FAULT TOLERANCE IN THE COMMUNICATION PATTERNS USING MPI

A DISSERTATION

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

of

MASTER OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

By

ANKUSH AGARWAL

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE-247 667 (INDIA)

JUNE, 2010

CANDIDATE'S DECLARATION

I hereby declare that the work, which is being presented in the dissertation entitled, "Fault Tolerance in the communication patterns using MPI", which is submitted in the partial fulfillment of the requirements for the award of degree of Master of Technology in Information Technology, submitted in the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee (India), is an authentic record of my own work carried out under the guidance of Dr. A. K. Sarje, Professor, Department of Electronics & Computer Engineering, Indian Institute of Technology Roorkee.

The matter embodied in the dissertation report to the best of our knowledge has not been submitted for the award of any other degree elsewhere.

[illegible]

Dated : June, 2010 (Ankush Agarwal)

Place : Roorkee

CERTIFICATE

This is to certify that the above statement made by the candidate is correct to the best of our knowledge.

[illegible] 30/6/2010

Dr. A. K. Sarje

Professor

Dept. of E&CE

IIT Roorkee

ACKNOWLEDGEMENTS

It gives me immense pleasure to take this opportunity to thank and express my deep sense of gratitude to my guide Dr. A. K. Sarje, Professor, Department of Electronic and Computer Engineering, Indian Institute of Technology Roorkee, for his valuable guidance. I would also like to thank my guide for his encouragement during the course of this work.

I would like to thank again to my guide for making its resources available for the purpose of this work and the staff of the Department of Electronics & Computer Engineering for their cooperation during this dissertation work.

My special sincere heartfelt gratitude to my friends, whose best wishes, support and encouragement has been a constant source of strength to me during the entire work.

Finally, I would also like to thank all my classmates and seniors for their support and valuable suggestions.

[illegible]

(Ankush Agarwal)

ABSTRACT

Recent trend is of high performance computing, in which message rate is of main measure. This message rate depends on communication pattern. The continued growth in platform scale, combined with emerging application area, are pushing platform to support increasing message rates. Best case message throughput has grown in hardware generation due to growing clock rates and on patterns.

Now, with the growing scale of high performance computing, fault tolerance has become the major issue. In computing work, there might be a chance of fault which will result in a false data. There are various techniques available to handle faults. A method named message logging is used to this overhead. There are various other techniques that are used to overcome fault tolerance.

Here I used an approach similar to the approaches of fault tolerance that exist. This approach is slightly differing with other existing approaches. With this approach we can handle faults in the communication patterns. For this, we have to change the parallel code of the communication patterns a bit.

Table of Contents

List of Figures

CHAPTER 1

INTRODUCTION

1.1 HPC – High Performance Computing

High-performance computing (HPC) uses super computers and computer clusters to solve advanced computation problems. High Performance Technical Computing (HPTC), generally refers to the engineering applications which is used in cluster based computing. HPC can also be applied to business uses of cluster based super computers such as data warehouse, transaction processing.

HPC can also be used as a synonym for supercomputing, but in other scenarios super computer is used to refer to a more powerful set of high performance computers.

As machine sizes are growing day by day, tends to thousand of nodes, so it is important that an application must utilize the machine resources effectively. So it is important to know how an application can utilize machine resources.

There are various measures on which we can find out the efficiency and effectiveness of cluster, but the three most commonly measured metrics are bandwidth, latency rate and message rate. However other parameters such as independent progress, host overhead, etc can also impact application performance. The difficult question is: how should the data be measured given that interconnect performance can vary dramatically based on the operating conditions of the application of the application using it? As an example, the average length of the MPI message queues will impact both the latency rate and message rate[2] that the network can deliver.

Of the three main measures of interconnect performance, message rate seems to be of measure. Message rate is the measure of how many distinct messages a node can send and/or receive in a given time period, and is often referred to as message throughput. For example a massage rate of 1 million messages per second would only be able to sustain a bandwidth of 8 MB/s for messages of size 8 bytes and a bandwidth of 1GB/s for messages of size 1 KB. Thus the message rate determines the minimum message size which can saturate the bandwidth of a given network.

1.2 Fault Tolerance

Fault tolerance is the property that enables a system to continue operating even in case of failure of machine component. In this issue, performance of a machine decreases. The decrease in performance is proportional to the occurrences of failure.

As most HPC applications are using the Message Passing Interface (MPI) [3] to manage data transfers, introducing failure recovery features inside the MPI library automatically benefits a large range of applications. One of the most popular automatic fault tolerant techniques, coordinating checkpoint, builds a consistent recovery set [4],[5]. Message logging is an alternative approach designed to avoid coordination, in order to recover faster from failures at the expense of a higher overhead on communications. From previous experiments, it has been proved that message logging is expected to be better than coordinated checkpoint when the Mean Time Between Failure (MTBF) is shorter.

However, the model of message logging was recently refined to match the reality of high performance network interface cards, where message receptions are decomposed in multiple interdependent events [6].

1.3 MPI – Message Passing Interface

The generic form of message passing in parallel processing is the message passing interface (MPI), which is used as a medium of communication. Most of the parallel programming languages differ in view of the address space. Message Passing Interface (MPI) was designed for writing applications and libraries for distributed memory environments. In message passing, data is moved from the address space of one to that of other by means of a cooperative operation such as a send/receive pair. The restriction sharply distinguishes the message passing model from the shared memory model. In shared memory model, processes have access to a common pool of memory and can simply perform ordinary memory operations (load from, store into) on some set of address.

1.4 Problem Statement

In HPC, there are some communications patterns on the basis of which a node can communicate with others. The list of the various communication patterns are listed below.

- Single Direction Communication
- Pair based Communication.

- Pre-posted Communication
- All start communication

Now during communication, there may be a possibility of software fault. A fault results in the false outcome. So as to overcome with this problem, we need to introduce fault tolerance in these communication patterns.

Up to now, fault tolerance has not been applied to these communication patterns. So we introduce fault tolerance in those communication patterns for the safe communication.

Hence with the use of fault tolerance, we can handle these faults and be able to reach to the correct final outcome.

CHAPTER 2

BACKGROUND STUDY

2.1 Models of parallel computing

Interconnection problem in modern HPC systems have a variety of performance parameters which impact application performance. Many metrics have been developed to try to describe this performance. One attempt is to model these parameters and access their impact on applications led to the development of the LogP model [8][9]. Follow up work incorporating an analysis of long messages generated the LogGP [10] model. More recent work has developed techniques for measuring the LogGP parameters on modern networks [11]. In addition to these modeling efforts, there is a body of work on benchmarks to measure various aspects of interconnect performance. NetPIPE [12] and Netperf [13], Ohio State's OMB [14], and Intel's MPI benchmark [15] can be used to measure network latency and bandwidth.

In addition to these general measures, more specific work has been done to quantify MPI performance. The OSU benchmark suite [14] has micro-benchmarks for measuring latency, streaming bandwidth and message rate, among others. Further research has worked to measure additional areas of interconnect performance such as overlap [16]. While others have looked at the impact of queue lengths [1] and overhead buffer re-use [17].

Unfortunately, many of the network micro-benchmarks measure performance in idealized conditions that do not match those present during application execution. Additionally, it is not uncommon for hardware and MPI developers to optimize the most common micro-benchmarks. Many areas where such optimizations improve micro-benchmark performance, but have little to no impact on applications performance have been identified in [18]. One issue identified with traditional micro-benchmarks is that the only operation performed is the sending and receiving of data, which means that the MPI data structures are always in cache. This is not the typical operating environment for real applications, which will intersperse communication with computation.

Another area of specific concern is message coalescing. Both Open MPI [19] and MVAPICH [20] coalesce short messages when running with software flow control. There are many ways to coalesce messages; the simplest method implemented today works only for zero-byte messages with identical MP envelope information. However, for coalescing to be generally useful to applications, it must work on messages with data sizes greater than zero and across messages with non-identical tags.

In cluster, a master node can communicate with other slave node/s on the basis of some pre-defined patterns. Each of the communication patterns works independently from others. These communication patterns transfer data from one node to other with coordinated send-receive pair.

2.2 Fault Tolerance

Fault-tolerant describes a system or component designed so that, in the event that a component fails, a backup component or procedure can immediately take its place with no loss of service. Fault tolerance can be provided with software, or embedded in hardware, or provided by some combination.

HPC applications are using the Message Passing Interface (MPI) [3] which manages data transfer. Now introducing failure recovery features inside the MPI library automatically benefits in a large range of applications. One of the most popular automatic fault tolerant techniques, coordinating checkpoint, builds a consistent recovery set [4],[5]. Message logging is an alternative approach designed to avoid coordination. From previous experiments, it has been proved that message logging is expected to be better than coordinated checkpoint when the Mean Time Between Failure (MTBF) is shorter.

2.2.1 Message Logging

Message logging is defined in the more general model of message passing distributed systems. Communications between processes are considered explicit: processes explicitly request sending and receiving messages; and a message is considered as delivered only when the receive operation associated with the data movement is complete. Additionally, from the perspective of the application each communication channel is FIFO, but there is no particular order on messages traveling along different channels. The execution model is pseudo-synchronous; there is no global shared clock among processes but there is some (potentially unknown) maximum propagation delay of messages in the network. An intuitive interpretation is to say the system is asynchronous. Failures can affect both the processes and the network. Usually, network failures are managed by some CRC mechanism. Therefore, the considered failure model is definitive crash failures, where a failed process completely stops sending any subsequent message.

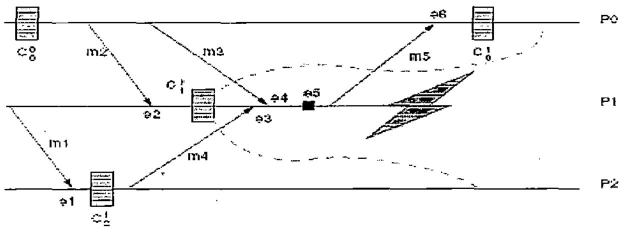


Figure 2.1

- **Events :** Each computational or communication step of a process is an event. An execution is an alternate sequence of events and process states, with the effect of an event on the preceding state leading the process to the new state. As the system is basically asynchronous, there is no direct time relationship between events occurring on different processes.

  These events can be classified into two categories: deterministic and non-deterministic. An event is deterministic when from the current state there is only one possible outcome state for this event. On the contrary, if an event can result in several different states, then it is non-deterministic. Examples of deterministic events are internal computations and message emissions, which follow the code-flow. Examples of nondeterministic events are message receptions, which depend on time constraints on message deliveries.

- **Checkpoints and Inconsistent States :** Checkpoints are used to recover from failures. The recovery line is the configuration of the application after some processes have been reloaded from checkpoints. Unfortunately, check pointing a distributed application is not as simple as storing each single process image without any coordination, as illustrated by the example execution of figure 1. When process P1 fails, it rolls back to checkpoint C11 . Messages from the past crossing the recovery line (m3,m4) are in transit messages; the restarted process will request their reception while the source process never sends them again, thus it is needed to save the messages. Messages from the future crossing the recovery line (m5) are orphan; following the Lamport relationship current state of P0 depends on reception of m5 and by transitivity on any event that occurred on P1 since C11 (e3, e4, e5). Since the channels between P0 and P1 and between P2 and P1 are asynchronous, the reception of m3 and m4 could occur in coalesce short messages when running with software flow control. There are many ways to coalesce messages; the simplest method implemented today works only for zero-byte messages with identical MP envelope information. However, for coalescing to be generally useful to applications, it must work on messages with data sizes greater than zero and across messages with non-identical tags.

In cluster, a master node can communicate with other slave node/s on the basis of some pre-defined patterns. Each of the communication patterns works independently from others. These communication patterns transfer data from one node to other with coordinated send-receive pair.

2.2 Fault Tolerance

Fault-tolerant describes a system or component designed so that, in the event that a component fails, a backup component or procedure can immediately take its place with no loss of service. Fault tolerance can be provided with software, or embedded in hardware, or provided by some combination.

HPC applications are using the Message Passing Interface (MPI) [3] which manages data transfer. Now introducing failure recovery features inside the MPI library automatically benefits in a large range of applications. One of the most popular automatic fault tolerant techniques, coordinating checkpoint, builds a consistent recovery set [4],[5]. Message logging is an alternative approach designed to avoid coordination. From previous experiments, it has been proved that message logging is expected to be better than coordinated checkpoint when the Mean Time Between Failure (MTBF) is shorter.

2.2.1 Message Logging

Message logging is defined in the more general model of message passing distributed systems. Communications between processes are considered explicit: processes explicitly request sending and receiving messages; and a message is considered as delivered only when the receive operation associated with the data movement is complete. Additionally, from the perspective of the application each communication channel is FIFO, but there is no particular order on messages traveling along different channels. The execution model is pseudo-synchronous; there is no global shared clock among processes but there is some (potentially unknown) maximum propagation delay of messages in the network. An intuitive interpretation is to say the system is asynchronous. Failures can affect both the processes and the network. Usually, network failures are managed by some CRC mechanism. Therefore, the considered failure model is definitive crash failures, where a failed process completely stops sending any subsequent message.

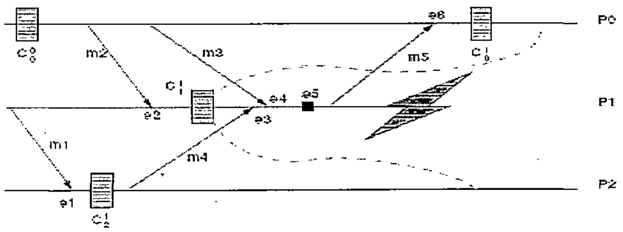


Figure 2.1

- **Events :** Each computational or communication step of a process is an event. An execution is an alternate sequence of events and process states, with the effect of an event on the preceding state leading the process to the new state. As the system is basically asynchronous, there is no direct time relationship between events occurring on different processes.

  These events can be classified into two categories: deterministic and non-deterministic. An event is deterministic when from the current state there is only one possible outcome state for this event. On the contrary, if an event can result in several different states, then it is non-deterministic. Examples of deterministic events are internal computations and message emissions, which follow the code-flow. Examples of nondeterministic events are message receptions, which depend on time constraints on message deliveries.

- **Checkpoints and Inconsistent States :** Checkpoints are used to recover from failures. The recovery line is the configuration of the application after some processes have been reloaded from checkpoints. Unfortunately, check pointing a distributed application is not as simple as storing each single process image without any coordination, as illustrated by the example execution of figure 1. When process P1 fails, it rolls back to checkpoint C11 . Messages from the past crossing the recovery line (m3,m4) are in transit messages; the restarted process will request their reception while the source process never sends them again, thus it is needed to save the messages. Messages from the future crossing the recovery line (m5) are orphan; following the Lamport relationship current state of P0 depends on reception of m5 and by transitivity on any event that occurred on P1 since C11 (e3, e4, e5). Since the channels between P0 and P1 and between P2 and P1 are asynchronous, the reception of m3 and m4 could occur in a different order during re-execution, leading during the recovery to a state of P1 that diverges from the initial execution. As the current state of P0 depends on states P1 can no longer reach, the overall state of the parallel application after the recovery is inconsistent. Checkpoints leading to an inconsistent state are useless and must be discarded. In the worst case all checkpoints are useless and the computation may have to be restarted from the beginning.

- **Event Logging :** In event logging, processes are considered as Piecewise deterministic: only sparse non-deterministic events occur separating large parts of deterministic computation. Considering that non-deterministic event outcomes, called determinants, are committed during the initial execution into some safe repository, a recovering process is able to replay exactly the same order for all non-deterministic events, and therefore, it is able to reach exactly the same state as prior to the failure. Furthermore, message logging considers the network as the only source of non-determinism and only logs the relative ordering of messages from different senders (e3, e4 in figure 1). The sufficient condition to define a consistent global state, from where a recovery can be successful, is that a process must never depend on an unlogged non-deterministic event from another process.

- **Synchronicity of Event Logging :** Pessimistic message logging is the most synchronous event logging technique. It ensures the always no-orphan condition: all the previous non-deterministic events of a process must be logged before a process is allowed to impact the rest of the system. Therefore any process has to ensure that every event is safely logged before any MPI send can proceed. Since no orphan process can be created, only the failed processes have to restart after a failure. In order to improve latency, the no-orphan condition can be relaxed. Causal message logging piggybacks unlogged events on outgoing messages. Then any process always depends on events either logged or known locally. Optimistic message logging pushes one step further; non-deterministic events are buffered in the process memory and logged asynchronously. While message sending is never delayed, the consequence is that a message sent by a process may depend on an unlogged event and may become orphaned. Thus a recovery protocol is needed to detect orphan messages and to recover the application in a consistent global state after a failure. To be able to detect orphan messages, dependencies between non-deterministic events need to be tracked during the entire execution; dependency information must be piggybacked on application messages.

- **Sender-Based Logging :** Event logging only saves events in the remote repository, without storing the message payload. However, when a process is recovering, it needs to replay any reception that happened between the last checkpoint and the failure. Therefore, all the in transit message payload needs to be saved (m3,m4 in figure1). During normal execution, every outgoing message is saved in the sender's volatile memory: a mechanism called sender-based message logging. This allows the surviving processes to serve past messages to recovering processes on demand, without rolling back. Unlike events, sender-based data do not require stable or synchronous storage. Should a process holding useful sender-based data crash, the recovery procedure of this process replays every outgoing send and thus rebuilds the missing messages.

2.3 MPI – MESSAGE PASSING INTERFACE

Message Passing Interface (MPI) is a specification for an API that allows many computers to communicate with one another. It is used in computer clusters and supercomputers.

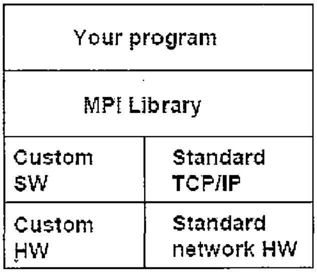


FIGURE 2.2

MPI is a language-independent communications protocol used to program parallel computers. Both point-to-point and collective communication are supported. MPI "is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation."[21] MPI's goals are high performance, scalability, and portability. MPI remains the dominant model used in high-performance computing today[22].

MPI is not sanctioned by any major standards body; nevertheless, it has become a de facto standard for communication among processes that model a parallel program running on a distributed memory system. Actual distributed memory supercomputers such as computer clusters often run these programs. The principal MPI-1 model has no shared memory concept, and MPI-2 has only a limited distributed shared memoryconcept. Nonetheless, MPI programs are regularly run on shared memory computers. Designing programs around the MPI model (as opposed to explicit shared memory models) has advantages on NUMA architectures since MPI encourages memory locality.

Although MPI belongs in layers 5 and higher of the OSI Reference Model, implementations may cover most layers of the reference model, with socket and TCP being used in the transport layer.

Most MPI implementations consist of a specific set of routines (i.e., an API) callable from Fortran, C, C++ or Java and from any language capable of interfacing with such routine libraries. The advantages of MPI over older message passing libraries are portability (because MPI has been implemented for almost every distributed memory architecture) and speed (because each implementation is in principle optimized for the hardware on which it runs).

2.4 Concepts of MPI

MPI provides a rich range of capabilities. The following concepts help in understanding and providing context for all of those capabilities and help the programmer to decide what functionality to use in their application programs. There are seven basic concepts of MPI, three of which are unique to MPI-2.

- **Communicator** : Communicators are objects connecting groups of processes in the MPI session. Within each communicator each contained process has an independent identifier and the contained processes are arranged in an ordered topology. MPI also has explicit groups, but these are mainly good for organizing and reorganizing subsets of processes. MPI understands single group intra-communicator operations, and bipartite (two-group) inter-communicator communication. Single group operations are most prevalent in MPI-1 whereas a bipartite operation plays a major role in MPI-2.

  Communicators can be partitioned using several commands in MPI, these commands include a graph-coloring-type algorithm called MPI_COMM_SPLIT, which is commonly used to derive topological and other logical sub-groupings in an efficient way.

- **Point-to-point basics :** A number of important functions in the MPI API involve communication between two specific processes. A much used example is the MPI_Send interface, which allows one specified process to send a message to a second specified process. Point-to-point operations are useful in irregular communication, for example, a data-parallel architecture in which each processor routinely swaps regions of data with specific other processors between calculation steps, or a master-slave architecture in which the master sends new task data to a slave whenever the previous task is completed.

  MPI-1 specifies mechanisms for both blocking and non-blocking point-to-point communication mechanisms, as well as the so-called 'ready-send' mechanism whereby a send request can be made only when the matching receive request has already been made.

- **Collective basics :** Collective functions in the MPI API involve communication between all processes in a process group (which can mean the entire process pool or a program-defined subset). A typical function is the MPI_Bcast call (short for "broadcast"). This function takes data from one specially identified node and sends that message to all processes in the process group. A reverse operation is the MPI_Reduce call, which is a function designed to take data from all processes in a group, performs a user-chosen operation (like summing), and store the results on one individual node. These types of calls are often useful at the beginning or end of a large distributed calculation, where each processor operates on a part of the data and then combines it into a result.

- **Derived Data types :** Many MPI functions require that you specify the type of the data which is sent between processors. This is because these arguments to MPI functions are variables, not defined types. If the data type is a standard one, such as, int, char, double, etc., you can use predefined MPI data-types such as MPI_INT, MPI_CHAR, MPI_DOUBLE. Suppose your data is an array of ints and all the processors want to send their array to the root with MPI_Gather.

  Here is a C example of how to do it:

```
int array[100];
int root, total_p, *receive_array;
MPI_Comm_size(comm, &total_p);
receive_array=(int *) malloc(total_p*100*sizeof(int));
```

```
MPI_Gather(array, 100, MPI_INT, receive_array, 100, MPI_INT,
root, comm);
```

However, you may instead wish to send your data as one block as opposed to 100 ints. You can do this by defining a continuous block derived data type.

```
MPI_Datatype newtype;
MPI_Type_contiguous(100, MPI_INT, &newtype);
MPI_Type_commit(&newtype);
MPI_Gather(array, 1, newtype, receive_array, 1, newtype, root,
comm);
```

Sometimes, your data might be a class or a data structure. In this case, there is not a predefined data type and you have to create one. You can make an MPI derived data type from MPI_predefined data types, by using MPI_Type_create_struct, which has the following format:

```
int MPI_Type_create_struct(int count, int blocklen[], MPI_Aint
disp[], MPI_Datatype type[], MPI_Datatype *newtype)
```

where,
count - number of blocks,
blocklen[] - number of elements in each block (array of integer),
disp[] - byte displacement of each block (array of integer),
type[] - type of elements in each block (array of handles to datatype objects).

- **One-sided Communication (MPI-2)** : 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 for synchronizing this communication - global, pairwise, and remote locks - as the specification does not guarantee that these operations have taken place until a synchronization point.

- **Dynamic Process Management (MPI-2)** : The key aspect of this MPI-2 feature 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 inter-communicator the function returns. MPI_Comm_spawn_multiple is an alternate interface that allows the different instances spawned to be different binaries with different arguments.[23]

- **MPI I/O (MPI-2)** : The Parallel I/O feature introduced with MPI-2, is sometimes shortly called MPI-IO,[24] and refers to a collection of functions designed to allow the difficulties of managing I/O on distributed systems to be abstracted away to the MPI library, as well as allowing files to be easily accessed in a patterned fashion using the existing derived data-type functionality. The little research has been done on this feature indicates the difficulty for good performance. For example, some implementation of sparse matrix-vector multiplications using the MPI I/O library disastrously fall in efficient parallelization.

2.5 Implementation of MPI

Cluster computing is the technique of linking two or more computers into a network (usually through a local area network) in order to take advantage of the parallel processing power of those computers. MPI is a widely used library that facilitates communication between parallel programs written in C, C++, FORTRAN, Python etc.

The concept of a cluster involves taking two or more computers and organizing them to work together to provide higher availability, reliability and scalability than can be obtained by using a single system. When failure occurs in a cluster, resources can be redirected and the workload can be redistributed. The use of MPI libraries have greatly helped in making it easier to utilize the power of clusters as the same implementation that runs on a multi-core system can also run on a cluster.

2.6 Key MPI Functions and Constants

- MPI_Init (int *argc, char ***argv)
- MPI_Finalize (void)

- MPI_Comm_rank (MPI_COMM comm, int *rank)
- MPI_Comm_size (MPI_COMM comm, int *size)
- MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- MPI_Recv (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
- MPI_CHAR, MPI_INT, MPI_LONG, MPI_BYTE
- MPI_ANY_SOURCE, MPI_ANY_TAG

2.7 MPI Communicators, contexts, groups

A distinguishable feature of the MPI standard is that it includes a mechanism for creating separate worlds of communication, accomplished through communicators, contexts and groups.

- A communicator specifies a group of processes that will conduct communication operations with in a specified context without affecting or being affected by operations occurring in other group or contexts elsewhere in a program.
- Define communication domain of a communication operation : set of processes that are allowed to communicate among themselves.
- Initially all in MPI_COMM_WORLD
- A group is an ordered collection of processes. Each processor has a rank in the group, the rank runs from 0 to n-1. A process can belong to more than one group; its rank in one group has nothing to do with its rank in other group. A context is the internal mechanism by which a communicator guarantees a safe communication space to the group.
- Communicator provide a caching mechanism, which allow an application to attach attribute to communicators. Attributes can be user data or any other kind of information.

Example

```
main (int argc, char *argv[])
{
        MPI_Init(&argc, &argv);
        ...
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        if (myrank == 0)
                master();
```

```
            else
                    slave();
            ...
            MPI_Finalize();
    }
```

2.8 MPI Message Tag

- Used to differentiate between different types of messages being sent.
- Message tag is carried out with in message and used in both send and receive calls.
- If special type matching is not required, a wild card message tag is used, so that the receive will match with any send.

2.9 Flavors of send/receives

- Synchronous message passing
  - Send/Receive routines that return when message transfer completed.
  - Synchronous Send wait until complete message can be accepted by receiving process before sending the message.
  - Synchronous Receive wait until the message it is expecting arrives.
  - Synchronous routine perform two actions: transfer data and synchronize processes
- Asynchronous message passing
  - Send/Receive do not wait for actions to complete before returning
  - Usually require local storage for messages
  - In general, they do not synchronize processes but allow processes to move forward sooner.

2.10 MPI Blocking and Non-Blocking

- Blocking - return after local actions complete, though the message transfer may not have been completed
- Non-blocking - return immediately
  - Assumes that data storage to be used for transfer is not modified by subsequent statements prior to being used for transfer.
  - Implementation dependent local buffer space is used for keeping message temporarily.

2.11 MPI Communication Modes

- Standard Mode
  - Does not assume that corresponding receive routine has been reached.
  - Does not specify whether messages are buffered
- Buffered Mode
  - Send may start and return before matching receive is reached.
  - Necessary to specify buffer space.
  - Can be done using the MPI routine MPI_Buffer_attach()
- Synchronous Mode
  - Send and receive can start before each other but can only complete together
- Ready Mode
  - Send can only start if matching receive has already been reached

2.11.1 Modes of Blocking MPI Sends

- Standard mode: MPI_Send
- Buffered mode: MPI_Bsend
- Synchronous mode: MPI_Ssend
- Ready mode: MPI_Rsend
- MPI_Recv works for all send modes
- Buffered mode requires user to provide buffer space using MPI_Buffer_attach

2.11.2 Modes of Non-Blocking MPI Sends

- Standard mode : MPI_Isend
- Buffered mode : MPI_Ibsend
- Synchronous mode : MPI_Issend
- Ready mode : MPI_Irsend
- MPI_Irecv non-blocking receive works for all.
- Functions to test or wait for completion
  - MPI_Test
  - MPI_Wait

2.11.3 MPI Group Communication

- MPI also provides routines that sends messages to a group of processes or receives messages from a group of processes

- Not absolutely necessary for programming
- More efficient than separate point to point routines.

- Examples : broadcast, multicast, gather, scatter, reduce, barrier
  - MPI_Bcast, MPI_Reduce, MPI_Allreduce, MPI_Alltoall, MPI_Scatter, MPI_Gather, MPI_Barrier

Broadcast

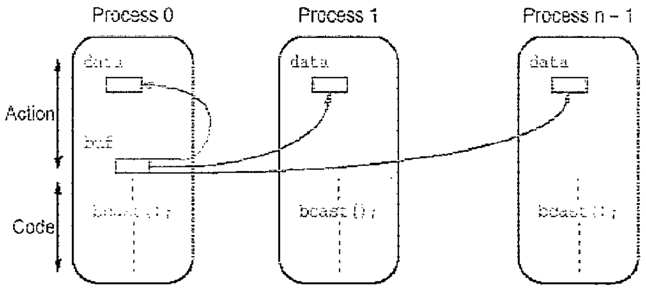


FIGURE 2.3

MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm Comm)

Scatter

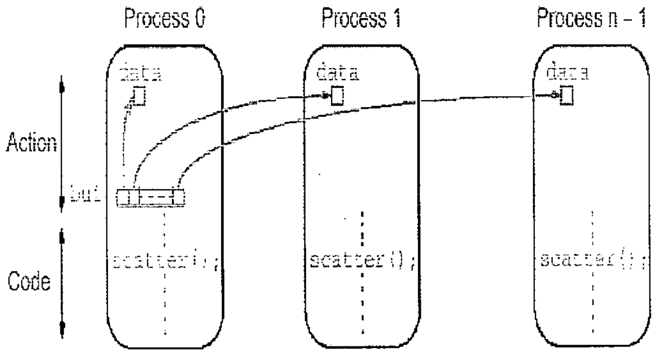


FIGURE 2.4

Gather

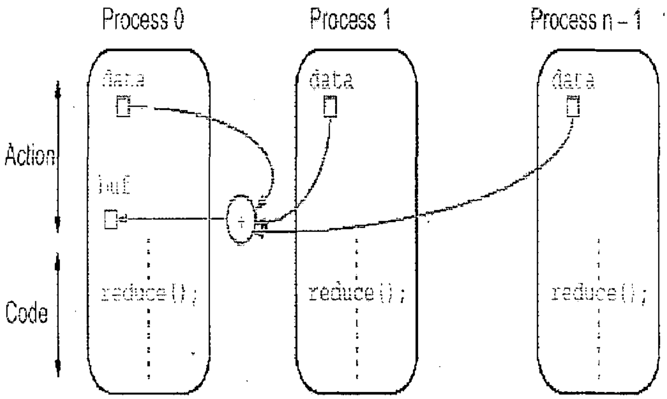


FIGURE 2.5

Reduce

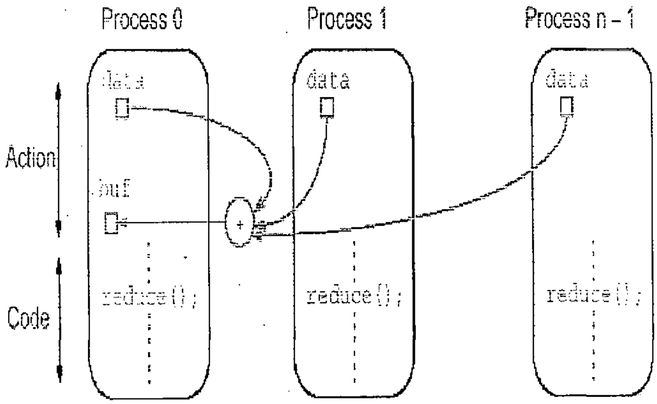


FIGURE 4.5

MPI_Reduce (void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

2.12 MPI Safe Message Passing

- Combined send/receive routine MPI_Sendrecv()
  - Guarntee not to deadlock
- Buffered send MPI_Bsend()
  - user provides explicit storage space
- Non blocking routines MPI_Isend(), MPI_Irecv()
  - return immediately
  - separate routine used to determine whether message has been received (MPI_Wait() etc)

2.13 Timing with MPI

- MPI_Wtime
  - elapsed time in seconds since some arbitrary point in past.
  - Return a double value.
- Clock values for different processes are not necessarily comparable or synchronized

2.14 Example programs of MPI

Here is a "Hello World" program in MPI written in C. In this example, we send a "hello" message to each processor, manipulate it trivially, send the results back to the main process, and print the messages out.

```
/*
/*"Hello World" Type MPI Test Program
*/
#include <mpi.h>
#include <stdio.h>
#include <string.h>

#define BUFSIZE 128
#define TAG 0
```

```
int main(int argc, char *argv[])
{
        char idstr[32];
        char buff[BUFSIZE];
        int numprocs;
        int myid;
        int i;
        MPI_Status stat;
        MPI_Init(&argc,&argv); /* all MPI programs start with
                                  MPI_Init; all 'N' processes exist
                                  thereafter */
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs); /* find out how big
                                                    the SPMD world is */
        MPI_Comm_rank(MPI_COMM_WORLD,&myid); /* and this processes'
                                                          rank is */
        /* At this point, all the programs are running equivalently,
        the rank is used to distinguish the roles of the programs in
        the SPMD model, with rank 0 often used specially... */
        if(myid == 0)
        {
        printf("%d: We have %d processors\n", myid, numprocs);
        for(i=1;i<numprocs;i++)
        {
        sprintf(buff, "Hello %d! ", i);
        MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG,MPI_COMM_WORLD);
        }
        for(i=1;i<numprocs;i++)
        {
        MPI_Recv(buff,  BUFSIZE,MPI_CHAR,  i,  TAG,MPI_COMM_WORLD,
                &stat);
        printf("%d: %s\n", myid, buff);
        }
        }
        else
        {
        /* receive from rank 0: */
        MPI Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD,
                &stat);
        sprintf(idstr, "Processor %d ", myid);
        strcat(buff, idstr);
        strcat(buff, "reporting for duty\n");
        /* send to rank 0: */
        MPI Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI COMM WORLD);
        }
        MPI Finalize(); /* MPI Programs end with MPI Finalize; this
                            is a weak synchronization point */
        return 0;
}
```

It is important to note that the runtime environment for the MPI implementation used (often called mpirun or mpiexec) spawns multiple copies of the program, with the total number of copies determining the number of process ranks in MPI_COMM_WORLD, which is an opaque descriptor for communication between the set of processes. A Single-Program-Multiple-Data (SPMD) programming model is thereby facilitated, but not required; many MPI implementations allow multiple, different, executables to be started in the same MPI job. Each process has its own rank, the total number of processes in the world, and the ability to communicate between them either with point-to-point (send/receive) communication, or by collective communication among the group. It is enough for MPI to provide an SPMD-style program with MPI_COMM_WORLD, its own rank, and the size of the world to allow for algorithms to decide what they do based on their rank. In more robust examples, I/O should be more carefully managed than in this example. MPI does not guarantee how POSIX I/O would actually work on a given system, but it commonly does work, at least from rank 0.

The notion of process and not processor is used in MPI. The copies of this program are mapped to processors by the runtime environment of MPI. In that sense, the parallel machine can map to 1 physical processor, or N where N is the total number of processors available, or something in between.

For maximal potential for parallel speedup more physical processors are used. It should also be noted that this example adjusts its behavior to the size of the world N, so it also seeks to be scalable to the size given at runtime. There is no separate compilation for each size of the concurrency, although different decisions might be taken internally depending on that absolute amount of concurrency provided to the program.

2.15 Programming Paradigms

The application users commonly used two types of programming paradigm: **SPMD** (Single Program Multiple Data) and **MPMD** (Multiple Program Multiple Data). In SPMD model, each process runs the same program in which branching statements may be used. The statements executed by various processes may be different in various segments of the program, but one executable (same program) file runs on all processes.

In MPMD programming paradigm, each process may execute different programs, depending on the rank of the processes. More than one executable (program) is needed in MPMD model. The application user writes several distinct program, which may or may not depend on the rank of the processes.

- For execution of the SPMD program, the command format used is:

```
mpirun -n <number of processes> <Executable>
```

- For execution of MPMD program, the command format used is:

```
mpirun -n <number of processes> -h<number of hosts> <Master Executable>: -n
<number of processes> -host <hosts> <Number of hosts> <Slave Executable>
```

3. Previous Work

3.1 Communication Patterns

We can measure message rate impersonating those found in typical scientific applications with the help of these communication patterns. In this benchmark, a test on the set of three different communication patterns, namely pair based communication, pre-posted receives and all-start model are verified without fault tolerance. A single direction pair-based test which is similar to the communication pattern found in most message rate benchmarks is also provided to offer comparison and validation with other existing benchmarks.

All tests share a number of features in common, including a computation/communication phase design, variable tags and the ability to send data during communication. The reported massage rate for each process includes both the sends and receives accomplished by that process.

Large scale applications can be divided into periods of computation and communication, which repeat for the life of the application. During the communication phase, a significant portion of main memory, is modified, resulting in cache misses during communication phases. Each test includes a cache invalidation step, which simulates an application working set for each computation phase.

A brief overview of the communication patterns are given below:

- **Single Direction Communication:** The single direction communication test mimics existing message rate benchmarks. Processes are paired off, with the lower rank sending message to the higher rank in a tight loop. The individual pair synchronize before communication begins to minimize jitter in measurements. Process pairs are chosen to minimize the number of pairs placed on the same node and maximize traffic across the network.

- **Pair-based Communication:** In pair based communication, each process communicates with a small number of remote processes in each communication phase. The communication is paired, so that a given process is both sending and receiving messages with exactly one other process at a time, rotating to a new process when communication is complete. A best effort is made to ensure that the remote processes a given process must communicate with are located on remote nodes, in order to more fully stress the network. This is likely a departure from our applicationcentric approach as it is likely that at least one communicating process would be on the same node in a multi-core environment.

- **Pre-posted Communication:** In order to increase the probability of expected message reception, a number of applications pre-post receives for the next communication phase before starting the computation phase. The long communication phase essentially guarantees that receive buffers will be available during the communication phase and that all messages are expected by the MPI layer.

  The pre-posted communication test simulates such a model by posting data receives from all communicating processes, invalidating the cache to simulate the computation working set and synchronizing with a barrier, followed by starting data send to all communicating processes. Although the test guarantees that all receives are posted, it also tends to push the receive queue out o cache for early receives due to the cache invalidation between posting the receives an starting the sends.

- **All-Start Communication:** The all start communication test processes many of the same properties as the pre-posted communication test, but does not guarantee that all receives are pre-posted and on validates the cache to simulate the computation working set before any communication calls in a give iteration. The test simulates an application which finishes a computation phase, then issues all communication calls at once with a single MPI_WAITALL call to complete all communication.

  Like the pre-posted communication test, the MPI is forced to deal with the large number of outstanding receives. The test will also likely cause the MPI to have to search a large portion of the expected queues for any incoming messages, as the queue is ordered by remote process. MPI implementations which optimize queue searching by maintaining per-process receive queues in addition to a global queue for handling MPI_ANY_SOURCE may be able to avoid the deep queue search.

3.2 Recent Amendments in Fault Tolerance

As we showed in section 2.3 message logging is better than coordinated check point. The model of message logging has been extensively evaluated in the past. The modifications are embedded into the model itself and drastically change the need and minimizes the overhead caused by message logging. In this section we describe the nature of the changes at the source for a new evaluation of the impact of synchronicity on event logging performance.

- **Adapted Message Logging Model for MPI Communications :** Though the previous model has been used in many implementations of message logging in the past, it is unable to capture the complexity of MPI communications. This was left unaddressed as long as the performance gap between network and memory bandwidth was hiding the ensuing overhead. But as the performance of network interface cards progressed it became clear that extra memory copies on the critical path of messages were the source of significant performance penalties.

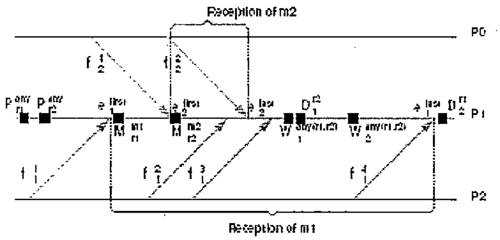


Figure 3.1

Discrepancies between the model and the reality of MPI communication basically lie in the existence of nonblocking communications. Those are intended to maximize opportunities for communication overlap by computation by allowing for the application to post its intention to communicate, compute while the communication actually takes place, and to wait for completion of the communications later. The rest of this section details the improved model used to better describe non-deterministic events with concurrent nonblocking messages.

  - **Fragments :** Every message is divided into a number of network fragments when it is transfered over the network, the number depending on its length. Though MPI enforces a FIFO semantic for messages from a particular sender, at the lowest network level there is no particular order between fragments. Consequently, as depicted in the example of figure 8, when receiving two different messages m1 and m2, the first fragment of m1 coming first does not imply that the last fragment of m1 arrives before the last fragment of m2.

Therefore, unlike in the classic model, with MPI communications the reception order of a message cannot be fully described by a single event denoting message reception, but rather depends on the relative ordering of the multiple fragments composing the messages. Although there is a very large number of such network nondeterministic events, only the order of events denoting the first and last fragments of messages are actually meaningful to the application, as described in the next paragraphs.

- **Matching** : In order to receive a message, an MPI application needs to post a reception request, using the MPI_Irecv or MPI_Recv functions. Each request contains a buffer, a source, a tag and its relative ordering to other requests, depending on the date it has been posted. When the first fragment of a message is delivered by the network, requests are considered in order by the matching logic; the first request with a matching source and tag is associated with the incoming message fragments. All upcoming fragments of this message are delivered directly into the requests reception buffer. If no request matches, the message is unexpected; it is copied into an internal buffer until it matches an upcoming posted request.

  A matching determinant is the event denoting the association between the first fragment of a message and a particular request. In the example of figure 2, Mm1 r1 is the matching determinant between the request created by the any-source non-blocking receive Pany r1 and the first fragment reception event efirst1 . Though the relative order of the fragments from the network is always non-deterministic, the FIFO by channel MPI semantic allows for most of the matching determinants to be deterministic. The only non-deterministic ones are promiscuous receptions, i.e., when a request can match a message coming from any-source. Those promiscuous matching determinants are the only events that need to be logged in order to replay a correct matching during recovery.

- **Waiting for completion of requests** : When using nonblocking communications, several requests can concurrently progress while the application is computing. When computation cannot process further without accessing buffers involved in an ongoing communication, the application waits for the completion of the corresponding requests. All the functions allowing the application to check the status of a request (like MPI_Wait) are represented by a completion test event. A delivery determinant is the event denoting the association between a particular completion test event and a message last fragment event. As an example, in figure 2, Dr2 1 is the delivery determinant associated to the last fragment reception event elast 2 and the completion test Wany(r1,r2) 1 . A special bottom event denotes that no last fragment event occurred since the last test for completion event. Again, the most common delivery determinants are always deterministic, namely the MPI_Recv, MPI_Send, MPI_Wait and MPI_Waitall functions. However, for MPI_Waitany, the outcome of the MPI call depends on the ordering between last fragment events of messages matched with the waited requests. MPI_Waitsome, MPI_Test, MPI_Testany, MPI_Testsome and MPI_Iprobe add to the previous source of non-determinism a dependency between the arrival date of the last fragments and the date of the completion test. Logging all the delivery determinant events appearing in a function where only a subset of the requests is allowed to complete is sufficient to ensure a deterministic replay of all non-deterministic deliveries.

- **Benefits from the improved model :** One of the most important optimizations for a high throughput communication library is zero copy: the ability to send and receive directly into the application's user-space buffer without intermediary memory copies. To enable it, the matching must be resolved upon arrival of the very first fragment. When it is delayed until the completion of the message, as it is necessary when using the legacy model of atomic message reception event, the actual result is that the message cannot be delivered directly into the application buffer. The MPI library has not yet associated a request with the message; every message pays the same penalty as if it were unexpected. The only software layer where the MPI matching can be delayed is the very low level interface with the network. Implementing message logging at this level has two severe limitations. First the message logging mechanism cannot easily take advantage of the optimized network drivers and second, at this level it is impossible to make a distinction between deterministic and non-deterministic delivery determinants. By interposing the event logging mechanism higher in the MPI library architecture, it is only necessary to log the communication events at the library level, and one can completely ignore the expensive events generated by the lower network layer, overall reducing by a large amount the number of events to log.

- **Active Optimistic Message Logging :** A new optimistic message logging solution, called active optimistic message logging [26], has been recently proposed to limit the drawbacks of existing optimistic message logging protocols.

Optimistic message logging has two main drawbacks. First, it is less efficient than pessimistic message logging on recovery because orphan processes may be created. In the event of a failure, a recovery protocol must be executed to detect orphan processes and these orphan processes must be rolled-back in addition to the failed processes. Second, to track dependencies between processes during failure free execution, dependency information must be piggybacked on application messages, adding overhead on communications [27].

In the standard model of optimistic message logging, determinants are buffered is the process memory and logged asynchronously. O2P is an active optimistic message logging protocol, i.e., it logs non-deterministic determinants on stable storage as soon as possible to reduce the probability that a message depends on an unlogged determinant when it is sent. Thus it reduces the risk of orphan message creation in case of failure.

To reduce the amount of data piggybacked on application messages, it has been proved that to be able to detect orphan messages only dependencies to unlogged non-deterministic determinants have to be tracked [28]. Since active optimistic message logging maximizes the probability that previous nondeterministic determinants are logged when a message is sent, it reduces the amount of data that needs to be piggybacked on application messages.

3.3 Existing Problem

The communication patterns that are described in section 3.1 are not safe in case of fault occurrence. So we introduce the technique of fault tolerance in these communication patterns for better and safe result.

CHAPTER 4

THE PROPOSED WORK

As defined in section 2.2, there are some communication patterns on the basis of which master node can communicate with the other nodes. These communication patterns are:

- Single Direction communications
- Pair-based communications
- Pre-posted communications
- All-start communications

Now in the above patterns of communication, there may be a chance of failure of node which results in the loss of data, as a result of which one has to suffer from communication as well as from computation loss.

So as to handle with the fault, here we proposed a procedure, which contains 4 steps. These steps are

- Processing
- Computation
- Detection
- Evaluation

In order to overcome the faults that may arise during the processing phase, we need to introduce fault tolerance ability in that. Hence with the use of fault tolerance we can handle faults and be able to reach to the correct final outcome. But before doing so, we also need to implement the parallel code for the same.

4.1 Parallel code for the communication patterns

I took the codes of communication patterns one by one and run it over an example to check whether or not it is working. Before that, we first need to parallelize the code of the patterns of communications.

- **Code for the Single Direction Communications**

```
for (i = 0 ; i < niters ; ++i)
{
nreqs = 0;
cache_invalidate();
synchronize();
start = timer();
if (rank < size / 2) {
for (k = 0 ; k < nmsgs ; ++k)
{

 MPI_Isend(send_buf + (nbytes * k), nbytes, MPI_CHAR, rank + (size / 2), tag,
 comm, &reqs[nreqs++]);
 }
 }
 else {
 for (k = 0 ; k < nmsgs ; ++k)
 {
 MPI_Irecv(recv_buf + (nbytes * k), nbytes, MPI_CHAR, rank - (size / 2), tag,
 comm, &reqs[nreqs++]);
 }
 }
 MPI_Waitall(nreqs, reqs,
 MPI_STATUSES_IGNORE);
 total += (timer() - start);
 }
```

- **Code for the pair based communications**

```
for (i = 0 ; i < niters ; ++i)
{
cache_invalidate();
MPI_Barrier(MPI_COMM_WORLD);
start = timer();
for (j = 0 ; j < npeers ; ++j)
{
nreqs = 0;
for (k = 0 ; k < nmsgs ; ++k)
{
offset = nbytes * (k + j * nmsgs);
MPI_Irecv(recv_buf + offset,nbytes, MPI_CHAR,recv_peers[j], tag,MPI_COMM_WORLD,
&reqs[nreqs++]);
}
```

```
        for (k = 0 ; k < nmsgs ; ++k)
        {
        offset = nbytes * (k + j * nmsgs);
        MPI_Isend(send_buf + offset, nbytes, MPI_CHAR, send_peers[npeers - j - 1], tag,
        MPI_COMM_WORLD, &reqs[nreqs++]);
        }
        MPI_Waitall(nreqs, reqs,
        MPI_STATUSES_IGNORE);
        }
        total += (timer() - start);
        }
```

- **Code for pre-posted communications**

```
start = timer();
for (j = 0 ; j < npeers ; ++j) {
for (k = 0 ; k < nmsgs ; ++k) {
offset = nbytes * (k + j * nmsgs);
MPI_Irecv(recv_buf + offsetnbytes, MPI_CHAR, recv_peers[j], tag,
MPI_COMM_WORLD, &reqs[nreqs++]);
}
}
total += (timer() - start);
for (i = 0 ; i < niters - 1 ; ++i)
{
cache_invalidate();
MPI_Barrier(MPI_COMM_WORLD);
start = timer();
for (j = 0 ; j < npeers ; ++j) {
for (k = 0 ; k < nmsgs ; ++k) {
offset = nbytes * (k + j * nmsgs);
MPI_Isend(send_buf + offsetnbytes, MPI_CHAR, send_peers[npeers - j - 1], tag,
MPI_COMM_WORLD, &reqs[nreqs++]);
}
}
MPI_Waitall(nreqs, reqs,
MPI_STATUSES_IGNORE);
nreqs = 0;
for (j = 0 ; j < npeers ; ++j) {
for (k = 0 ; k < nmsgs ; ++k) {
offset = nbytes * (k + j * nmsgs);
MPI_Irecv(recv_buf + offsetnbytes, MPI_CHAR, recv_peers[j], tag,
MPI_COMM_WORLD, &reqs[nreqs++]);
}
}
total += (timer() - start);
}
start = timer();
for (j = 0 ; j < npeers ; ++j)
{
for (k = 0 ; k < nmsgs ; ++k)
{
offset = nbytes * (k + j * nmsgs);
MPI_Isend(send_buf + offset, nbytes, MPI_CHAR, send_peers[npeers - j - 1], tag,
MPI_COMM_WORLD, &reqs[nreqs++]);
}
}
MPI_Waitall(nreqs, reqs, MPI_STATUSES_IGNORE);
total += (timer() - start);
```

- **All-start communications**

```
for (i = 0 ; i < niters ; ++i)
{
cache_invalidate();
MPI_Barrier(MPI_COMM_WORLD);
start = timer();
nreqs = 0;
for (j = 0 ; j < npeers ; ++j)
{
for (k = 0 ; k < nmsgs ; ++k)
{
offset = nbytes * (k + j * nmsgs);
MPI_Irecv(recv_buf + offset, nbytes, MPI_CHAR, recv_peers[j], tag, MPI_COMM_WORLD,
&reqs[nreqs++]);
}
for (k = 0 ; k < nmsgs ; ++k)
{
offset = nbytes * (k + j * nmsgs);
MPI_Isend(send_buf + offset, nbytes, MPI_CHAR, send_peers[npeers - j - 1], tag,
MPI_COMM_WORLD, &reqs[nreqs++]);
}
}
MPI_Waitall(nreqs, reqs, MPI_STATUSES_IGNORE);
total += (timer() - start);
}
```

4.2 Fault Tolerance Technique

Fault tolerance is the property that enables a system to continue operating properly in the event of the failure of some of its component. If its operating quality decreases at all, the decrease is proportional to the severity of the failure.

Recovery from errors in fault tolerant systems can be characterized as either roll-forward or roll-back. When the system detects that it has made an error , roll-forward recovery take the system state back to some earlier, correct version. Roll-back recovery reverts the system state back to some earlier, correct version.

The basic characteristics of fault tolerance require:

- No single point of repair.
- Fault isolation to the failing component.
- Fault containment to propagation of the failure.
- Availability of reversion modes.

Here I used the technique similar to that of one discussed in section 2.3. We can use these techniques, but there are certain problems due to which I cant use them in recovering from fault. Some of the problems associated with these techniques are listed below:

- If the fault occur frequently, then we need to restore it at the earlier safe point which is of-course a hectic job to do.
- It may suffer from the communication latency.
- The performance penalty in case of fault.
- The other drawback of the co-ordinated check point are the synchronization cost before the checkpoint, the synchronized checkpoint cost and the restart cost after the fault.

4.3 MPI commands

The MPI API used for communication among the threads of execution during phases are:

1. int MPI Send(void *buf, int count, MPI Datatype datatype, int dest, int tag,MPI Comm comm)

   where,

   buf Starting address of send buer (choice).

   count Number of elements to send (nonnegative integer).

   datatype Datatype of each send buer element (handle).

   dest Rank of destination (integer).

   tag Message tag (integer).

   comm Communicator (handle).

2.
```
int MPI Recv(void *buf, int count, MPI Datatype datatype, int source,
int tag, MPI Comm comm, MPI Status *status)
```

where,

count Maximum number of elements to receive (integer).

datatype Datatype of each receive buer entry (handle).

source Rank of source (integer).

tag Message tag (integer).

comm Communicator (handle).

4.4 Approach

Here I proposed a procedure, which contains 4 steps. These steps are

- **Processing :** Sends data to the monitored system and make a record of an offset and some other relevant data.
- **Computation :** Compute the data that was sent by monitoring system. Make the record of the offset and some other relevant data and then sends back the result to the monitored system along with some checking parameters.
- **Detection:** Now the monitoring system verify the parameters with its own recored parameters, if it matches then it allow the result else it marked as faulty.
- **Evaluator :** Now if the monitored system is found faulty, then it computes the result of the data associated with that faulty processor else it ignore this phase.

4.5 Algorithm

As defined above, the technique/method that I used to recover from fault. The algorithm is divided into three phases each of which perform certain operation to find out the faulty node and then to overcome from the fault that occur.

Processing :

A master or monitoring node sends the data that needs some computation to the slave node. Before sending data to the slave node, it makes a record of an offset and some other relevant data that will needed when the slave node will send the result to it. It is necessary because it is use for comparison or matching.

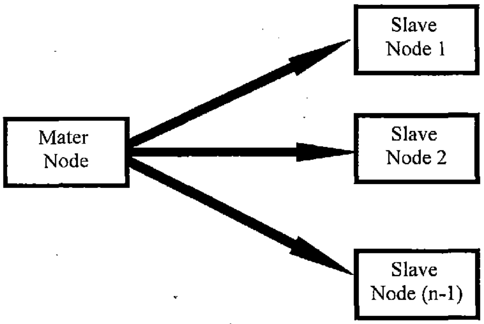


Figure 4.1

Computation :

In this phase the monitored node computes the result or perform some analysis on the data that was sent by monitoring node. It also make a record of the offset value and some other relevant data.

After the computation, it sends back the result to the monitoring node with the offset value and some other parameters that was recorded by that node.

The offset value and other relevant data is stored in a temporary memory at the monitored node. As the monitored node sends back the result to the monitoring node, the memory at monitored node flashes.

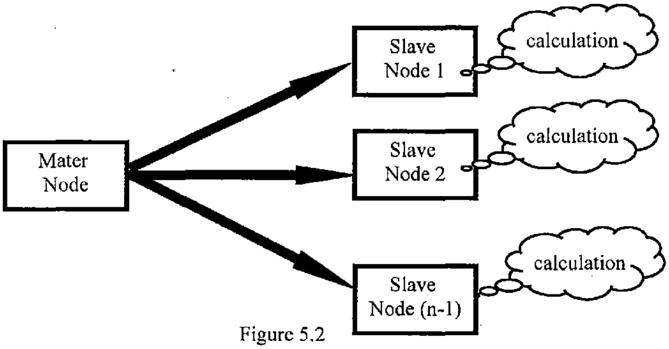


Figure 5.2

Detection :

This is an important phase among the all because in this phase, monitoring node will detect whether a monitored node is faulty or not.

What all the monitoring node do is that, during the time of receiving of data from the monitored node, it checks the offset value and other relevant data. It then verify the receiving parameters with the one it recorded previously.

If they both matches, then the monitored node is not faulty i.e. data received by monitored data is correct and can be use further. But if they dont match, then the monitored node is faulty and the data sent by the node is ignored.

Now in the case of fault detection, fourth phase will work.

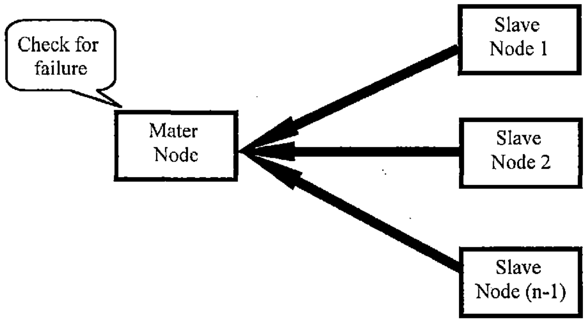


Figure 5.3

Evaluation

In this phase, if a node was found faulty in third phase, then we would not reach to the final correct outcome. So as to reach to the correct final outcome, a monitoring node computes the result of the data associated with that faulty processor.

Monitoring system takes the same offset value and other relevant data of that faulty node and computes the result.

Thus we reach to the final correct outcome.

It is assumed that the monitoring system never fails until there is some power breakdown or some external natural cause.

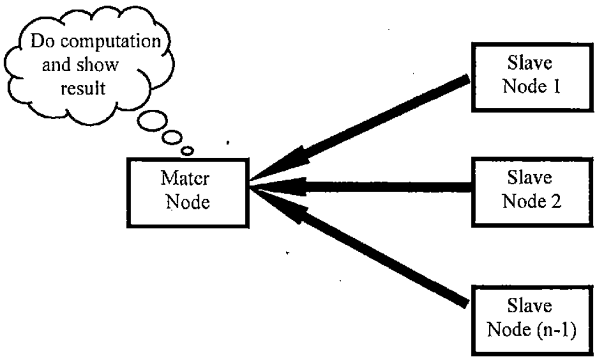


Figure 5.4

Step 1: Makes a record of the offset value and come other relevant data that has to be send for computation.

Step 2: Sends the data to the monitored node/s

Step 3: Monitored node/s receive the data and also makes a record of the offset value and other parameters.

Step 4: Monitored node/s computes the result of the data that was sent by monitoring node.

Step 5: After computation, monitored node/s sends back the result to the monitoring node with the offset value and some other relevant data and flashes the memory that was used during the time of record.

Step 6: Monitoring node receives the result from monitored node/s with the offset value and some other relevant data.

Step 7: It then verifies the offset value and other parameters with its own recorded one's.

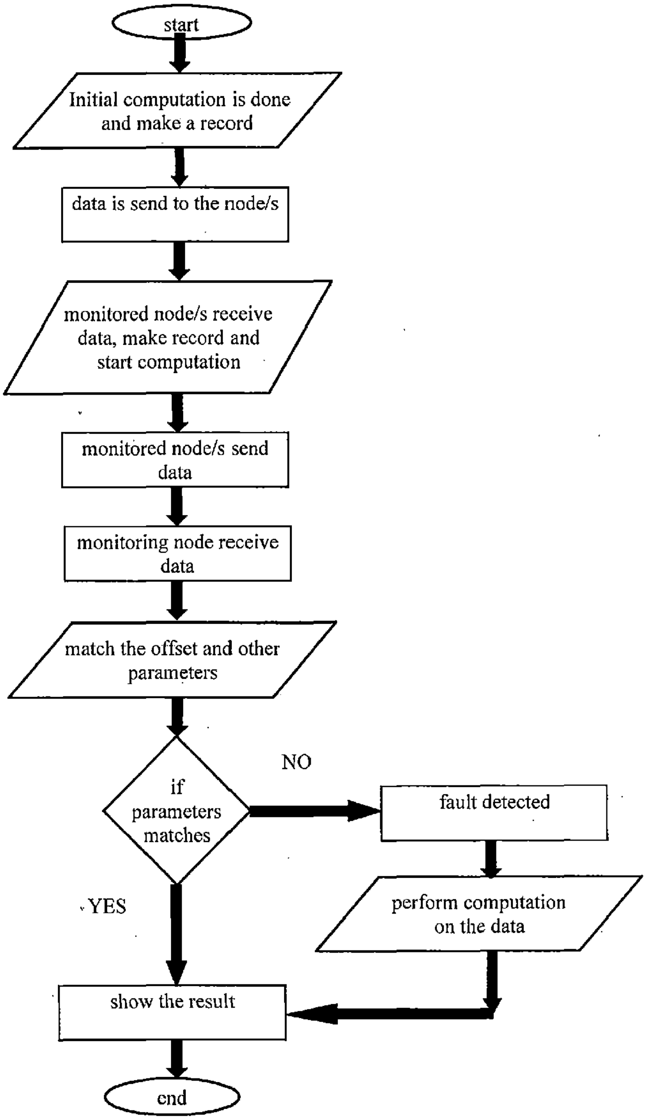
Step 8: If the data matches, then OK and END

else it declares the faulty node.

Step 9: If the node was faulty, then monitoring processor computes the result of the data associated with that faulty processor.

Step 10: Now the result is OK and the final outcome or result will be declare by the monitoring node.

4.6 Flow chart

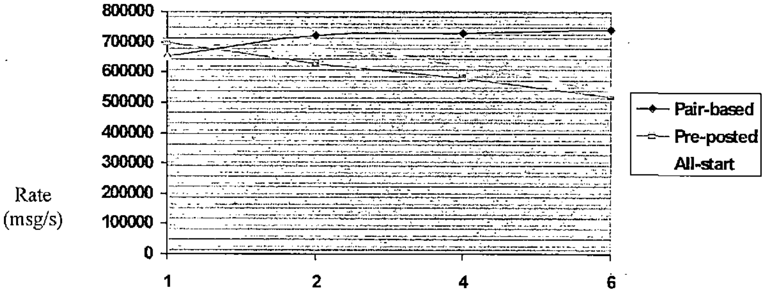

start
Initial computation is done and make a record
data is send to the node/s
monitored node/s receive data, make record and start computation
monitored node/s send data
monitoring node receive data
match the offset and other parameters
if parameters matches
NO
fault detected
perform computation on the data
YES
show the result
end

CHAPTER 5

RESULT ANALYSIS

5.1 Peer Count Analysis

The results for pair-based, pre-posted, and all-start communication patterns presented, assume a process is communicating with six other peers. In this section, the impact of varying the number of peers utilized in communication is examined. Two explanations for the performance behavior are the longer message queues of the pre-posted and all-start tests and a limitation in the network stack when receives are posted for messages from multiple peers. The performance of the pairbased test does not eliminate either hypothesis, as the process only posts receives from a single peer at a time, which also results in a significantly shorter receive queue.



Number of peers

Figure 5.1

5.2 Process per Node Scaling

The results present the impact of running multiple processes on a node. Rather than raw message rates, the results are presented as scalability based on total node message rate divided by one process per node message rate. The total number of processes in the experiment varies based on the number of process per node. The communication pattern between nodes is setup such that two processes on the same node will not communicate.

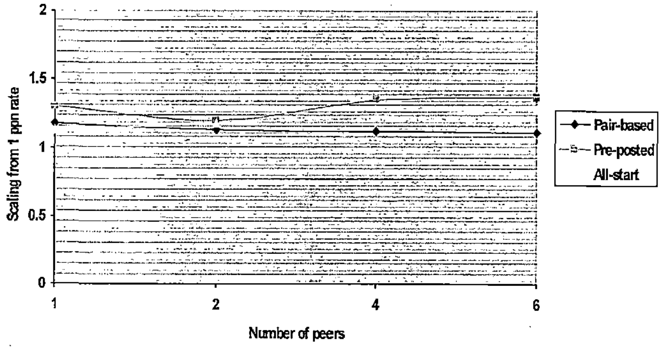


Figure 5.2

CHAPTER 6

CONCLUSION

Network design and platform procurement is, in part, driven by performance. Therefore it is critical that it accurately reflects the performance characteristics of real applications.

Here we test the message rate under fault tolerance scenario likely to be encountered by real applications by simulating an application working set and both sending and receiving data with multiple nodes.

The impact of the working set size, number of processes and the number of processes per node will affect the result under the conditions.

In peer count analysis, pair-based communication shows the best result.

In the number of peer per node analysis, we can see that all have almost equal starting and ending values. But they differ in their values in between points due to which pre-posted communication shows the best result .

CHAPTER 1

INTRODUCTION

1.1 HPC – High Performance Computing

High-performance computing (HPC) uses super computers and computer clusters to solve advanced computation problems. High Performance Technical Computing (HPTC), generally refers to the engineering applications which is used in cluster based computing. HPC can also be applied to business uses of cluster based super computers such as data warehouse, transaction processing.

HPC can also be used as a synonym for supercomputing, but in other scenarios super computer is used to refer to a more powerful set of high performance computers.

As machine sizes are growing day by day, tends to thousand of nodes, so it is important that an application must utilize the machine resources effectively. So it is important to know how an application can utilize machine resources.

There are various measures on which we can find out the efficiency and effectiveness of cluster, but the three most commonly measured metrics are bandwidth, latency rate and message rate. However other parameters such as independent progress, host overhead, etc can also impact application performance. The difficult question is: how should the data be measured given that interconnect performance can vary dramatically based on the operating conditions of the application of the application using it? As an example, the average length of the MPI message queues will impact both the latency rate and message rate[2] that the network can deliver.

Of the three main measures of interconnect performance, message rate seems to be of measure. Message rate is the measure of how many distinct messages a node can send and/or receive in a given time period, and is often referred to as message throughput. For example a massage rate of 1 million messages per second would only be able to sustain a bandwidth of 8 MB/s for messages of size 8 bytes and a bandwidth of 1GB/s for messages of size 1 KB. Thus the message rate determines the minimum message size which can saturate the bandwidth of a given network.

REFERENCES

[1]. K. D. Underwood and R. Brightwell, "The impact of MPI queue usage on message latency," in Proceedings of the International Conference on Parallel Processing (ICPP), Montreal, Canada, August 2004.

[2]. L. Dickman, G. Lindahl, D. Olson, J. Rubin, and J. Broughton, "Path- Scale InfiniPath: A first look," in Proceedings of the 13th Symposium on High Performance Interconnects (HOTI'05), August 2005.

[3]. The MPI Forum, "MPI: a message passing interface," in Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing. New York, NY, USA: ACM Press, 1993, pp. 878–883.

[4]. P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, and F. Cappello, "Improved message logging versus improved coordinated checkpointing for fault tolerant MPI," in IEEE International Conference on Cluster Computing (Cluster 2004). IEEE CS Press, 2004.

[5]. S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The LAM/MPI checkpoint/restart framework: System-initiated checkpointing," in Proceedings, LACSI Symposium, Sante Fe, New Mexico, USA, October 2003.

[6]. A. Bouteiller, G. Bosilca, and J. Dongarra, "Redesigning the message logging model for high performance," in International Supercomputer Conference (ISC 2008), Dresden, Germany, June 2008.

[8]. D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a realistic model of parallel computation," in Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1993, pp. 1–12.

[9]. R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson, "Effects of

communication latency, overhead, and bandwidth in a cluster architecture," in Proceedings of the 24th Annual International Symposium on Computer Architecture, June 1997.

[10]. A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Sheiman, "LogGP: Incorporating long messages into the LogP model," Journal of Parallel and Distributed Computing, vol. 44, no. 1, pp. 71–79, 1997.

[11]. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick, "An evaluation of current highperformance networks," in 17th International Parallel and Distributed Processing Symposium (IPDPS'03), Apr. 2003.

[12]. Q. O. Snell, A. Mikler, and J. L. Gustafson, "NetPIPE: A network protocol independent performance evaluator," in Proceedings of the IASTED International Conference on Intelligent Information Management and Systems, June 1996.

[13]. Netperf, http://www.netperf.org.

[14]. Network-Based Computing Laboratory Benchmarks, http://mvapich.cse.ohio-state.edu/benchmarks/.

[15]. Intel MPI Benchmarks, http://software.intel.com/en-us/articles/intel-mpibenchmarks/.

[16].W. Lawry, C. Wilson, A. B. Maccabe, and R. Brightwell, "COMB: A portable benchmark suite for assessing MPI overlap," in IEEE International Conference on Cluster Computing, September 2002, poster paper.

[17]. J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. K. Panda, "Performance comparison of MPI implementations over InfiniBand, Myrinet and Quadrics," in The International Conference for High Performance Computing and Communications (SC2003), November 2003.

[18]. K. Underwood, "Challenges and issues in benchmarking MPI," in Recent Advances in

Parallel Virtual Machine and Message Passing Interface: 13th European PVM/MPI Users' Group Meeting, Bonn, Germany, September 2006 Proceedings, ser. Lecture Notes in Computer Science, B. Mohr, J. L. Traff, J. Worringen, and J. Dongarra, Eds., vol. 4192. Springer-Verlag, 2006, pp. 339–346.

[19]. E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 2004, pp. 97–104.

[20] .W. Huang, G. Santhanaraman, H.-W. Jin, Q. Gao, and D. K. Panda, "Design and Implementation of High Performance MVAPICH2: MPI2 over InfiniBand," in International Sympsoium on Cluster Computing and the Grid (CCGrid), Singapore, May 2006.

[25].http://caml.inria.fr/pub/ml-archives/camllist/2003/07/155910c4eeb09e684f02ea4ae342873b.en..html

[26]. T. Ropars and C. Morin, "O2P: An Extremely Optimistic Message Logging Protocol," INRIA Research Report 6357, November 2007.

[27]. M. Schulz, G. Bronevetsky, and B. R. Supinski, "On the Performance of Transparent MPI Piggyback Messages," in Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 194–201.

[28]. M. Schulz, G. Bronevetsky, and B. R. Supinski, "On the Performance of Transparent MPI Piggyback Messages," in Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 194–201.