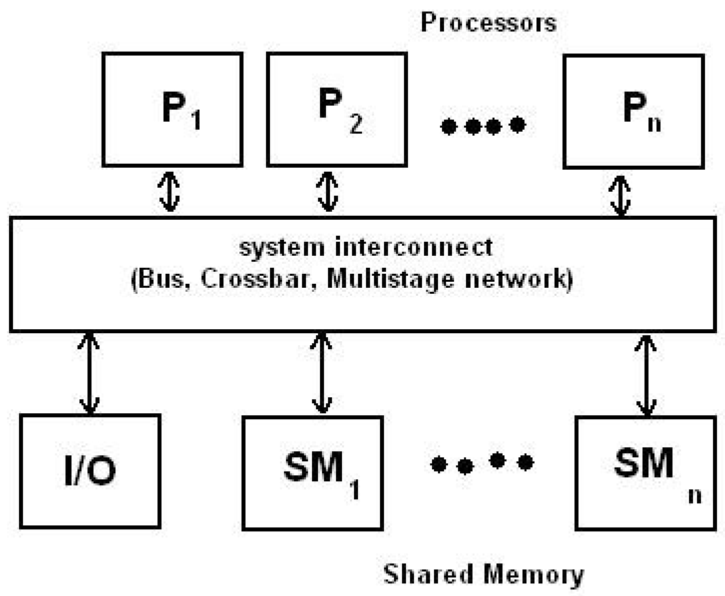


Figure 2.5 UMA

In a UMA multiprocessor model the physical memory is uniformly shared by all the processors. All processors have equal access time to all memory words, i.e. why it is called uniform memory access. Each processor may use a private cache. Peripherals are also shared in some fashion. Multiprocessors are called tightly coupled systems due to the high degree of resource sharing. The system interconnect takes the form of a common bus, a crossbar switch or a multistage network.

The UMA model is suitable for general purpose and time sharing applications by multiple users. It can be used to speed up the execution of a single large program in time-critical applications. To coordinate parallel events, synchronization and communication among processors are done through using shared variables in the common memory. When all processors have equal access to all peripheral devices, the system is called a symmetric multiprocessor.

In an asymmetric multiprocessor, only one or a subset of processors are executive capable. An executive or a master processor can execute the operating system and handle I/O. The remaining processors have no I/O capability and thus are called attached processors (APs). Attached processors execute user codes under the supervision of the master processor. In both multiprocessor and attached processor configurations, memory sharing among master and attached processors is still in place.

2.2.1.2 The NUMA Model

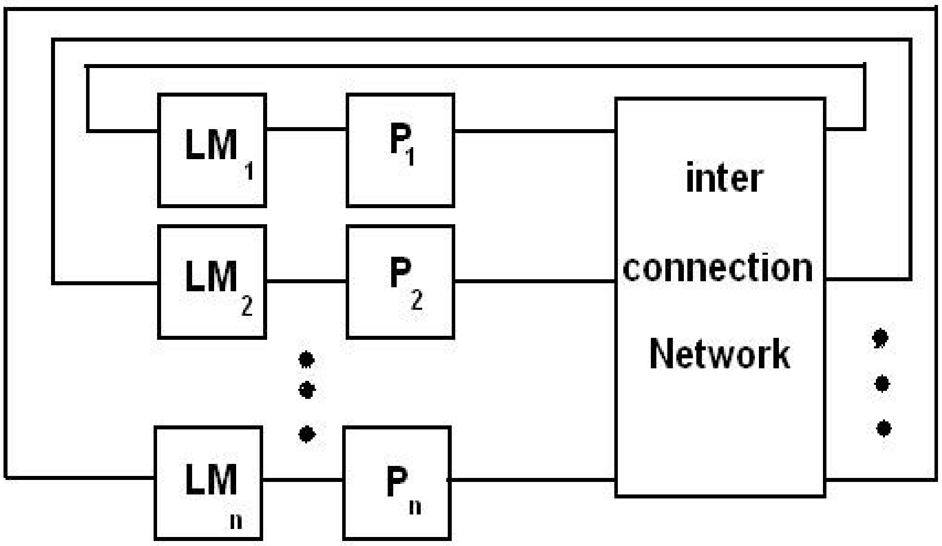


Figure 2.6 Shared Local Memory

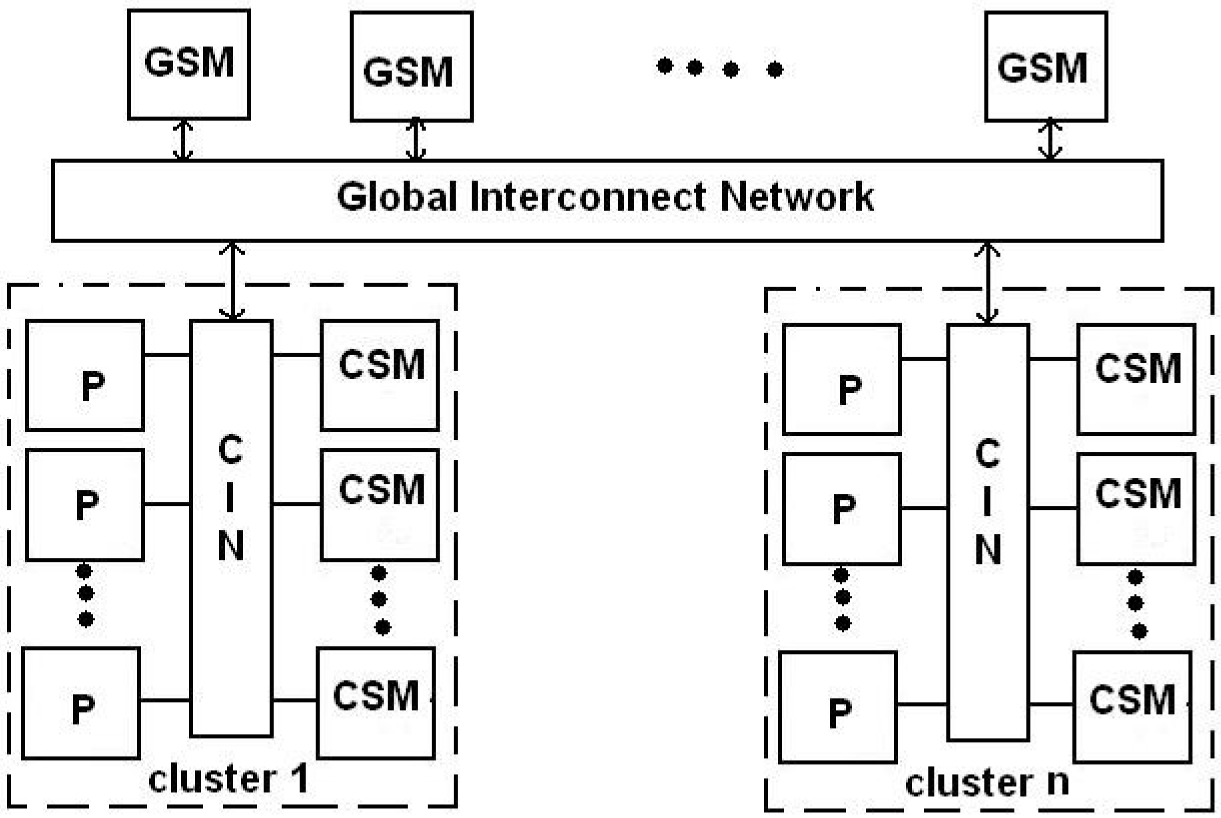


Figure 2.7 Hierarchical Cluster Model

A NUMA multiprocessor is a shared-memory system in which the access time varies with the location of the memory word. Two NUMA machine models are:

(a) Shared local memories

(b) A hierarchical cluster model

The shared memory is physically distributed to all processors, called local memories. The collection of all local memories forms a global address space accessible by all processors. It is faster to access a local memory with a local processor. The access of remote memory attached to other processors takes longer due to the added delay through the interconnection network.

Besides distributed memories, globally shared memory can be added to a multiprocessor system. In this case, there are three memory-access patterns: The fastest is local memory access. The next is global memory access. The slowest is access of remote memory. As a matter of fact, the models can be easily modified to allow a mixture of shared memory and private memory with pre specified access rights.

2.2.1.3 The COMA Model

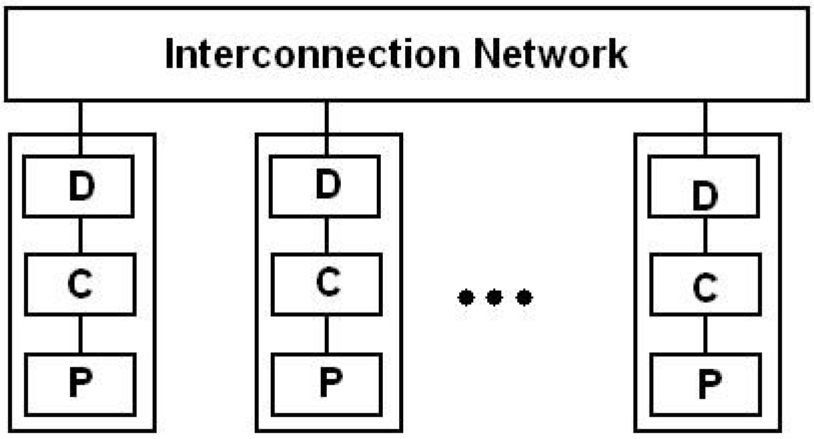


Figure 2.8 COMA

COMA model: A multiprocessor using cache-only memory assumes this model. The COMA model is a special case of a NUMA machine, in which the distributed main memories are converted to caches. There is no memory hierarchy at each processor node.

D: Directory

C: Cache

P: Processor

All the caches form a global address space. Remote cache access is assisted by the distributed cache directories. Depending on the interconnection network used, sometimes hierarchical directories may be used to help locate copies of cache blocks. Initial data placement is not critical because data will eventually migrate to where it will be used.

2.2.2 Distributed-Memory Multicomputer

A distributed-memory multicomputer system consists of multiple computers, often called nodes, interconnected by a message-passing network. Each node is an autonomous computer consisting of a processor, local memory, and sometimes attached disks or I/O peripherals.

2.2.2.1 NORMA

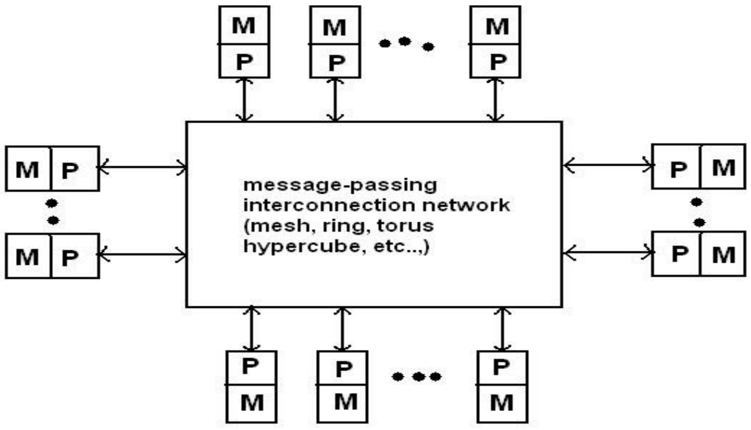

M
P
message-passing interconnection network (mesh, ring, torus hypercube, etc..)

Figure 2.9 NORMA

The message-passing network provides point-to-point static connections among the nodes. All local memories are private and are accessible only by local processors. For this reason, traditional multicomputers have been called no-remote-memory-access (NORMA) machines. However, this restriction will gradually be removed in future multicomputer with distributed shared memories. Internode communication is carried out by passing messages to the static connection network.

CHAPTER 3

DESIGNING PARALLEL PROGRAMS

To design parallel programs, a programmer needs think in a radically different way. The possibilities both of getting things done and making mistakes are multiplied.
The process begins with understanding the underlying architecture; the next step is the understanding of library functions that are used to distribute the tasks. Once these steps are crossed, a programmer thinks through the problem in hand and tries to develop a parallel algorithm. This is usually done by writing a serial algorithm and identifying the portions that can be executed in parallel. The final step in algorithm design is and then figuring out the partial results shall be combined to yield the desired output.

A detailed explanation of the steps mentioned above follows.

3.1 Multicomputer

A multicomputer comprises a number of von Neumann computers, or nodes, linked by an interconnection network. Each computer executes its own program. This program may access local memory and may send and receive messages over the network. Messages are used to communicate with other computers or, equivalently, to read and write remote memories. In the idealized network, the cost of sending a message between two nodes is independent of both node location and other network traffic, but does depend on message length.

A defining attribute of the multicomputer model is that accesses to local (same-node) memory are less expensive than accesses to remote (different-node) memory. That is, read and write are less costly than send and receive. Hence, it is desirable that accesses to local data be more frequent than accesses to remote data. This property, called *locality*, is a third fundamental requirement for parallel software, in addition to concurrency and scalability. The importance of locality depends on the ratio of remote to local access costs. This ratio can vary from 10:1 to 1000:1 or greater, depending on the relative performance of the local computer, the network, and the mechanisms used to move data to and from the network.

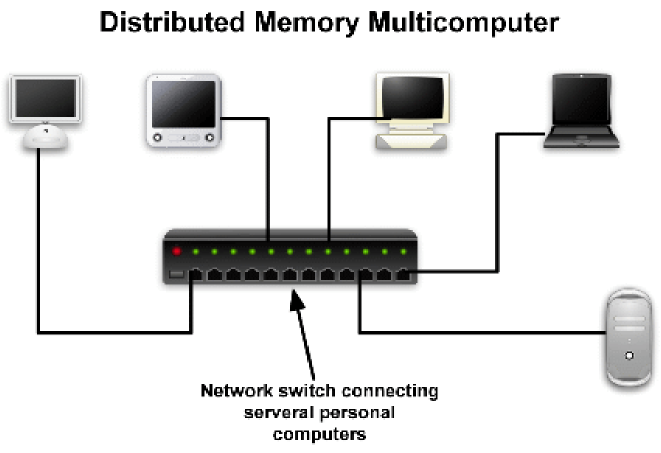

Distributed Memory Multicomputer
Network switch connecting serveral personal computers

Figure 3.1

The multicomputer is an idealized parallel computer model. Each node consists of a von Neumann machine: a CPU and memory. A node can communicate with other nodes by sending and receiving messages over an interconnection network.

3.2 Writing Parallel Programs using MPI

The chapter on MPI programming lists the various functions in detail. Here, the basics of writing a parallel program using MPI are described.

MPI programs follow what is called the SPMD model. The same program is executed on every machine. Though it may be very non obvious at first glance, the approach is justified once we compare the size of the programs and the gains achieved.

The simplest MPI program looks like:

```
#include <stdio.h>
#include <mpi.h>
#define ROOT 0
int main(intargc, char** argv) {
int myrank, nprocs;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if(myrank==ROOT)
{
printf("Special Greetings From The ROOT!\n");
}
printf("Hello from processor %d of %d\n", myrank, nprocs);
MPI_Finalize();
return 0;
}
```

Perhaps the best way to understand the MPI programming model is to compare it with the forkexecve call chain used to spawn child processes from a process in Linux.

The ways child processes are spawned are as follows:

1. pid_t childPID;

   We declare a variable to store process id of child.

2. childPID=fork()

   Calls fork(), fork returns process ID of the child process to the parent and 0 to the spawned child.

3. Both parent and the child execute the same code from this point onwards.

Now, to decide what to run in the parent, use:

```
if(childPID==0)
{
//this is child, replace it's memory with code of some other process
}
```

Similarly in MPI, every process runs **MPI_Comm_rank(MPI_COMM_WORLD, &myrank).** This fills the variable myrank with the rank of the process. After this, we can distribute work among the processes as usual. Usually, it is the root that finishes the initialization and the supporting nodes are given some subset of the problem to work upon. The usual work flow thus is:

```
if(myrank==0)
{
        //1. finish initialization
}
        //2. Distribute the work
MPI_Bcase()
MPI_Scatter()
MPI_Send()
        //3. Collect Results
MPI_Gather()
MPI_Recv()
        //4. Nodes solve their bit
if(myrank==0)
{
        //5. Integration of the sub-solutions
}
```

3.3 Parallelizing Algorithms

Calculating the sum to N terms

Serial Algorithm:

```
set sum=0
set i=1
while i <=N
repeat
sum<-sum+ i
i<-i+1
```

```
        end repeat
        print sum
```

Parallel Variant:

To parallelize this algorithm, we must try to find a way to create partitions of the set (1,2,...N) such that each partition has the same number of elements.

One way to achieve this is equivalence classes. For example, for 4 nodes, the division would be:

[0]={0,4,8,12,..., K1}

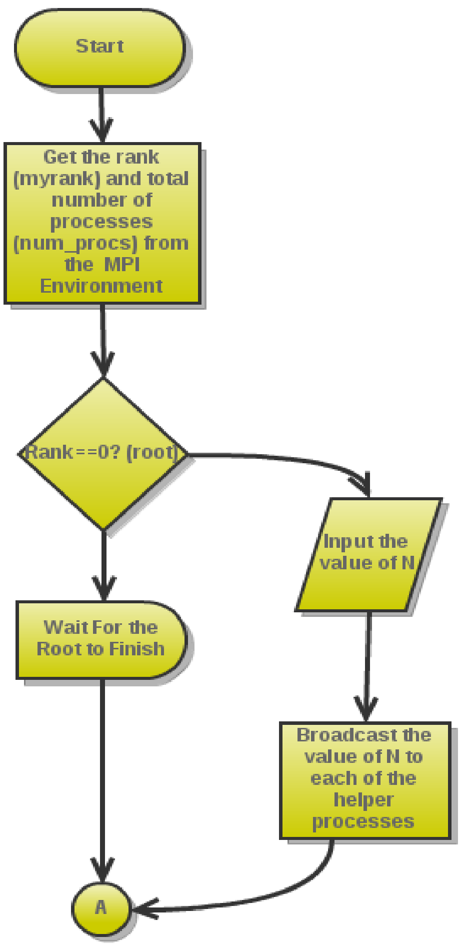
[1]={1,5,9,13,...,K2}

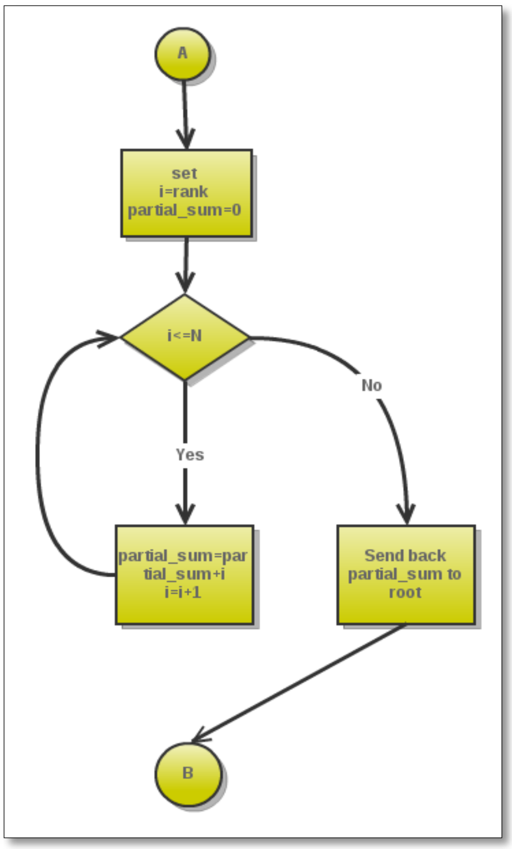
[2]={2,6,10,...,K3}

[3]={3,7,11,...,K4}

Where Ki is the maximum number before N in the respective class.

The following flowcharts neatly sum up the process:


Start
Get the rank (myrank) and total number of processes (num_procs) from the MPI Environment
Rank==0? (root)
Input the value of N
Wait For the Root to Finish
Broadcast the value of N to each of the helper processes
A



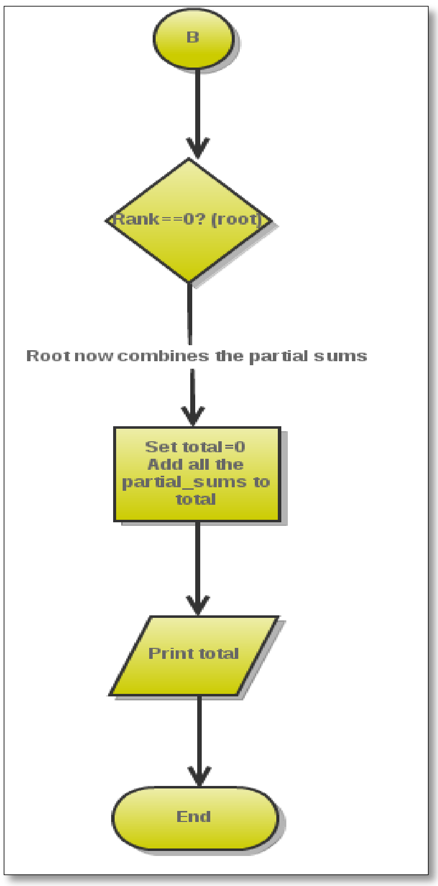

B
Rank==0? (root)
Root now combines the partial sums
Set total=0
Add all the partial_sums to total
Print total
End

Figure 3.2 Flowchart for Designing Parallel Program to Calculate Value of ∑N

CHAPTER 4

EVALUATION OF PARALLEL PROGRAMS

The execution of a program on a parallel computer may use different numbers of processor at different time periods during its execution cycle. Parallel programs are designed in such a way that the available processors are utilized to their maximum capacity and the efficiency of the overall system is high. This is achieved by striking a balance between the available hardware and software. The main objective is to exploit the features of pipelining and networking in such a way that the overall gain exceeds the communication latency between different processors and resource limitations, if any.

The evaluation of performance of parallel computing system is not only defined by the speed up factor but also depends on the balance between software parallelism and hardware parallelism, the system efficiency, utilization and quality of the parallel computation.

4.1 Performance Metrics

At any point of time the number of processors being used to execute the programs is the Degree of Parallelism and this value is used to study and compare the performance with different metrics for different algorithmic structure, resource utilization and run-time conditions. When the degree of parallelism exceeds the number of available processors the program has to be executed in sequentially running parts.

4.1.1 System Efficiency

Low value of system efficiency corresponds to the entire program being executed on a single processor. While maximum efficiency corresponds to all n processors fully used.

The total number of operations performed is O(n) and execution time is T(n) for an n-processor system. For a uniprocessor system T(1)=O(1) and for multi-processor system executing more than 1 instruction per cycle T(n)<O(n).

System Efficiency, $E(n) = S(n)/n = T(1)/(nT(n))$

Thus, for $1 \leq S(n) \leq n$, efficiency is $1/n \leq E(n) \leq 1$

4.1.2 Redundancy and Utilization

Redundancy is the extent to which software parallelism matches with hardware parallelism i.e. the balance between the two.

Redundancy, $R(n) = O(n)/O(1)$

Utilization is the percentage of available resources that are used throughout the execution of parallel program.

Utilization, $U(n)= O(n)/ (nT(n))$

4.1.3 Quality

Quality is the extent to which parallelization is achieved by the system. It is directly proportional to speedup and efficiency and inversely to redundancy.

Different levels of parallelism available in different types of programs:

Scientific Codes……………………..	High Degree of Parallelism due to data parallelism
Computation-intensive Codes……………………..	500-3500 arithmetic operations per cycle
Instruction level parallelism……………………..	Factor of 5-7 but reaches 17 instructions per cycle for ideal case
Superscalar processor system……………………..	2-5.8 instructions per cycle

4.2Standard Performance Measures

Standard Performance Measures are often stated in terms of MIPS but the real performance is always program-dependent or application-driven. This is due to the reason that MIPS depends on instruction-set and varies directly between programs, inversely with performance. Thus, a reference machine is taken and the performance for a system is defined by relative MIPS with respect to this reference machine.

Two standard measures to compare various computer performances are:

(A) Dhrystone Results: It consists of around 100 high-level languages and data types with no floating-point operations. Dhrystone statements are balanced with respect to statement type, data type and locality of reference. Thus, the rating is CPU-intensive benchmark and used to measure integer performance of computers.

(B) Whetstone Results: It is used for assessing floating-point performance and a benchmark for both integer and floating-point operations. These operations include array indexing, subroutine calls, parameter passing and conditional branching. The rating does not contain any vectorizable code.

4.3 Massively Parallel Processing

A machine that has thousands of processors is a massively parallel processing system. Massive parallelism can be exploited at instruction level and data level.

Parallelism at the instruction level is limited because rarely a parallel computer can execute more than two instructions per cycle for same program. Instruction level parallelism is restricted by program behaviour, compiler capabilities and program flow built into the system.

However, Data parallelism, situation where same operation is executed over large array of data, is much higher than instruction parallelism. It has thus been implemented in SIMD array processors and SPMD or MPMD multicomputer systems.

4.4 Relation Between:

(A) Machine Size and Workload

(B) Machine Size and Efficiency

A scalable system is a machine in which the number of processors (machine size) must increase proportionally with increase in problem size. The lines in the graph show the relation between machine size and the efficiency.

Linear function denotes the ideal case which if not achievable, the system must try to have sub-linear scalability. The efficiency curve for such a system follows the Gustafson's Law.

Constant function i.e. keeping the problem size unchanged is undesirable as the communication overhead reduces the efficiency rapidly. The corresponding efficiency curve follows Amdahl's Law and decreases rapidly.

If the system follows the exponential function then the system is poorly scalable as the problem size should increase faster than machine size to overcome memory or I/O limits. Such a system is not implementable due to storage resource limitations.

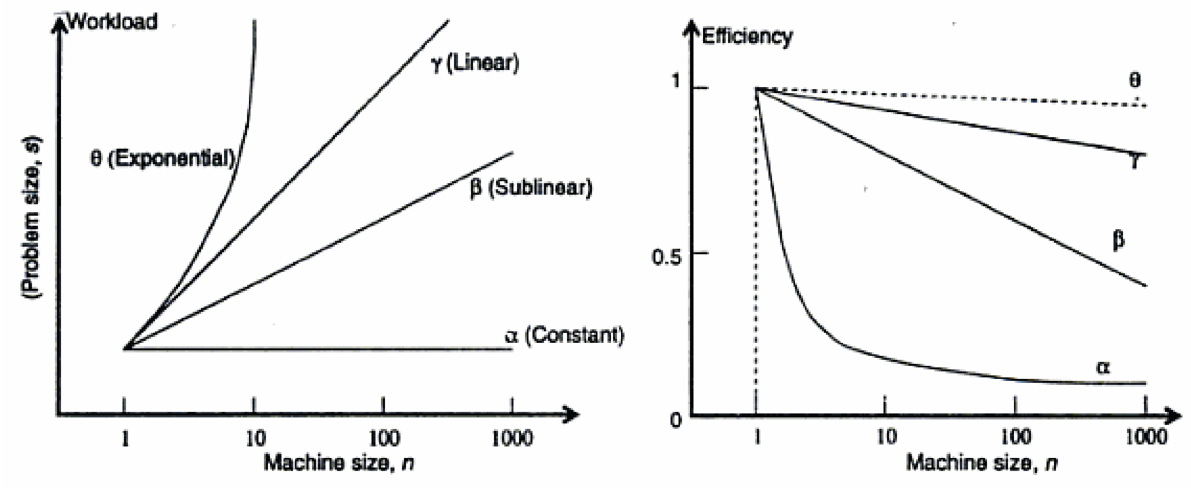


Figure 4.1 Machine Size versus Workload/Efficiency

The relation between the three factors provides the basis for improvements to be made in the present design. A single scalability metric cannot cover all aspects and thus different measures are to be taken depending upon the problem for which system is designed.

Computer cost and programming overhead are also important parts of the scalability analysis. It may be possible that some problems are poorly scalable while others have good scalable characteristics. Exploiting parallelism for higher performance requires both scalable architecture and algorithms. Trade-offs that exist in scalable architecture are communication latency, limited memory capacity and limited CPU speed.

4.5 Characteristics of Parallel Algorithms

An algorithm is efficient only when it can be implemented on a machine with cost-effectiveness and overcome the communication overheads and architectural limitations. A parallel algorithm is specially designed for execution on parallel computers simultaneously.

The following are the important characteristics of a parallel algorithm:

1 Deterministic: Only deterministic algorithms with polynomial time complexity are implementable on machines.
2 Granularity: The size of data and program modules determine whether the program is fine, medium or coarse grained.
3 Communication and Synchronization: Communication includes both memory access and inter-processor communication. It follows a pattern that may be static or dynamic. While static pattern is for SIMD, dynamic pattern is for MIMD machines. Synchronization affects the efficiency of the algorithm.
4 Extent of Parallelism: The degree of parallelism of the algorithm determines how effective will the parallel execution be.
5 Uniformity across the algorithm: If the operations performed in the program are uniform across the data set then SIMD structure is preferable otherwise MIMD.
6 Resource Requirements: Memory space and the data structure determine the efficiency to a vast extent as it determines the time and space complexity during data movement.

4.6 Speedup Performance Laws

4.6.1 Amdahl’s Law

Amdahl’s Law is applicable when problem size is fixed. It is used to find the maximum expected improvement to an overall system when only part of the system is improved. It is used to calculate maximum speed up for a multi-processor system. Amdahl’s Law defines a relationship between the expected speedup of parallelized implementation to that of sequential implementation. The law takes into consideration the fraction of sequential bottleneck i.e. the portion that cannot be parallelized, for calculating the speedup.

The model of evaluation is the one used by us in this project because we are also researching the speedup with fixed load.

Proof:

For a fixed workload, as the number of processors increase the load is distributed for parallel execution.

Let,

Maximum number of parallelism profile is m.

R_i be the execution rates and $w=f_i$ be the weighted distribution of programs $i=\{1,2,\ldots,m\}$.

Number of homogeneous processors available is n.

Then,

Mean execution time per instruction, $T_i = 1/ R_i$

Arithmetic mean execution time per instruction, $T_a = (1/m) * (\sum T_i)$

Harmonic mean execution rate, $R_h = 1/T_a = m/(\sum (1/R_i))$

Weighted Harmonic mean execution rate, $R_h^* = 1/ (\sum (f_i / R_i)) = 1/ T^*$

Thus,

Speedup, $S = T_1 / T^* = 1/ (\sum (f_i / R_i))$ [As $T_1 = 1$ for single processor]

Assume,

$R_i = i$ [$R_1 = 1$; i processors are used at any given interval]

w= (p,0,0,…,1-p) [This means that system works in either pure sequential or fully parallel state and p is the probability of sequential bottle-neck]

The speedup expression, $\mathbf{S_n= n / (1+(n-1)p)}$

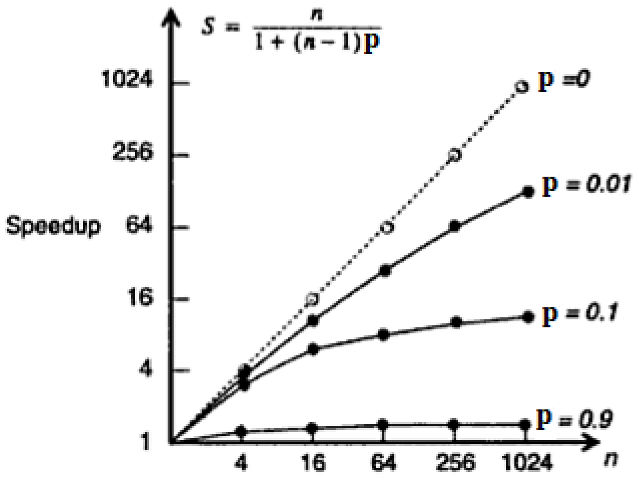


Figure 4.2 Comparison of Speedup and No. of Processors

This is known as Amdahl's Law. The maximum speedup that can be achieved is for infinite processors where S_n is upper-bounded by 1/p.

The speedup expression derived above is for ideal case where the communication latency is ignored. However, for real machines the inter-processor communication and memory access latency comes into play and reduces the speedup factor. Apart from this the delay caused by interrupts and operating system overheads are also included. All this can be summed up as Q(n).

Thus, Speedup factor $\mathbf{S_n = T(1) / (T(n) + Q(n))}$

The overhead delay is difficult to determine as it is application and machine dependent.

Amdahl's Law implies that the sequential portion of the program remains the same while the parallel portion is evenly executed by the n processors available. The speedup factor decreases rapidly as the sequential nature denoted by ‘p’ increases which means that even with a small portion of sequential code, the entire performance gets restricted till 1/p.

4.6.2 Gustafson’s Law

Gustafson’s Law overcomes the limitation of Amdahl’s Law of fixed workload. It is based on fixed-time speedup that implies solving large problems with large machines in the same time as smaller ones will be executed on small machines.

The assumption taken is that time of execution T(1) for a uniprocessor is equal to the time of execution T’(n) for a scaled problem.

Speedup factor according to Gustafson’s Law, $\mathbf{S'_n = n - p(n-1)}$

Gustafson’s Law does not support scalable performance as the machine size increases. Thus, problem size is increased such that all processors remain busy and sequential portion of the program is no longer the limiting factor.

4.7 Scalability Analysis

Scalable system follows the rule of linearly increasing the performance with the number of processors used for execution.

The following metrics determine the scalability of the system:

1. Machine Size (n): Number of processors and the resources determine the machine size.
2. Clock Rate (f): The clock rate applied to processor determines the cycle.
3. CPU Time (T): It is the time elapsed in executing a problem on parallel system with n processors.
4. Problem Size (s): The portion of workload that has to be executed by a single processor is the problem size.
5. Memory Capacity (m): The size of main memory available to the program also affects the frequency of secondary memory access and thus overhead.

6. Communication Overhead (h): The extra time taken in communicating between the processors and during synchronization.
7. Programming Overhead (p): The overhead involved in designing algorithms that can be executed on a parallel system and compatible with different architecture.

CHAPTER 5

MPI

Message Passing Interface (MPI) is a standardized and portable message-passing system designed by a group of researchers from academics and industry to function on a wide variety of parallel computers. MPI effort was started in summer of 1991 and a preliminary draft was proposed in November 1992 which is now known as MPI 1.

MPI is a specification for the developers and users of message passing libraries. By itself, it is NOT a library - but rather the specification of what such a library should be. The goal of the Message Passing Interface is to establish a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs. MPI is neither an IEEE standard nor ISO standard rather it is an industry level standard.

MPI provides parallel hardware vendors with a clearly defined base set of routines that can be efficiently implemented. As a result, hardware vendors can build upon this collection of standard low-level routines to create higher-level routines for the distributed-memory communication environment supplied with their parallel machines. MPI provides a simple-to-use portable interface for the basic user, yet powerful enough to allow programmers to use the high-performance message passing operations available on advance machines.

5.1 History of MPI

- MPI resulted from the efforts of numerous individuals and groups over the course of a 2 year period between 1992 and 1994.
- April, 1992: Workshop on Standards for Message Passing in a Distributed Memory Environment, sponsored by the Centre for Research on Parallel Computing, Williamsburg, Virginia. The basic features essential to a standard message passing interface were discussed, and a working group established to continue the standardization process. Preliminary draft proposal developed subsequently.

- November 1993: Supercomputing 93 conference - draft MPI standard presented.
- Final version of draft released in May, 1994.
- MPI-2 picked up where the first MPI specification left off, and addressed topics which go beyond the first MPI specification. The original MPI then became known as MPI-1. MPI-2 was finalized in 1996.
- The MPI Forum is now drafting the MPI-3 standard.

5.2 Need of MPI

- Standardization - MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.
- Portability - There is no need to modify the source code when you port your application to a different platform that supports the MPI standard.
- Performance Opportunities - Vendor implementations should be able to exploit native hardware features to optimize performance. For more information about MPI performance see the MPI Performance Topics tutorial.
- Functionality - Over 115 routines are defined in MPI-1 alone.
- Availability - A variety of implementations are available, both vendor and public domain.

5.3 Concepts of MPI 1

5.3.1 Communicator

Communicator objects connect groups of processes in the MPI session. Each communicator gives each contained process an independent identifier and arranges its contained processes in an ordered topology. MPI also has explicit groups, but these are mainly good for organizing and reorganizing groups of processes before another communicator is made. MPI understands single group intra communicator operations, and bilateral intercommunicator communication.

5.3.2 Point To Point Basics

A number of important MPI functions involve communication between two specific processes. A popular example is MPI_Send, which allows one specified process to send a message to a second specified process.
MPI supports mechanisms for both blocking and non-blocking point-to-point communication mechanisms.

5.3.3 Collective Basics

Collective functions involve communication among all processes in a process group. A typical function is the MPI_Bcast call. This function takes data from one node and sends it to all processes in the process group. A reverse operation is the MPI_Reduce call, which takes data from all processes in a group, performs an operation, and stores the results on one node. Reduce is often useful at the start or end of a large distributed calculation, where each processor operates on a part of the data and then combines it into a result.

5.3.4 Derived Data Types

Many MPI functions require that you specify the type of data which is sent between processors. This is because these functions pass variables, not defined types. If the data type is a standard one, such as int, char, double, etc., you can use corresponding predefined MPI data types such as MPI_INT, MPI_CHAR, MPI_DOUBLE respectively.
In case of user defined types we need to create our own MPI_NewType.
Example:

```
int MPI_Type_create_struct(int count, intblocklen[], MPI_Aintdisp[],  MPI_Datatype type[], MPI_Datatype *newtype);
```

It creates a user defined data type structure with respective parameters.

5.4 Concepts of MPI2

MPI 2 has 8 concepts with first four are common for both MPI 1 and MPI 2 and others are:

5.4.1 One-Sided Communication

MPI-2 defines three one-sided communications operations, Put, Get, and Accumulate, being a write to remote memory, a read from remote memory, and a reduction operation on the same memory across a number of tasks. Also defined are three different methods to synchronize this communication (global, pair wise, and remote locks) as the specification does not guarantee that these operations have taken place until a synchronization point.
These types of call can often be useful for algorithms in which synchronization would be inconvenient (e.g. distributed matrix multiplication), or where it is desirable for tasks to be able to balance their load while other processors are operating on data.

5.4.2 Collective Extensions

Collective Extensions allows the application of collective operations to inter communicative Collective communication must involve all processes in the scope of a communicator. All processes are by default, members in the communicator MPI_COMM_WORLD. It is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operations.

Types of Collective Operations:

1 Synchronization - processes wait until all members of the group have reached the synchronization point.
2 Data Movement - broadcast, scatter/gather, all to all.
3 Collective Computation (reductions) - one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.

5.4.3 Dynamic Process Management

The key aspect is "the ability of an MPI process to participate in the creation of new MPI processes or to establish communication with MPI processes that have been started separately." The MPI-2 specification describes three main interfaces by which MPI processes can dynamically establish communications, MPI_Comm_spawn, MPI_Comm_accept/MPI_Comm_connect and MPI_Comm_join. The MPI_Comm_spawn interface allows an MPI process to spawn a number of instances of the named MPI process. The newly spawned set of MPI processes form a new MPI_COMM_WORLD intra communicator but can communicate with the parent and the intercommunicator the function returns.

5.4.4 Parallel I/O

The parallel I/O feature is sometimes called MPI-IO, and refers to a set of functions designed to abstract I/O management on distributed systems to MPI, and allow files to be easily accessed in a patterned way using the existing derived data type functionality.

CHAPTER 6

MPICH2

MPICH is a freely available, portable implementation of MPI, a standard for message-passing for distributed-memory applications used in parallel computing. MPICH is Free Software and is available for most flavours of UNIX (including Linux), Mac OS X and Microsoft Windows.

The CH part of the name was derived from "Chameleon", which was a portable parallel programming library developed by William Gropp, one of the founders of MPICH. The original implementation of MPICH (MPICH1) implements the MPI-1.1 standard. The latest stable version of MPICH2 is 1.5. The latest implementation is the MPI-3.0rc1 standard.

MPICH2 is one of the most popular implementations of MPI. It is used as the foundation for the vast majority of MPI implementations including IBM MPI (for Blue Gene), Intel MPI, Cray MPI, Microsoft MPI, Myricom MPI, OSU MVAPICH/MVAPICH2, and many others.

The goals of MPICH are:

1 To provide an MPI implementation that efficiently supports different computation and communication platforms including commodity clusters, high-speed networks and proprietary high-end computing systems

2 To enable cutting-edge research in MPI through an easy-to-extend modular framework for other derived implementations.

6.1 MPICH2 Built-in Scripts

Language	**Script Name**	**Underlying Compiler**
C	Mpicc	Gcc
	Mpigcc	Gcc
	Mpiicc	Icc
	Mpipgcc	Pgcc
C++	mpiCC	g++
	mpig++	g++
	Mpiicpc	Icpc
	mpipgCC	pgCC
Fortran	mpif77	g77
	Mpigfortran	Gfortran
	Mpiifort	Ifort
	mpipgf77	pgf77
	mpipgf90	pgf90

Table 6.1 MPICH2 Build in scripts

6.2 Header File

For C version: #include ”mpi.h”

For Fortran version: include ‘mpif.h’

6.3 General MPI Program Structure

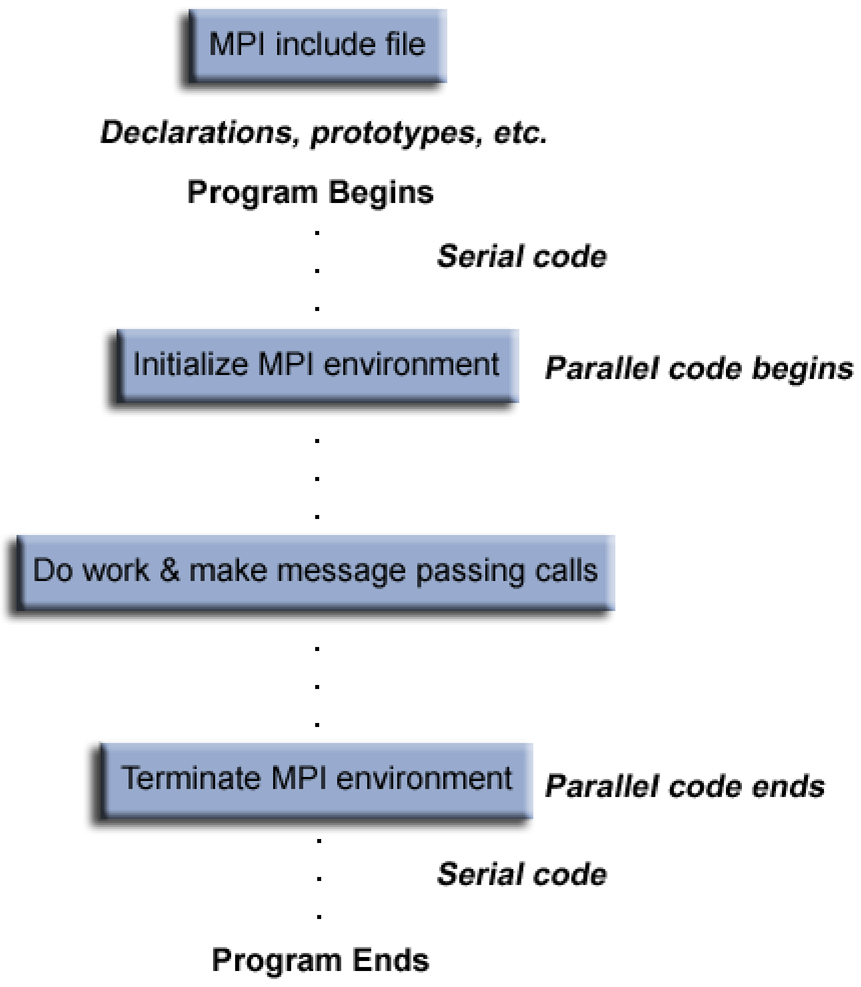

MPI include file
Declarations, prototypes, etc.
Program Begins
Serial code
Initialize MPI environment
Parallel code begins
Do work & make message passing calls
Terminate MPI environment
Parallel code ends
Serial code
Program Ends

Figure 6.1 MPICH2 Program Structure

6.4 Data Types

MPI C version Data Types	Equivalent C Data Type/Meaning
MPI_CHAR	signed char
MPI_WCHAR	wchar_t - wide character
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT MPI_LONG_LONG	signed long long int
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	Float
MPI_DOUBLE	Double
MPI_LONG_DOUBLE	long double
MPI_BYTE	8 binary digits

Table 6.2 MPI Data Types

6.5 Environment Management Routines

MPI environment management routines are used for an assortment of purposes, such as initializing and terminating the MPI environment, querying the environment and identity, etc. Most of the commonly used ones are described below.

- MPI_Init(&argc,&argv)

Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program.

- MPI_Comm_size(comm,&size)

Returns the total number of MPI processes in the specified communicator, such as MPI_COMM_WORLD. If the communicator is MPI_COMM_WORLD, then it represents the number of MPI tasks available to your application.

- MPI_Comm_rank(comm,&rank)

Returns the rank of the calling MPI process within the specified communicator. Initially, each process will be assigned a unique integer rank between 0 and number of tasks - 1 within the communicator MPI_COMM_WORLD. This rank is often referred to as a task ID. If a process becomes associated with other communicators, it will have a unique rank within each of these as well.

- MPI_Abort(comm,errorcode)

Terminates all MPI processes associated with the communicator. In most MPI implementations it terminates ALL processes regardless of the communicator specified.

- MPI_Get_processor_name(&name,&resultlength)

Returns the processor name. Also returns the length of the name.

- MPI_Get_version(&version,&subversion)

Returns the version (either 1 or 2) and subversion of MPI.

- MPI_Initialized(&flag)

Indicates whether MPI_Init has been called - returns flag as either logical true (1) or false(0). MPI requires that MPI_Init be called once and only once by each process.

- MPI_Wtime()

Returns an elapsed wall clock time in seconds (double precision) on the calling processor.

- MPI_Finalize()

Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program - no other MPI routines may be called after it.

6.6 Point to Point Communication Routines

6.6.1 Blocking Message Passing Routines

The more commonly used MPI blocking message passing routines are described below.

- MPI_Send(&buf,count,datatype,dest,tag,comm)

Basic blocking send operation. Routine returns only after the application buffer in the sending task is free for reuse. Note that this routine may be implemented differently on different systems.

- MPI_Recv(&buf,count,datatype,source,tag,comm,&status)

Receive a message and block until the requested data is available in the application buffer in the receiving task.

- MPI_Ssend(&buf,count,datatype,dest,tag,comm)

Synchronous blocking send: Send a message and block until the application buffer in the sending task is free for reuse and the destination process has started to receive the message.

- MPI_Wait(&request,&status)

MPI_Wait blocks until a specified non-blocking send or receive operation has completed. For multiple non-blocking operations, the programmer can specify any, all or some completions.

- MPI_Bsend(&buf,count,datatype,dest,tag,comm)

Buffered blocking send: permits the programmer to allocate the required amount of buffer space into which data can be copied until it is delivered. Insulate against the problems associated with insufficient system buffer space.

- MPI_Buffer_attach(&buffer,size) and MPI_Buffer_detach(&buffer,size)

Used to allocate/deallocate message buffer space to be used by the MPI_Bsend routine. The size argument is specified in actual data bytes - not a count of data elements. Only one buffer can be attached to a process at a time.

- MPI_Probe(source,tag,comm,&status)

Performs a blocking test for a message. The "wildcards" MPI_ANY_SOURCE and MPI_ANY_TAG may be used to test for a message from any source or with any tag.

6.6.2 Non-Blocking Message Passing Routines

The more commonly used MPI non-blocking message passing routines are described below.

- MPI_Isend(&buf,count,datatype,dest,tag,comm,&request)

Identifies an area in memory to serve as a send buffer. Processing continues immediately without waiting for the message to be copied out from the application buffer. A communication request handle is returned for handling the pending message status.

- MPI_Irecv(&buf,count,datatype,source,tag,comm,&request)

Identifies an area in memory to serve as a receive buffer. Processing continues immediately without actually waiting for the message to be received and copied into the the application buffer. A communication request handle is returned for handling the pending message status.

- MPI_Issend(&buf,count,datatype,dest,tag,comm,&request)

Non-blocking synchronous send. Similar to MPI_Isend(), except MPI_Wait() or MPI_Test() indicates when the destination process has received the message.

- MPI_Ibsend(&buf,count,datatype,dest,tag,comm,&request)

Non-blocking buffered send. Similar to MPI_Bsend() except MPI_Wait() or MPI_Test() indicates when the destination process has received the message. Must be used with the MPI_Buffer_attach routine.

- MPI_Irsend(&buf,count,datatype,dest,tag,comm,&request)

Non-blocking ready send. Similar to MPI_Rsend() except MPI_Wait() or MPI_Test() indicates when the destination process has received the message.

- MPI_Test(&request,&flag,&status)

MPI_Test checks the status of a specified non-blocking send or receive operation. The "flag" parameter is returned logical true (1) if the operation has completed, and logical false (0) if not. For multiple non-blocking operations, the programmer can specify any, all or some completions.

6.7 Collective Communication Routines

- MPI_Barrier(comm)

Creates barrier synchronization in a group. Each task, when reaching the MPI_Barrier call, blocks until all tasks in the group reach the same MPI_Barrier call.

- MPI_Bcast(&buffer,count,datatype,root,comm)

Broadcasts (sends) a message from the process with rank "root" to all other processes in the group.

- MPI_Scatter(&sendbuf,sendcnt,sendtype,&recvbuf,recvcnt,recvtype,root,comm)

Distributes distinct messages from a single source task to each task in the group.

- MPI_Gather(&sendbuf,sendcnt,sendtype,&recvbuf, recvcount,recvtype,root,comm)

Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of MPI_Scatter.

- MPI_Allgather(&sendbuf,sendcount,sendtype,&recvbuf,recvcount,recvtype,comm)

Concatenation of data to all tasks in a group. Each task in the group, in effect, performs a one-to-all broadcasting operation within the group.

- MPI_Reduce(&sendbuf,&recvbuf,count,datatype,op,root,comm)

Applies a reduction operation on all tasks in the group and places the result in one task.

CHAPTER 7

IMPLEMENTATION OF THE CLUSTER: NETWORK FILE SYSTEMAND MORE COMMAND LINEACTION

Implementation of the cluster was a necessarily two-step process:

* *Creating a connection between the systems: Making systems talk*

**Setting up the MPI environment: Making the systems talk about work*

These 2 steps are covered in essential details.

7.1 Creating The LAN Connection

For creating a LAN, unarguably, the best device is the switch. Switch is a clear choice over the other options like Bridges and Hubs for the reasons Switch Reduces the collision domain. Each port of the switch is one collision domain. Thus the number of collisions is greatly reduced.

Now, switches come in variety. We have fast Ethernet switches and Gigabit switches as the 2 most popular flavours available. We decided to go with fast Ethernet switches because most organizations have dumped them!

This is justified because one of the major objectives of the project is determination of the computing power of a cluster made of dumped machines. (Debatable line: So for this one case, we can actually draw a parallel between computer hardware and wine)

The switch we used was a simple 5 port unmanaged switch from D link.

Figure 7.1 Switch

Other specifications of the switch are as follows:

Protocols Supported	IEEE 802.3, IEEE 802.3u, ANSI/IEEE 802.3, IEEE 802.3x
Forward rate	10 Mbps port: 14,880 packets/sec, 100 Mbps port: 148,800 packets/sec
MAC Address database	2000
Management Protocol	CSMA/CD
Model	5-Port 10/100BASE-T Unmanaged Switch

Table 7.1   Switch Specification

So the said switch and some straight coax wires were enough to get the machines to ping each other. The next step, setting up of the MPI environment, is discussed next.

7.2 Setting Up The MPI Environment

The steps to set up the environment are very nicely covered in the official Ubuntu tutorial and we had little to do but to simply follow the steps listed. However, there were several local issues that had to be sorted in order to set up the cluster.

The steps that were followed are listed in order as follows:

1) Installing NFS

NFS or network file system is a tool that allows users across the LAN to share their files. A common file system, hosted by the master (NFS server), is mirrored to all the nodes (NFS clients).

2) Installing GCC, MPICH2 and other development tools

All the required tools were available pre-packaged and were simply fetched using the

$get-apt install command

3) Installing SSH Server and setting up password less SSH for communication between nodes

The mpiexec command executes the same program at all the nodes and thus it needs password less ssh.

So with these 2 steps completed, we had the working MPI environment. Simple!
Note: The system is highly scalable. We can simply install NFS Client in any computer, plug it up in the LAN and it will be ready to contribute!

Hitherto we have discussed the background required before one starts with parallel programming. In next chapter, we discuss the results obtained after running different types of programs. We discuss the speed up observed for massively parallel problems and the hit we see for problems involving huge interchange of data.

CHAPTER 8

PROGRAMMING USING MPI

In this chapter, we will see how MPI works by studying structure of the simplest possible MPI program. The simplest possible program looks like this:

```
#include <stdio.h>
#include <mpi.h>
#define ROOT 0
intmain(intargc, char** argv) {
intmyrank, nprocs;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if(myrank==ROOT)
{
printf("Special Greetings From The ROOT!\n");
}
printf("Hello from processor %d of %d\n", myrank, nprocs);

MPI_Finalize();
   return 0;
}
```

MPI follows the SPMD (Single Program, Multiple Data) type model of parallelism (Chapter 2).

The same code is executed on each of the nodes. The individual nodes ascertain what they have to do by figuring out their rank and then executing the code wrapped in the correct if() { … } block .

Initialize The MPI Environment:

```
MPI_Init(&argc, &argv);
```

Determine The Number of Processes:

```
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
```

Find Out Your Rank:

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

CHAPTER 9

IMPLEMENTATION

9.1 Introduction

Various programs were made to analyse the cluster and the performance gain using different configurations requiring single node or multiple nodes, with or without the Fox node, which is the Intel's i3 processor. The programs were made using iterative algorithms. Number of iterations was taken from the user and the graphs were plotted against the execution time using Octave software. The various programing problems analysed were:

1. Calculating the value of pi.
2. Calculating the summation of N numbers.
3. Calculating the global maximum element in an array: the curious case of distributed memories.

9.2 Calculating The Value of PI

A program was made to calculate the value of an irrational number pi using iterative algorithm. The value can be calculated more accurately with increasing the number of iterations. The iterations are divided among the nodes thus providing the speedup.

The results obtained are as follows:
The program distributed the processing equally among all the participating nodes. The outputs for various values of iterations are shown in figure 9.1 and figure 9.2.

Figure 9.1 shows the expected gain in performance with increasing number of nodes working together. The time taken with all nodes working together is least, as expected. The percentage gain is almost equal with every node added to the cluster. As the number of iterations is increased, the communication delay, a fixed constant, subdues and we see very sharp gains.

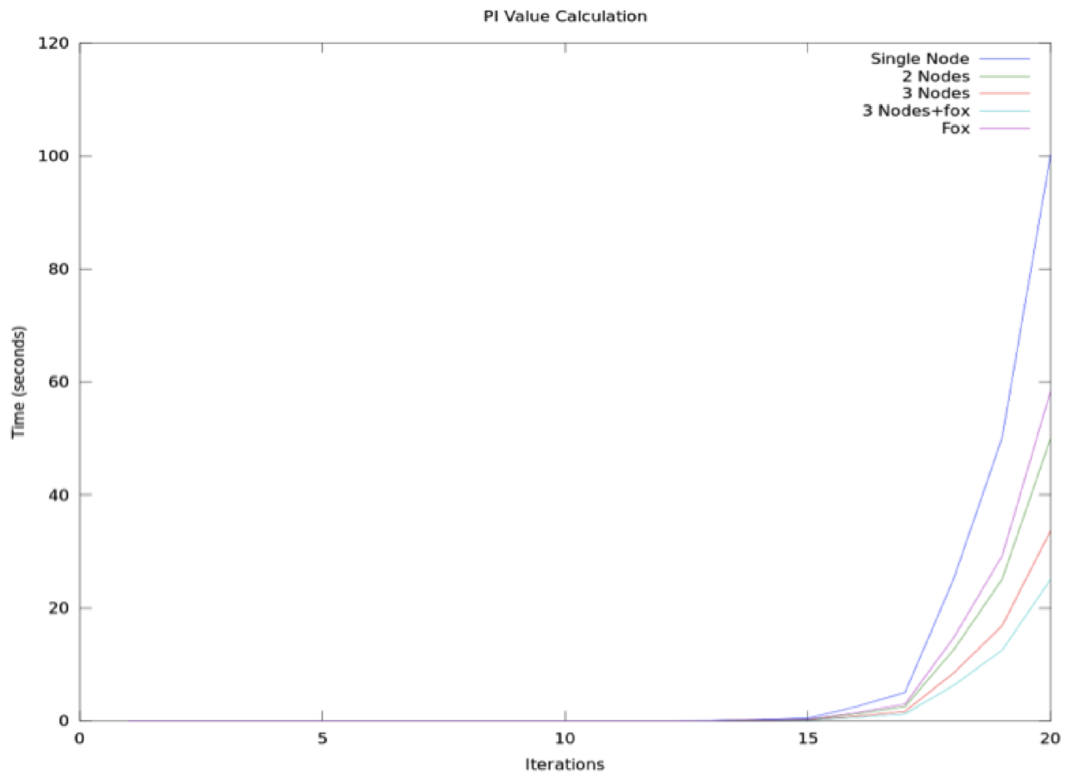


Figure 9.1 Time against number of iterations for different set of nodes combinations for calculating the PI

The following table shows the factor by which there is an increase in the performance with respect to the performance with the single node.

Configuration	**Average Speedup Relative to Single Node**
Single Node	1
Two Nodes	2.0011
Three Nodes	2.9749
Fox	1.73143
Three Nodes + Fox	4.0017

Table 9.1 Speedup Factor for Figure 9.1

Relative Speedup For pi Value Calculation

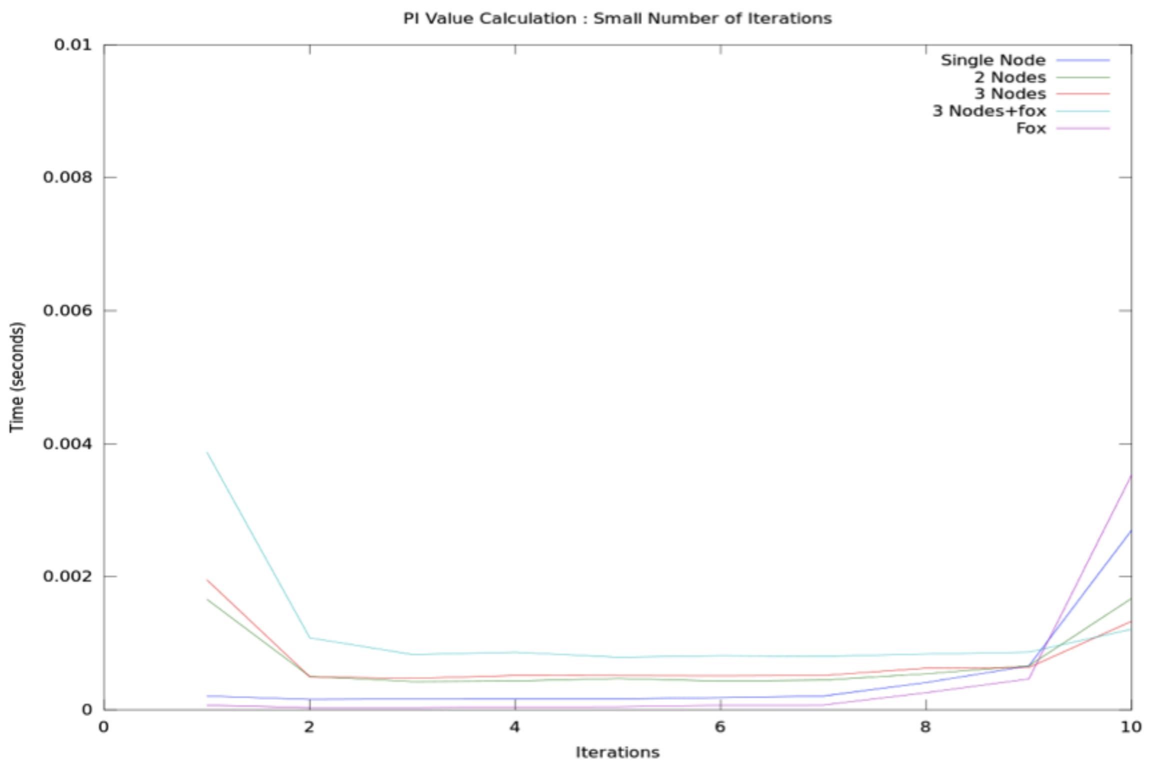


Figure 9.2

Time against comparatively less number of iterations for different set of nodes combinations for calculating the PI

Figure 9.2 shows that there is *negative gain* when the numbers of iterations are small. The plausible reason for this is the communication overhead, a constant, which comes into play for dividing the problem. The single node gives the best result for solving small problems. Even if the other node (Fox) has more processing speed than the node working alone, there is no performance gain. Hence for small number of iterations the single Pentium 4 processor gives best results working alone.

The communication delay is constant for this problem due to the fact that the only the number of iterations is broadcasted to the cluster and partial results from all the nodes are then collected. Thus, for a given number of nodes, the communication cost is independent of the number of iterations.

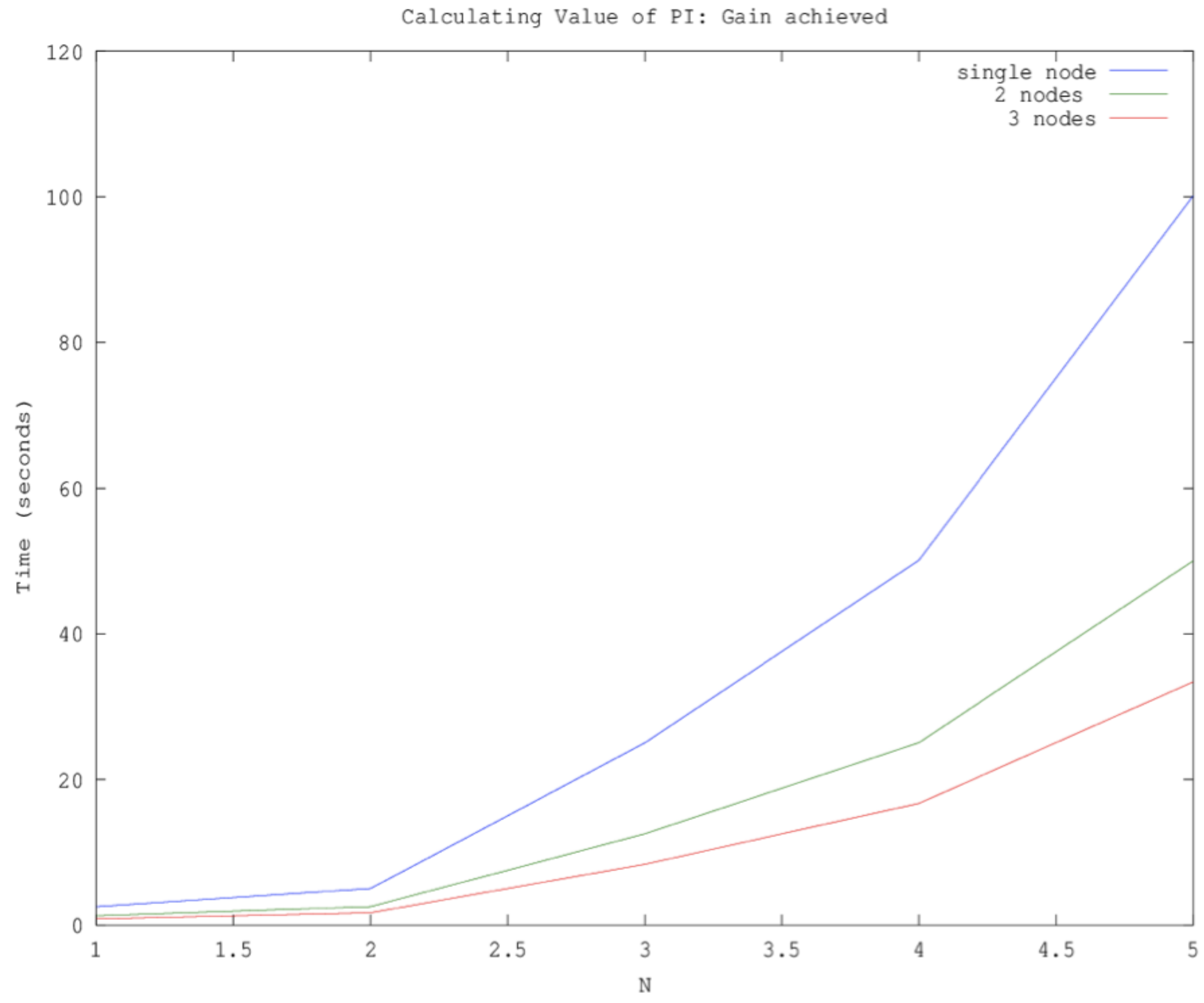


Figure 9.3

Time against number of iterations for different set of node combinations for calculating the PI

Figure 9.3 shows the expected result for very large number of iterations with no conflict in the expected rise in the performance with each node added. This is because the time of execution has surpassed the communication delay time enough to show the expected rise in performance and decrease in the execution time.

9.3 Calculating Value of ΣN

Value calculation of sigma N is another problem that showed remarkable gains, the reason being minimal communication overhead and a very remarkable parallelism in the problem. N nodes are given a task of summing their share. Each node is assigned the task of summing an equivalence class of N% k, where k is the number of nodes. Eg, for k =3, we have 3 equivalence classes, [0]={0,3,6,...} , [1]={1,4,7,...} and [2]={2,5,8,...}

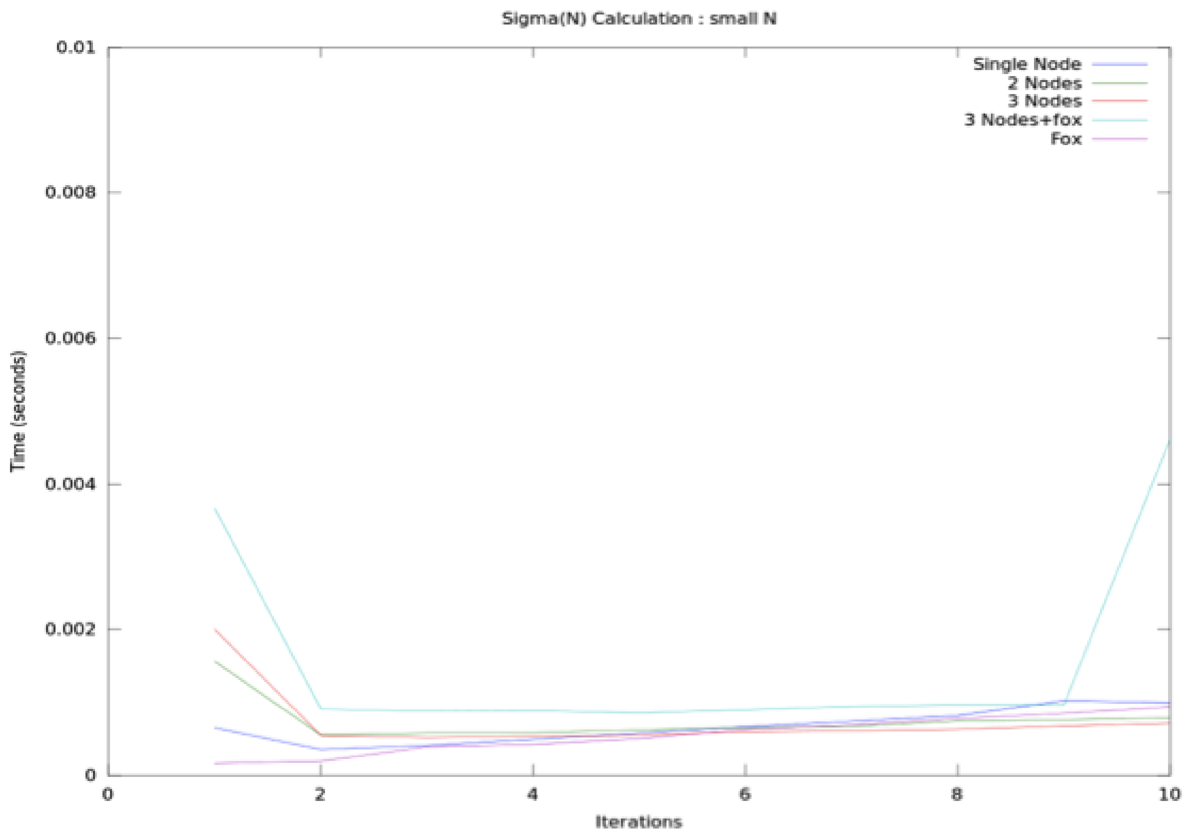


Figure 9.4

Time against number of iterations for different set of nodes combinations for calculating the

∑N

The time taken for small number (e.g. 3 to 4) of iterations to be solved for single Pentium 4 processor is less than the time with single i3 processor. It can be inferred from this search result that there is a possibility for a problem to take lesser amount of time to be solved on older generation processor (Pentium 4) than the younger generation (i3). This is thus evident that the addition of new technologies in the i3 processor adds up the computation delay for certain size of problems.

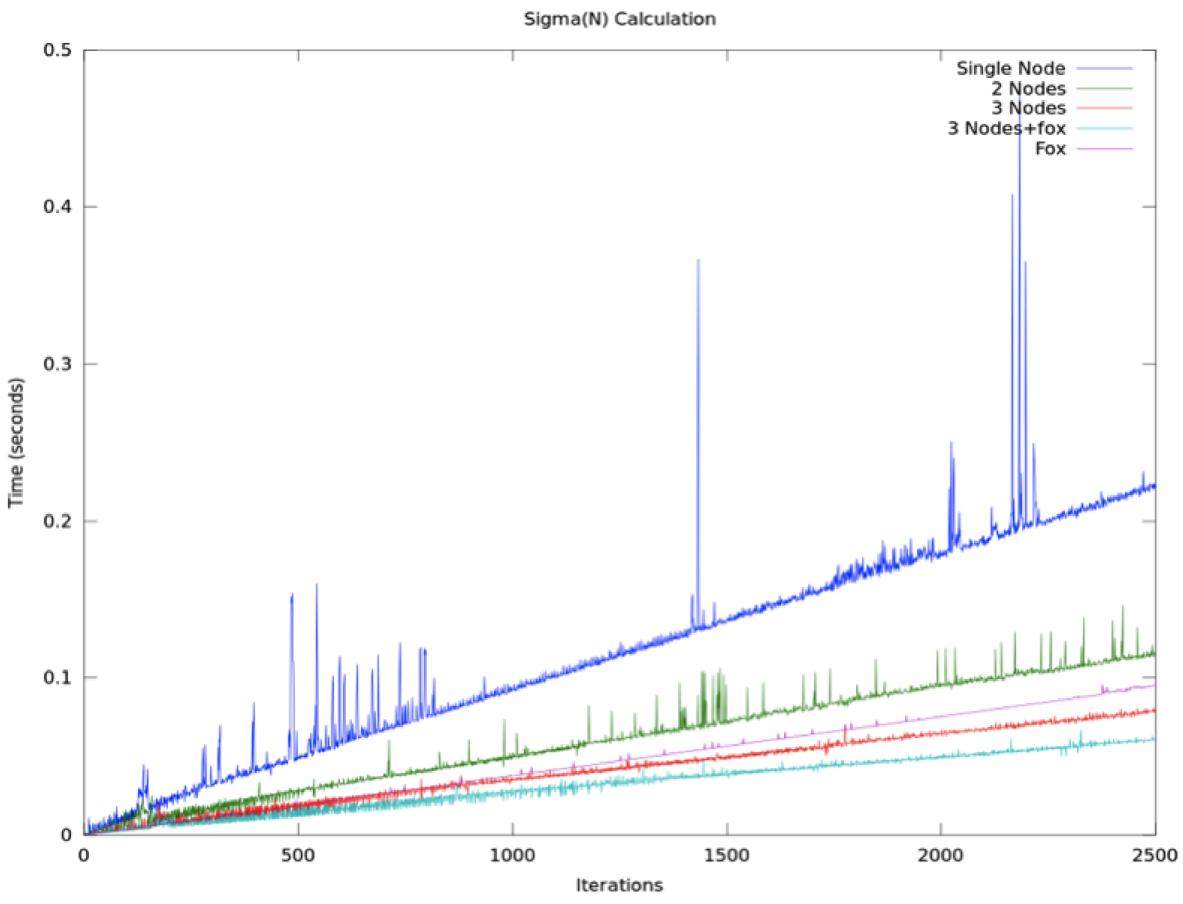


Figure 9.5

Time against moderate number of iterations for different set of nodes combinations for calculating the ∑N

It can be inferred from the graph that for few hundred iterations to few thousands number of iterations, the graph shows the expected result. The execution time is decreasing as more nodes are added. The execution time decreases to almost its half with every node added. A closer look tells that we don't gain much by adding a third, slower node.

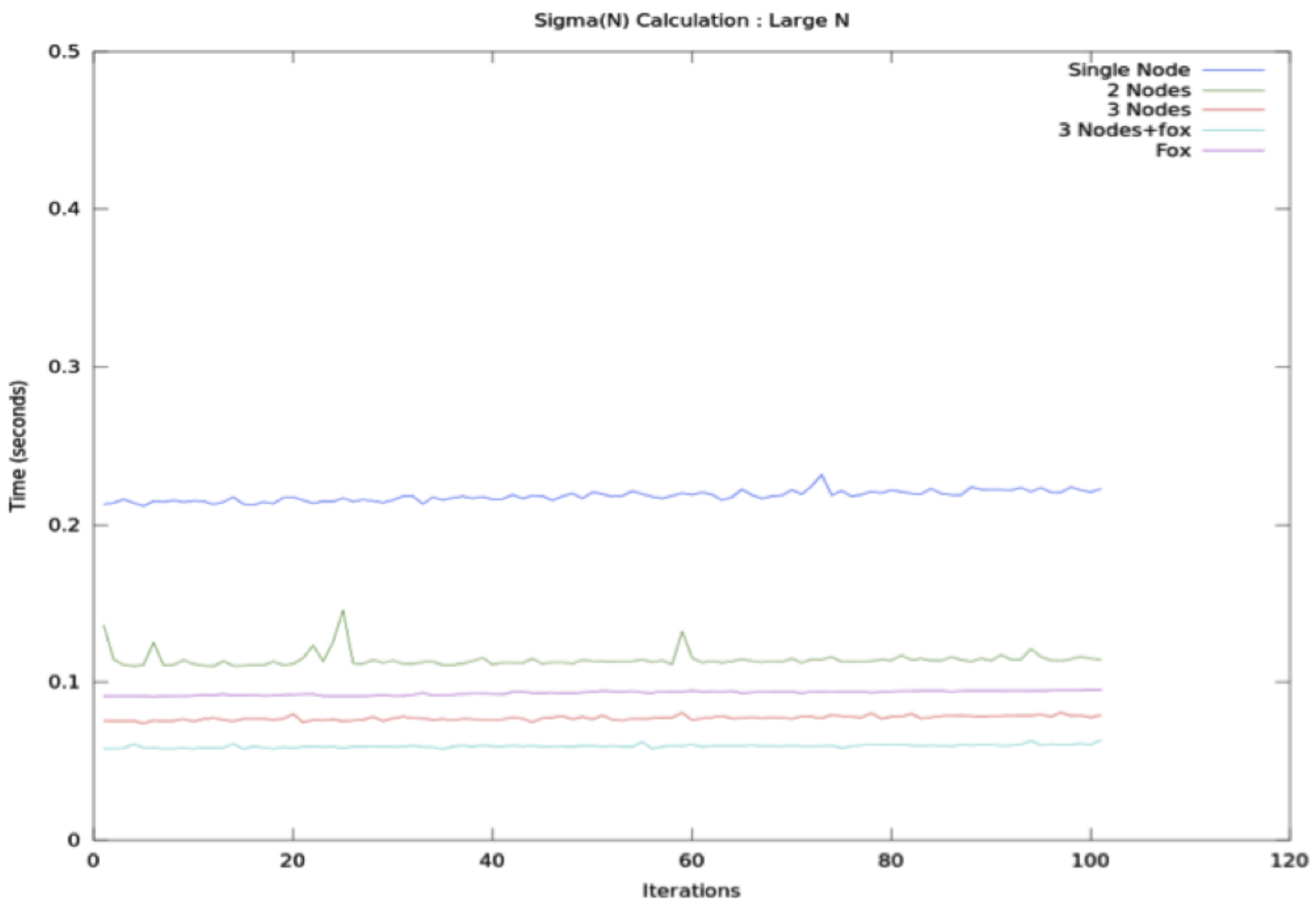


Figure 9.6

Time against large number of iterations for different set of nodes combinations for calculating the ∑ N

The figure 9.6 shows the graph results for very large number of iterations where the lines seem to get parallel. The results at time number of iterations give more idea about the effect of increasing the number of nodes. It can be easily noticed that the execution time decreases to half its value with every added node. It can also be inferred from the graph that three Pentium 4 processors working in a cluster together can beat the performance of one FOX node for this problem, as the time of execution for same number of iterations is less for three Pentium 4 processors than the time taken by the FOX node (Intel i3 processor) to solve the same problem.

Configuration	**Average Speedup Relative to Single Node**
Single Node	1
Two Nodes	1.9028
Three nodes	2.7979
Fox	2.4731
Three Nodes + Fox	3.6212

Table 9.2 Speedup Factor for Figure 9.6

9.4 CALCULATING THE GLOBAL MAXIMUM ELEMENT IN AN ARRAY: THE CURIOUS CASE OF DISTRIBUTED MEMORIES

A program was made to calculate the maximum element in an array. This was done by dividing the array into several nodes and calculating the local maximum. The results from each individual node were calculated and then the global maximum was calculated. Each node used its local memory to calculate the local maximum.

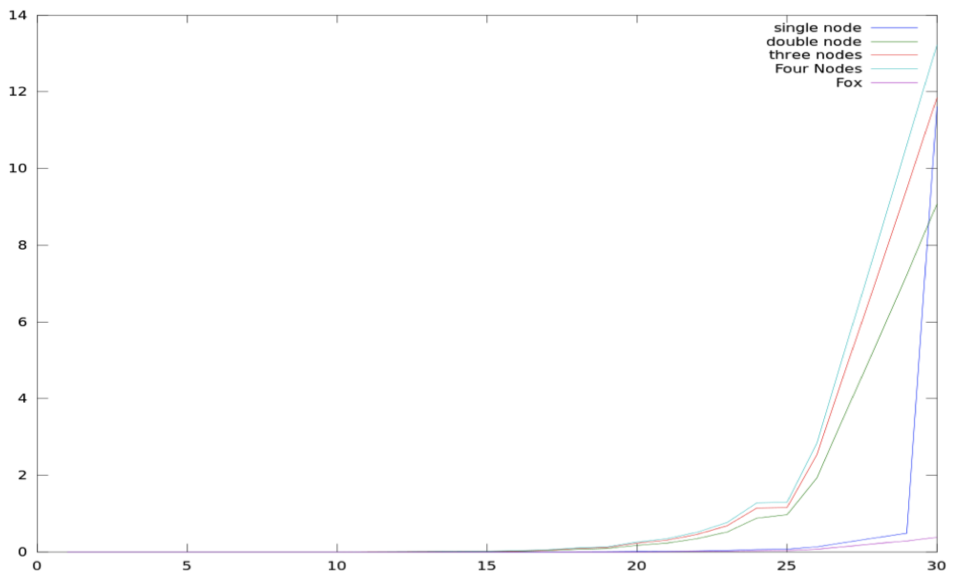


Figure 9.7

Graph between Number of Iterations against Execution Time

Distributed memories are usually very slow, with the order of performance hit coming out to be as much as 20 times. A similar kind of performance hit was observed with this program that involves a large amount of data interchange.

Configuration	**Relative Speedup**
Single Node	1
Two Nodes	0.069 (1/16.66)
Three nodes	0.053 (1/ 18.868)
Fox	1.75
Three Nodes + Fox	.047 (1/21.2)

Table 9.3 Speedup Factor for Figure 9.7

As can be seen from the table, with distributed memories, a hit of up to 21 times was experienced. The i3 processor again succeeded in reducing the running time to about a half.

CHAPTER 10

SUMMARY AND CONCLUSIONS

Design and implementation of a parallel system was a unique experience. It helped us in improving our experience with the Linux command line, and helped us appreciate the beauty of parallel programming. Apart from numerous facts that we have learned about parallel programs, there are three major conclusions that may be categorically drawn from our project:

1. For problems that are embarrassingly parallel, the speed up for k nodes is about k. In a nutshell, the speed up achieved is near perfect for such problems.

2. Distributed memories are slow. Shared memories are relatively very fast. The fact that we have used a relatively slower variant of the network switch actually makes distributed memories all the more slower.

4. A cluster of "dumped" systems yields computing power that can outperform modern systems if handled properly.

5. For relatively simpler problems, using modern hardware is actually anoverkill. Pentiums are faster than i3 for small number of iterations.

REFERENCES AND BIBLIOGRAPHY

[1] Glenn R. Luecke, Marina Kraeva, Jing Yuan and Silvia Spanoyannis " Performance and scalability of MPI on PC clusters" 291 Durham Center, Iowa State University, Ames, IA 50011, U.S.A.

[2] Graham, R.L. Adv. Comput. Lab., Los Alamos Nat. Lab., NM Shipman, G.M. ; Barrett, B.W. ; Castain, R.H. ; Bosilca, G. ; Lumsdaine, A. "Open MPI: A High-Performance, Heterogeneous MPI", 25-28 Sept. 2006.

[3] http://www.mpi-forum.org/

[4]. Discussion on Parallel Computing By Ms Preeti At Indian Institute of Science, Bangalore on 12th July 2012.

[5]. The MPICH2 User's guide http://www.mcs.anl.gov/research/projects/mpich2/documentation/files/mpich2-1.4.1-userguide.pdf

[6]. The Wiki page on parallel computing http://en.wikipedia.org/wiki/Parallel_computing

[7]. KrsteAsanovic, RasBodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, Katherine A. Yelick, titled "The Landscape of Parallel Computing Research: A View From Berkeley", December 18, 2006.

[8]Keqin Li, Yi Pan, Si-Qing Zheng, titled "Parallel Computing Using Optical Interconnections".

[9]David Patterson and a cast of thousands Pardee Professor of Computer Science, U.C. Berkeley Director, RAD Lab, U.C. Berkeley Past President, Association for Computing Machinery, titled "The Berkeley View: A New Framework & a New Platform for Parallel Research", November, 2006

APPENDIX-I

MPI PROGRAMS

1.1 DISTRIBUTED MERGE SORT: PARTIAL

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mergesort.c"
/*Reads a file into the array and
returns the number of integers*/
int readFile(char * fn,int * arr)
{
	int i=0;
	FILE * fp=fopen(fn,"r");
	if(!fp)
	{
		return -1;
	}
	while(fscanf(fp,"%d",&arr[i]))
	{
		if(arr[i]==-1)
		{
			break;
		}
```

```
                    i++;
          }
          return i;
}
int main(int argc,char *argv[])
{
          /* argv[1] : file name
           * argv[2] : numbers
           */

int myid,           /*id of the process*/
                    numprocs, /* total number of processes*/
                    i                   /* iterator*/;
double startwtime = 0.0, endwtime; //time trackers
int share; //each processes share
int * arr; //the receive buffer
int N; //total number of processes
int * initbuff; //send buffer
int namelen;//for processor name length
char processor_name[MPI_MAX_PROCESSOR_NAME];
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);

MPI_Get_processor_name(processor_name,&namelen);

if(myid==0)
   {
                    //first read all the values in an initial array
```

```
                initbuff=(int *)malloc(sizeof(int)*atoi(argv[2]));
                int total=readFile(argv[1],initbuff);
                if(total==-1)
                {
                        printf("Error!");
                        return 0;
                }
                //calculate the share of each node
                share=total/numprocs;
                //now scatter the array to all the processes
                startwtime = MPI_Wtime();
        }
                MPI_Bcast(&share, 1, MPI_INT, 0, MPI_COMM_WORLD);
                arr=(int*)malloc(sizeof(int)*share);
MPI_Scatter(initbuff,share, MPI_INT,arr,share, MPI_INT,0,MPI_COMM_WORLD);
        printf("id=%d share=%d\n",myid,share);
mergeSort(arr,0,share-1);
     //dump(arr,(myid)*(share),(myid+1)*(share)-1,myid);
                dump(arr,0,share-1,myid);
                if(myid==0)
                {
endwtime = MPI_Wtime();    //now everyone sorts their share
                printf("wall clock time = %f\n", endwtime-startwtime);
                }
                fflush(stdout );
MPI_Finalize();
return 0;
}
```

1.2 PI VALUE CALCULATION

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>

double f(double a)
{
return (4.0 / (1.0 + a*a));
}

int main(int argc,char *argv[])
{
int done = 0, n, myid, numprocs, i;
double PI25DT = 3.141592653589793238462643;
double mypi, pi, h, sum, x;
double startwtime = 0.0, endwtime;
int namelen;
char processor_name[MPI_MAX_PROCESSOR_NAME];

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
MPI_Get_processor_name(processor_name,&namelen);

   /*
fprintf(stdout,"Process %d of %d is on %s\n",
            myid, numprocs, processor_name);
```

```
fflush(stdout);
    */


while (!done) {
if (myid == 0) {
fprintf(stdout, "Enter the number of intervals: (0 quits) ");
            fflush(stdout);
if (scanf("%d",&n) != 1) {
                        fprintf(stdout, "No number entered; quitting\n" );
                        n = 0;
                }
            startwtime = MPI_Wtime();
        }
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (n == 0)
done = 1;
else {
            h   = 1.0 / (double) n;
sum = 0.0;
for (i = myid + 1; i<= n; i += numprocs) {
                x = h * ((double)i - 0.5);
                sum += f(x);
            }
mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);


if (myid == 0) {
printf("pi is approximately %.16f, Error is %.16f\n",
pi, fabs(pi - PI25DT));
```

```
                endwtime = MPI_Wtime();
                printf("wall clock time = %f\n", endwtime-startwtime);
                fflush(stdout );
            }
        }
    }
MPI_Finalize();
return 0;
}
```

1.3 CALCULATION OF ∑N

```
#include "mpi.h"
#include <stdio.h>
#include<stdlib.h>
#include <math.h>
int main(intargc,char *argv[])
{
int done = 0, n=0, myid, numprocs, i;
double share,total;
double startwtime = 0.0, endwtime;
int namelen;
int N;
n=atoi(argv[1]);
char processor_name[MPI_MAX_PROCESSOR_NAME];
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
MPI_Get_processor_name(processor_name,&namelen);
startwtime = MPI_Wtime();
share=0;
MPI_Bcast(arr, 1, MPI_INT, 0, MPI_COMM_WORLD);
for (i = myid ; i<= n; i += numprocs)
        {
            share+=i; //everyone calculates their bit, then they sum it up
      }
MPI_Reduce(&share, &total, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
if (myid == 0) {
printf("sum till %d is approximately %.16f",n,total);
            endwtime = MPI_Wtime();
            printf("wall clock time = %f\n", endwtime-startwtime);
            fflush(stdout );
        }
MPI_Finalize();
return 0;
}
```

1.4 PROGRAM TO CALCULATE LARGEST IN AN ARRAY

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define ROOT 0
/*Reads a file into the array and
returns the number of integers*/
int maxN(int * arr,int n)
{
	int i=0,max;
	for(i=0;i<n;i++)
	{
		if(arr[i]>max)
		{
			max=arr[i];
		}
	}
	return max;
}
void dump(int * arr,intN,int id)
{
	int i=0;
	printf("%d's Share=",id,N);
	while(i<N)
	{
		printf("\n%d",arr[i]);
		i++;
	}
}
int readFile(char * fn,int * arr)
{
	int i=0;
	FILE * fp=fopen(fn,"r");
	if(!fp)
	{
		return -1;
	}
	while(fscanf(fp,"%d",&arr[i]))
```

```
          {
                    if(arr[i]==-1)
                    {
                              break;
                    }
                    i++;
          }
          return i;
}
int main(int argc,char *argv[])
{
int myid, /*id of the process*/
                    numprocs, /* total number of processes*/
                    i /* iterator*/;
doublestartwtime = 0.0, endwtime; //time trackers
int share; //each processes share
int * arr ;//the receive buffer
int N; //total number of processes
int * initbuff; //send buffer
int namelen;//for processor name length
int max,maxlocal;
char processor_name[MPI_MAX_PROCESSOR_NAME];
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
MPI_Get_processor_name(processor_name,&namelen);
if(myid==ROOT)
   {
                    //first read all the values in an initial array
```

```
                initbuff=(int *)malloc(sizeof(int)*atoi(argv[2]));
                int total=readFile(argv[1],initbuff);
                if(total==-1)
                {
                        printf("Error! opening %s",argv[1]);
                        return 0;
                }
                //calculate the share of each node
                share=total/numprocs;
                //now scatter the array to all the processes
                        startwtime = MPI_Wtime();
        }
                MPI_Bcast(&share, 1, MPI_INT, 0, MPI_COMM_WORLD);
                arr=(int*)malloc(sizeof(int)*share);
MPI_Scatter(initbuff,share, MPI_INT,arr,share, MPI_INT,ROOT,MPI_COMM_WORLD);
        //printf("%d's Share=%d\n",myid,share);
        //dump(arr,share,myid);
                maxlocal=maxN(arr,share);
                MPI_Reduce(&maxlocal, &max, 1, MPI_INT, MPI_MAX, ROOT,
MPI_COMM_WORLD );
                if(myid==0)
                {
endwtime = MPI_Wtime();    //now everyone sorts their share
                printf("Maximum = %d\n",max);
                printf("wall clock time = %f\n", endwtime-startwtime);
                }
                fflush(stdout );
                MPI_Finalize();
return 0;
}
```

APPENDIX-II

SAMPLE SHELL SCRIPT TO TEST A PROGRAM

```
declare -a file_sizes=(
'500' '5000' '50000' '500000' '5000000' '50000000'
'400' '4000' '40000' '400000' '4000000' '40000000'
'300' '3000' '30000' '300000' '3000000' '30000000'
'200' '2000' '20000' '200000' '2000000' '20000000'
'100' '1000' '10000' '100000' '1000000' '10000000'
)
i=0
fortest_file in `ls|sort -rn`
do
mpiexec -f ../f ../a.out  $test_file ${file_sizes[$i]}|tee -a $1
echo " executing : mpiexec .././a.out -f ../f $test_file ${file_sizes[$i]}|tee -a $1"
i=`expr $i + 1`
done
```

APPENDIX-III

ABBREVIATIONS USED

CPU	Central Processing Unit, called as the brain of Computer
Multiprocessor	A system having more than one CPU
Multi Core	A system which have more than one core of processing unit in the same processor
CU	Control Unit
PU	Processing Unit
SISD	Single Instruction Stream Single Data Stream
SIMD	Single Instruction Stream Multiple Data Stream
MIMD	Multiple Instruction Stream Multiple Data Stream
MISD	Multiple Instruction Stream Single Data Stream
UMA	Uniform Memory Access
NUMA	Non Uniform Memory Access
COMA	Cache Only Memory Access
NORMA	No Remote Memory Access
MPI	Message Passing Interface
MPICH	MPICH is a high performance and widely portable implementation of the Message Passing Interface (MPI) standard