Programming a Clustered System

Virendra V. Bagade*, Savita Kadam**, Sonali Kadam*** and Pallavi Rege****

Abstract

The Message Passing Interface (MPI) has been a great boon to the development of parallel applications, since it can be used not just on a single vendor's system, but also as a bridging tool to allow multiple systems to be tied together to ensure that the integrity of the programming model is maintained. This Paper presents MPI based implementation of parallel programming that is fundamentally different from sequential code. Parallel programming is centered on finding ways to decompose problems in two basic ways: (i) functional decomposition i.e. multiple program multiple data oriented programming (MPMD) and (ii) Data decomposition i.e. single program multiple data oriented programming (SPMD). Programs can be parallelized in order to run faster. In this paper 80% of a program is parallelized and computational time is reduced to 40% of the time required for the sequential program. Comparisons with existing methods are drawn and the advantages and disadvantages of each are examined. Several examples are illustrated with resulting outputs. This paper works out the best technique in terms of processing power and complexity.

Index Terms: Feature based parallel model, Computational time, Sequential and Parallel programs.

1. Introduction

Towards the start of the computer revolution a typical network topology was to have one main, powerful computer which processes all the information. The "dumb terminal" was basically used as a window to the main frame. These days, each workstation processes its own information and the network is simply a medium by which information is passed between computers. Message Processing Interface (MPI) is a tool that brings all the computers on a network together under one umbrella and effectively turns many workstations in to one virtual high performance parallel processing computer. The Message Passing Interface Forum sets the standard for MPI. It is a library of subroutines/functions which can be used for large problems that demand high computational time (access to more memory) and data that is spread across different data sets. It can also be used for distributed memory machines such as clusters of Unix work stations, clusters of NT/Linux PCs or IBM pSeries.

As the importance of parallel processing and its supporting environments become increasingly realized, some questions are raised concerning its performance, reliability, stability and user-friendliness. This paper attempts to answer these questions. Due to its networking background, Unix has been the preferred platform for network operating systems. However, Microsoft Windows is now becoming a real competitor to Unix. This is due to its multitasking environment, stability and huge popularity. MPI environments have been developed so that Windows-based networks can make use of the benefits due to MPI.

The main goal of parallel programming is to utilize all the processors and minimize the elapsed time of the programs. Using the current software technology, there is no software environment or layer that absorbs the difference in the architecture of parallel computers and provides a single programming model. So, one may have to adopt different programming models for different architectures in order to balance their performance and the effort required for programming.

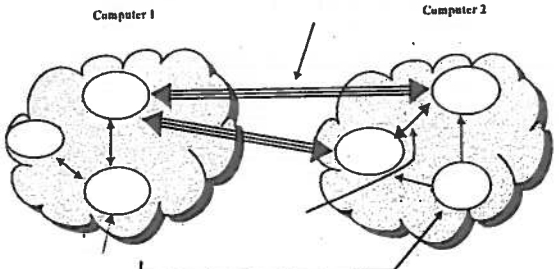

Fig.1: Basic Process of Message Passing Interface.

The basic process view of MPICH is shown in Figure 1. Remote processes communicate via TCP. Processes sharing memory use Windows shared memory Semantics.

2 Models of Parallel Programming

2.1 SMP Based Models

Multi-threaded programs are the best fit with Symmetric Multiprocessor (SMP) Architecture because threads that belong to a process share the available resources. One can either write a multi-thread program using the POSIX threads library (pthreads) or let the compiler generate multithread executables. Generally, the former option places the burden on the programmer, but when done well, it provides good performance because one has complete control over how the programs behave. On the other hand, if the latter option is used, the compiler automatically parallelizes certain types of DO loops, or else, one must add some directives to tell the compiler what is to be done by it. However, one has less control over the behavior of threads [1].

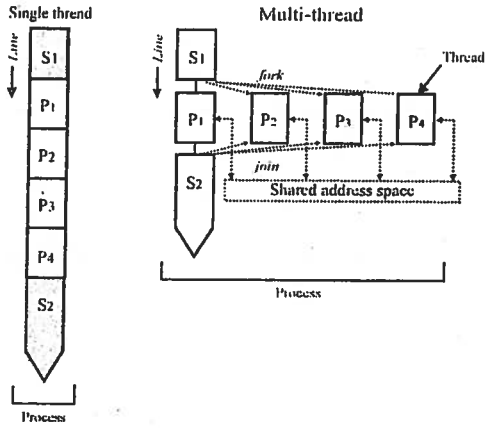

Figure 2: The fork-join model

In Figure 2, the single-thread program processes S1 through S2, where S1 and S2 are inherently sequential parts and P1 through P4 can be processed in parallel. The multi-thread program proceeds in the *fork-join model*. It first processes S1, and then the first thread forks three threads. Here, the term *fork* is used to imply the creation of a thread, not the creation of a process. The four threads process P1 through P4 in parallel, and when finished they are joined to the first thread. Since all the threads belong to a single process, they share the same address space and it is easy to reference data that other threads have updated. There is some overhead in forking and joining threads.

2.2 MPP Based on Uniprocessor Nodes

If the address space is not shared among nodes, parallel processes have to transmit data over an interconnecting network in order to access data that other processes have updated. High Performance Fortran (HPF) may do the job of data Transmission for the user, but it does not have the flexibility that hand-coded message-passing programs have, since the class of problems that HPF resolves is limited.

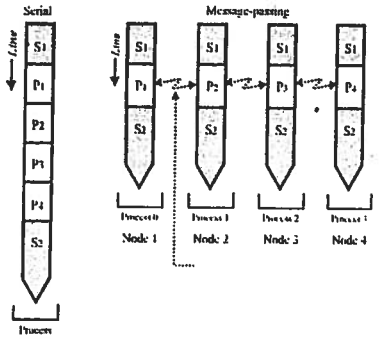

Figure 3: Message-Passing

Figure 3 illustrates how a message-passing program runs. One process runs on each node and the processes communicate with each other during the execution of the parallelizable part, P1-P4. The figure shows links between processes on the adjacent nodes only, but each process communicates with all the other processes in general. Due to the communication overhead, workload unbalance, and synchronization, time spent for recessing each of P1- P4 is generally longer in the message-passing program than in the serial program. All processes in the message-passing program are bound to S1 and S2. Image gets set to something appropriate. In the forward mapping case, some pixels in the destination

might not get painted, and would have to be interpolated. One can calculate the image deformation as a reverse mapping.

2.3 Comparison of SPMD and MPMD

When one runs multiple processes with message passing, there are further categorizations regarding how many different programs are cooperating in parallel execution. In the Single Program Multiple Data (SPMD) model, there is only one program and each process uses the same executable working on different sets of data (Figure3 (a)). On the other hand, the Multiple Programs Multiple Data (MPMD) model uses different programs for different processes, but the processes collaborate to solve the same problem.

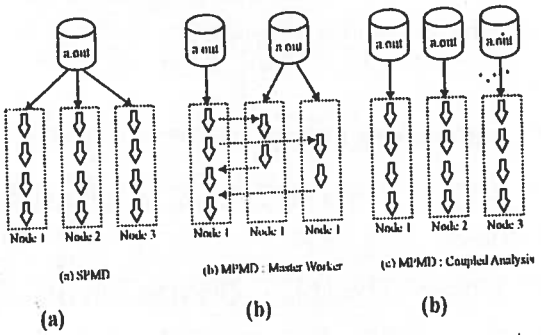

Fig.4 SPMD and MPMD

Figure 4(b) shows the master/worker style of the MPMD model, where a.out is the master program, which dispatches jobs to the worker program, b.out. There are several workers serving a single master. In the coupled analysis (Figure 4(c)), there are several programs (a.out, b.out, and c.out), and each program does a different task, such as structural analysis, fluid analysis, and thermal analysis. Most of the times, they work independently, but once in a while, they exchange data to proceed to the next time step.

2.4 Why message passing is necessary for parallelization

Sequential program that reads data from a file, does some computation on the data, and writes the data to a file. In figure 6, white circles, squares, and triangles indicate the initial values of the elements; and black objects indicate the values after they are processed. In the SPMD model, all the processes execute the same program. To distinguish between processes, each process has a unique integer called *rank*. One can let the processes behave differently by using the value of rank. Hereafter, the process whose rank is r is referred to as process r.

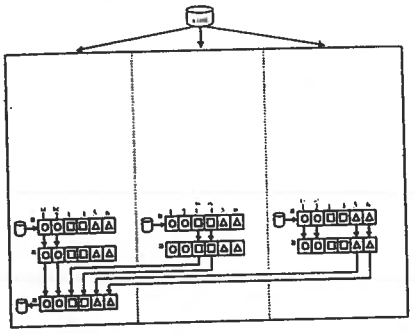

Figure 6. Parallel Program

In figure 6, all the processes read the array in Step 1 and get their own rank in Step 2. In Steps 3 and 4, each process determines which part of the array it is in charge of, and it processes that part. After all the processes have finished in Step 4, none of the processes have all of the data, which is an undesirable side effect of parallelization. It is the role of message passing to consolidate the processes separated by the parallelization. Step 5 gathers all the data to a process and that process writes the data to the output file. To summarize:

The purpose of parallelization is to reduce the time spent for computation. Ideally, the parallel program is p times faster than the sequential program, where p is the number of processes involved in the parallel execution, but this is not always achievable.

Message-passing is the tool to consolidate what parallelization has separated. It should not be regarded as the parallelization itself.

2.5 MPI Program Structure

- Handles
- MPI Communicator
- MPI_Comm_world
- Header files

- MPI function format
- Initializing MPI
- Communicator Size
- Process Rank
- Exiting MPI

Handles:

- MPI controls its own internal data structures
- MPI releases "handles" to allow programmers to refer to these
- C handles are of defined typedefs

MPI Communicator

- Programmer view: group of processes that are allowed to communicate with each other
- All MPI communication calls have a communicator argument

MPI_COMM_WORLD Communicator

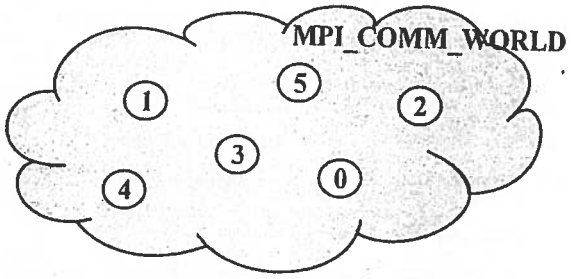

Figure 7.

Header Files

- MPI constants and handles are defined here in C language : #include <mpi.h>

MPI Function Format

```
error = MPI_Xxxxx(parameter,...);
MPI_Xxxxx(parameter,..);
```

Initializing MPI

- must be the first routine called (only once) int MPI_Init(int *argc, char ***argv)

Communicator Size

- How many processes are contained within a Communicator?

MPI_Comm_size (MPI_Comm comm, int *size)

Process Rank

- Process ID number within the communicator Starts with zero and goes to (n-1) where n is the number of processes requested
- Used to identify the source and destination of messages
- Also used to allow different processes to execute different code simultaneously

MPI_Comm_rank(MPI_Comm comm, int *rank)

Exiting MPI

- Must be called last by "all" processes MPI_Finalize ()

Sample program

```
#include <mpi.h>
void main(int argc, char *argv[])
{int rank, size;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
/* ... Your code here ... */
MPI_Finalize();}
```

2.6 Parallelization and Computational Time

One can parallelize a program in order to run it faster. How much faster will the parallel program run? Suppose that in terms of running time, a fraction *P* of a program can be parallelized and that the remaining *1-P* cannot be parallelized. In the ideal situation, if one executes the program using processors, the parallel running time will be (1 $p + p/n$) of the serial running time. This is a direct consequence of Amdahl's law applied to an ideal case of parallel execution. For example, if 80% of a program can be parallelized and there are four processors, the parallel running time will be 10.8+ 0.8 / 4 = 0.4, that is, 40% of the serial

running time as shown in Figure 8.

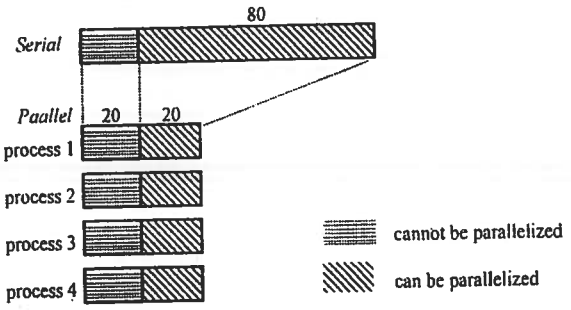

Figure 8.

Since 20% of the program cannot be parallelized, one gets only 2.5 times speed-up using the four processors. For this program, the parallel running time is never shorter than 20% of the serial running time (five times speed-up) even if infinitely many processors are used. According to Amdahl's law, it is important to identify the fraction of a program that can be parallelized. When a parallel program is run, there is a communication overhead and a workload imbalance among processes in general. Therefore, the running time will be as Figure 9 shows.

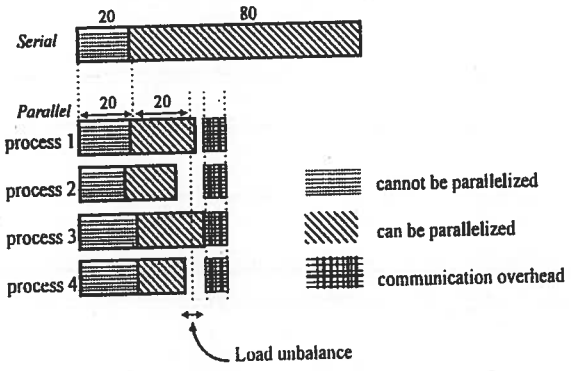

Fig.9 Parallel Speed-Up: An Actual Case

What can be done to balance the workload of processes? Several measures can be taken according to the nature of the program. Changing the distribution of matrices from block distribution to cyclic distribution is one of them. The communication time is expressed as follows: The latency is the sum of sender overhead, receiver overhead, and time of flight, which is the time for the first bit of the message to arrive at the receiver. Using this formula, the effective bandwidth is calculated as follows:

The effective bandwidth approaches the network bandwidth when the message size grows towards infinity. It is clear that the larger the message is, the more efficient the communication becomes.

3. Conclusion and Results

In terms of performance, MPICH is clearly the most efficient. It uses optimal network bandwidth and establishes connections between nodes in the least amount of time. Users who are new to MPI should have no problem with this interface, provided the environment has been correctly installed. Overall, this environment should appeal to the expert. It offers excellent performance and accommodates scalability of the network with the simple to use, GUI Resultant out put for matrix

For Parallel Program
Here is the result matrix = [1000][1000]
Wall clock time = 45.94387, Finished writing log file.
For Sequential Program
Here is the result matrix=[1000][1000]
Wall clock time = 33.10156, Finished writing log file.
Calculate the delay in the Network by sending and receiving only rows & column without calculation
Here is the result matrix=[1000][1000]
Wall clock time = 26.4293, Finished writing log file
Computational time for Parallel Program is Total time-Network delay < Sequential Time
45.94 - 26.421=19.52 i.e. < Sequential program time

REFERENCES

[1] *Message Passing Interface Standard.* [Online] Availablefrom:http://www.unix.mcs.anl.gov/mpi/mpich [Accessed: April 2000]
[2] Microsoft Windows NT server White Paper. http://www.microsoft.com/ntserver/nts/exec/default.asp [Accessed: Nov '99]
[3] MPICH from Argonne,

http://www.mcs.anl.gov/mpi/mpich
[4] Microsoft: *Windows NT server*. White Paper:
http://www.microsoft.com/ntserver/nts/exec/default.asp
[5] NPAC REU Program Labs and Tutorials, -
http://www.npac.syr.edu/REU/reu96/Labs/MPI/mpilab.html

***Sinhagad Engineering College, Lonavala**
**** Computronics, Pune**
*****Bharati Vidyapeeth College of Engineering for Women, Pune-43**
****** VIIT College of Engineering, Pune**