AN AUTOMATIC LOGFILE ANALYZER FOR PARALLEL PROGRAMS

AN AUTOMATIC LOGFILE ANALYZER FOR PARALLEL PROGRAMS

By

Haitong Zhang, M.Eng.

A Thesis

Submitted to the School of Graduate Studies

in partial fulfillment of the requirements

for the degree of

Master of Science

McMaster University

MASTER OF SCIENCE (1999) McMASTER UNIVERSITY
(Computer Science)

TITLE: An Automatic Logfile Analyzer For Parallel Programs

AUTHOR: Haitong Zhang, M. Eng.
(Beijing University of Aeronautic and Astronautics)

SUPERVISOR: Dr. Sanzheng Qiao

NUMBER OF PAGES: xi, 86

Abstract

Detecting communication errors in parallel programs has remained a very challenging research and application area. In the thesis, we will present a new approach to detect communication errors in parallel programs. We also implement a complete tool to achieve this goal. The Logfile Analyzer is a tool for automatically analyzing the logfiles generated during a parallel program execution. The purpose of the tool is to check the communication consistency of parallel programs using MPI. It can help the programmer to detect the communication errors, improve the parallel program reliability, and give the programmer an overall picture of the communication sequence. Along with the Analyzer, the tool also provides the Logger, the Wrapper and the Preprocessor, enabling the automatic generation of the logfiles from MPI programs. The logging procedure is hidden from the user. The logfiles contain all information needed for analysis. Currently, the tool supports all major MPI functions and can be run on UNIX and Windows systems. The tool is implemented in C++. It is designed to be extensible and reusable.

In the thesis, we will discuss the design idea and implementation details. We also provide a substantial number of testing cases to prove tool utility.

Acknowledgements

I would like to express my sincere appreciation for Professor Sanzhang Qiao for guiding me in the work that led to this thesis. His continuous assistance and constructive remarks have been great help to me.

I also want to express my gratitude to Professor Skip Poehlman and Ridha Khedri for serving on my thesis committee.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Parallel programming using message passing is error prone. There are various errors that may occur in parallel programming. It is difficult or impossible to avoid all of the communication errors at the design stage. Among these errors, the ones related to communications are very difficult to detect. Some typical communication errors are deadlocks, disordered messages, unmatched communication pairs, message with incorrect parameters and pending messages. These errors can not be detected by current compilers. Some potential errors may not happen in every execution, even though they happen in some executions, but trying to locate them is very difficult and time consuming.

Why is debugging parallel programs so difficult? First, unlike sequential programs, parallel programs run on several processors concurrently and the processors communicate with each other by passing messages. The communication sequence of each program execution is indeterminate. Secondly, most message passing libraries,

such as MPI (Message Passing Interface) do not provide functionality for tracing program execution. For example, we can use "printf" to trace a sequential program by simply adding some debugging outputs. MPI does not provide parallel I/O operations. The programmer cannot trace a program execution on remote processes. If some errors occur on the remote process, such as program hang, failed or getting incorrect result, it is very difficult to find out the error source, because the programmer has no idea about what happens on the other processes. Even if MPI provided parallel I/O, there would be additional problems since parallel I/O requires communications which could be the source of more problems.

All these things make writing reliable parallel programs a very difficult task. How to detect the communication errors, or in other words, how to check the communication consistency in parallel programs, and thus improve the reliability of parallel programs has remained a challenging research area.

1.2 Contributions

In this thesis, we present a new approach to checking communication consistency in parallel programs using MPI. It allows us to improve program reliability, detect errors in communication, and also obtain an overall picture of the execution of a program.

The tool consists of three essential components and one add-on component. Each component is designed and implemented independently. They are Logger, Wrapper, Analyzer plus Pre-preprocessor. The Analyzer plays key role in the tool.

The tool only focuses on communication part of the parallel program and assumes the sequential part is correct. Our tool supports most of the major MPI communication functions. Currently, it handles two kinds of communication: point-to-point and collective. The point-to-point communication functions include blocking send, blocking receive, non-blocking send, non-blocking receive. For blocking and non-blocking send, our tool also supports different modes, such as Buffered mode, Ready mode, Standard mode and Synchronous mode. The collective communication functions include broadcast, reduce, scatter and gather. They all perform in synchronous mode. The tool also supports the non-blocking completion functions such as Waits and Tests. Although the current version of the tool does not support all MPI functions now, it is designed to be easily extended to support other communication functions.

The basic idea behind the tool is to use logfiles to record intermediate communication events which occur during an execution of a parallel program. Then we use the analyzer to process the logfiles automatically, check communication sequences and detect errors. The major work is done in three stages, execution stage and post execution stage. In the first stage, the log files are generated. In the second stage, the logfiles are analyzed, which is the major purpose of the tool. Before the execution stage, the pre-processor needs to process the program replacing the original MPI functions

with wrapped MPI functions. So, actually the whole tool works on three stages, pre execution, execution and post execution stages.

We know there are some existing profiling tools using logfiles, such as upshot [1], but they focus on different issues, such as performance tuning, rather than debugging. Although we implemented our tool on top of MPI [2], the ideas are applicable to other message-passing communication libraries, for example PVM and PARMACS [3] [4]. We use object oriented technology to design and implement the tool because it can provide good modularity and reusability. The functions in this tool can be easily extended or reused. The MPI functions can be easily replaced by other similar communication library functions.

The Automatic Logfile Analyzer is a complete system. It covers pre-processing, generating runtime library and powerful analyzing. We have finished the design and implementation of all parts. So far, the testing results are satisfactory. They successfully meet our design goal. The tool can run on both Unix and Windows. The tool has been presented on the Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications [5]. The executable files for Unix and Windows with the complete reference manual page are available now for free referencing.

1.3 Organization of the Thesis

The thesis is organized into seven chapters. In chapter 2, we introduce some basic terms and conventions used in MPI Standard. We will also give some basic

knowledge about the parallel programming and the MPI Standard. In chapter 3, we will discuss the communication types covered by our tool, their definitions, prototypes and typical errors in parallel programming. The major communication types described in this chapter include blocking point-to-point, non-blocking point-to-point, collective and non-blocking completion operations. In chapter 4, we will give detailed information about the architecture of the tool: how many components are involved, the relationship among the different components, the entire system layout and the functionality of each component. We devote a whole chapter 5 to the core part of the tool — Analyzer. We address the interface design as well as the algorithms. In chapter 6, we present the test cases. The design of the test cases includes some real applications. We present the whole procedure to show how the Automatic Logfile Analyzer can help the programmer to resolve the communication problems. So the reader can obtain some real appreciation about the usage of our tool, we also give the test results. Finally, we will draw conclusions and discuss future work in chapter 7.

Chapter 2

Message Passing Interface

Before we discuss our tool, we need to introduce Message Passing Interface (MPI) [2], and some terms and data types used in MPI. We will use these data types extensively in this thesis. Understanding these terms and data types is very important for developing our tool.

2.1 Introduction

MPI stands for message passing interface. It was developed by an open, international forum consisting of representatives from industry, academia, and government laboratories. MPI is not a new programming language. It is a standard specification for a message passing library of subprograms which can be called from C and Fortran 77 programs. MPI has been carefully designed to permit maximum performance on a variety of systems, and it is based on message passing, one of the most powerful and widely used paradigms for programming in parallel systems.

There are two major parallel computer models. One is called the shared-memory parallel computer [6] where such a computer has a global memory that can be accessed by all the processors. The cost of building a shared-memory parallel computer increases rapidly with the number of processors. The other model is called the distributed-memory parallel computer [7]. Such a computer connects several processors together, each with its own local memory. For example, some distributed-memory computers are simply a collection of workstations or personal computers linked together by an electric network (e.g. Ethernet). This kind of parallel system is inexpensive to build and offers great flexibility in terms of computational resources. The more powerful distributed-memory computers consist of a fixed number of processors, all of the same architecture and connected together by a fixed communication network.

Most programming languages and environments shield the programmer from the intricacy of working directly with processors. They supply instead, a higher level concept, called a process. Usually only one process runs on one processor.

Shared-memory and distributed-memory parallel computers have resulted in two major parallel programming models. One is High Performance Fortran (HPF) [8], which is based on shared-memory data-parallel model with implicit parallelism. The other major parallel programming model is a distributed model with explicit control parallelism, also referred to as a message passing programming model [9]. Processes in a message passing programming model are only able to read and write into their respective local memory. They synchronize with one another by explicitly calling library procedures. The message passing programming model is regarded as a viable

means of programming both on multicomputers and heterogeneous networks of workstations. That is a definite advantage when developing applications that will be ported onto other parallel mulitcomputers. Applications from many of the quantitative disciplines including physics, engineering, earth science, climate and even urban traffic simulation [10] [11] have been developed under the message passing model, making use of the MPI standard.

The scope of the MPI standard has been deliberately limited to the message passing programming model. There is no global address space in the message passing model. Since the MPI standard is not a complete parallel programming environment, issues such as parallel I/O, parallel program composition, and debugging are not addressed by MPI.

2.2 Terms and Conventions in MPI

In this section, we introduce some MPI terms and conventions related to our tool.

2.2.1 Processes and Process Groups

An MPI program consists of certain number of processes, executing their own code, in an MIMD (multiple instruction multiple data) style. The code executed by processes need not be identical. The processes communicate via calls to MPI communication library. The parallel programs which our tool handles, assume that only MPI calls are used for communication routines, no other communication library is

involved. The current MPI version does not support dynamic creation or deletion of processes during program execution, the total number of processes is fixed and given by the user when the program starts to run.

The processes which participate in the program execution can form different groups. A process group is an ordered collection of processes, and each process is uniquely identified by its rank within the ordering. For a group of n processes the ranks run from 0 to n-1. One of the most important ways in which the process groups can be used is to specify which processes are involved in a collective communication operation, such as broadcast. If the program does not specifically define the process groups, there is only one group including all of the initial processes. Currently our tool only supports the default process groups called communication world which includes all of the initial processes.

2.2.2 Messages and Message Selectivity

A message consists of a piece of information, called its body, together with some additional data called an envelope. The envelope of a message specifies which recipient process can read the message.

The message envelope consists of a fixed number of fields; they are source, destination, tag and communicator. The destination is determined by the rank of the receiver process. The tag is the unique integer specified by the programmer. It is used by the program to distinguish different types of messages. The range of the tag is implementation dependable. The communicator is an opaque object, it defines the

communication context and process group. A message sent within a context must be received in the same context. The communicator is always initialized to be communication world.

The message selectivity of MPI point-to-point communication is explicitly based on source/destination process, message tag, and communicator. The message source is determined by the identity of the message sender. The other fields are specified by arguments in the send operation. The message destination is specified by the dest argument.

2.2.3 Types of MPI Calls

Basically, all MPI communication calls can be classified into the following five types.

- **non-blocking** If the procedure may return before the operation completes, and before the user is allowed to re-use the resource (such as buffers) specified in the call. For example, the send function MPI_Isend is a non-blocking call. The non-blocking function returns immediately, before the communication is established. It only serves to initiate the communication, not to complete the communication. The non-blocking function returns no matter whether the communication succeeds or fails.
- **blocking** If the return from the procedure indicates the user is allowed to re-use resources specified in the call. For example, the synchronous send (MPI_Ssend) is a blocking call. The function returns after the communication completes. That means the

message has been successfully copied from sender's buffer to the receiver's buffer. The blocking function may cause deadlock if the communication fails.

- **local** If the completion of the procedure depends only on the local executing process. Such an operation does not require communication with another user process. For example, the non-blocking communication completion function call MPI_Wait, completes a specified non-blocking communication started on the local process.
- **non-local** If the completion of the operation may require the execution of some MPI procedures on another process. Such an operation may require communication with another user process. For example, the blocking send function MPI_Send, is a non-local function call, because it cannot complete until the destination process has started to receive the message.
- **collective** If all processes in a process group need to invoke the same procedure. For example, the broadcasting – MPI_Bcast, the function can complete only after all of the participating processes call the same function.

2.2.4 Opaque Objects

MPI manages system memory which is used to buffer messages and to store internal representations of various MPI objects such as groups, communicators, datatypes, etc. This memory is not directly accessible to the user, and the objects stored there are opaque: their sizes and shapes are not visible to the user. Opaque objects are accessed via handles, which exist in user space. MPI procedures that operate on opaque

objects are passed handle arguments to access these objects. In addition to their use by MPI calls for object access, handles can participate in assignments and comparisons. A different handle type is defined for each category of objects. The Request type which our tool uses has type INTEGER.

Opaque objects are allocated and deallocated by calls that are specific to each object type. The calls accept a handle argument of matching type. In an allocate call, this is an argument that returns a valid reference to the object. For example,

```
MPI_Isend ( void* buf, int count, MPI_Datatype datatype, int dest,
                int tag, MPI_Comm comm, MPI_Request *request )
```

the output argument "request" allocates a handle to the request object and returns the handle. In a call to deallocate this handle is an argument which returns a "null handle" value. For example,

```
MPI_Wait ( MPI_Request *request, MPI_Status *status )
```

takes the "request" handle as an input and returns it as a "null handle" value. MPI provides a "null handle" constant for each object type. The value of "null handle" for "request" type is 0. Comparisons to this constant can be used to test for validity of the handle. After the handle is deallocated, the object is not accessible to the user after the call.

An opaque object and its handle are significant only at the process where the object was created, and cannot be transferred to another process. For example, the function call MPI_Isend allocates a send request handle which assumes to be 1120 at process 1, any other active handles allocated at the same process must be different from

1120. However, it is not necessary to be different from the active handles allocated at the other processes. For example at process 2 or 3 there may have handles with the same value 1120. The number which is assigned to the handle can be reissued after the previous handle becomes inactive.

2.2.5 Array Arguments

Some MPI calls may need an argument that is an array of opaque objects, or an array of handles. The array-of-handles is a regular array with entries that are handles to objects of the same type in consecutive locations in the array. Whenever such an array is used, an additional len argument is required to indicate the number of valid entries (unless this number can be derived otherwise). The valid entries are at the beginning of the array; len indicates how many of them there are, and need not be the entire size of the array. For example in the function call

```
MPI_Waitall ( int count, MPI_Request *array_of_request, MPI_Status
                    *array_of_statuses )
```

the second input argument is an array of requests, the first one is the count, which is the length of the array.

2.2.6 Named Constants

MPI procedures sometimes assign a special meaning to a special value of a basic type argument. For example, tag is an integer-valued argument of point-to-point

communication operations, with a special wild-card value, MPI_ANY_TAG. Such arguments will have a range of regular values, which is a proper subrange of the range of values of the corresponding basic type; special values (such as MPI_ANY_TAG) will be outside the regular range.

MPI also provides predefined named constant handles, such as MPI_COMM_WORLD which is a handle to an object that represents all processes available at start-up time and allowed to communicate with any of them. All named constants can be used in initialization expressions or assignments. These constants do not change values during execution. Opaque objects accessed by constant handles are defined and do not change value between MPI initialization (MPI_INIT) and MPI completion (MPI_FINALIZE). The following table lists all the constants used in our tool.

NAME	*VLAUE*
MPI_PROC_NULL	0
MPI_ANY_TAG	-1
MPI_ANY_SOURCE	-2
MPI_ANY_UNDEFINED	-32766
MPI_UNDEFINED_RANK	-32766
MPI_INIT	6
MPI_FINALIZE	7
MPI_COMM_WORLD	91

Table 1. Named Constants

Chapter 3

The Communications Covered by Our Tool

As we mentioned in the chapter 1, our tool can support two major kinds of communications: point-to-point and collective. The basic point-to-point communication operations are send and receive. They can be blocking or non-blocking operations. In order to improve the program efficiency and flexibility, MPI also provides different communication modes for send operations. The collective communications include Broadcast, Reduce, Gather and Scatter. They only perform in a synchronous way. There are many issues about point-to-point and collective communications, we only address those related to our tool.

3.1 Blocking Point-to-Point Communications

The blocking point-to-point communications have blocking send and receive; these functions are the most often used when writing parallel programs. The function prototypes for blocking send and receive are shown in the following tables.

MPI_Send (buf, count, datatype, dest, tag, comm)		
buf	Input	Initial address of send buffer
count	Input	Number of elements in send buffer (integer)
datatype	Input	Datatype of each send buffer element (handle)
dest	Input	Rank of destination (integer)
tag	Input	Message tag (integer)
comm	Input	Communicator (handle)

Table 2. Prototype for Blocking Send

MPI_Recv (buf, count, datatype, sources, tag, comm, status)		
buf	Input	Initial address of receive buffer
count	Input	Number of elements in receive buffer (integer)
datatype	Input	Datatype of each receive buffer element (handle)
source	Input	Rank of source (integer)
tag	Input	Message tag (integer)
comm	Input	Communicator (handle)
status	Output	Status object (handle)

Table 3. Prototype for Blocking Receive

We give an example to illustrate how the send and receive work. Process rank 0 sends a message to process rank 1 using MPI_Send. The operation specifies a send buffer in the sender memory from which the message is taken. The location, size and type of the send buffer are specified by the first three parameters of the MPI_Send. In addition, the send operation associates an envelope with the message. This envelope specifies the message destination and contains distinguishing information that can be used by the receive operation MPI_Recv to select a particular message. The last three parameters of the send operation specify the envelope for the sent message.

Process rank 1 receives this message using the MPI_Recv. The message to be received is selected according to the value of its envelope, and the message data is stored into the receive buffer. This is the same as the MPI_Send, where the first three parameters in MPI_Recv specify the location, size and type of the receive buffer. The next three parameters are used for selecting the incoming message. The last parameter is used to return information on the message just received.

Because they are blocking operations, the MPI_Send cannot return until the message has been successfully copied from the user buffer into the system buffer if the system provides, or into receive buffer if the system does not provide, a buffer. The MPI_Recv cannot return until the sending message has been successfully copied into the user's buffer.

3.2 Non-blocking Point-to-Point Communications

Since the blocking operations block the execution when they are communicating, the performance can be improved by overlapping computation and communication as long as the computation does not modify the buffer specified by the communications. So, MPI provides non-blocking operations. The prototypes for non-blocking send and receive are almost the same as blocking ones, except they import a new parameter "request". We list non-blocking send and receive in the following tables.

MPI_Isend (buf, count, datatype, dest, tag, comm, request)		
buf	Input	Initial address of send buffer
count	Input	Number of elements in send buffer (integer)

datatype	Input	Datatype of each send buffer element (handle)
dest	Input	Rank of destination (integer)
tag	Input	Message tag (integer)
comm	Input	Communicator (handle)
request	Output	Communication request (handle)

Table 4. Prototype for Non-blocking Send

MPI_Recv (buf, count, datatype, sources, tag, comm, request)		
buf	Input	Initial address of receive buffer
count	Input	Number of elements in receive buffer (integer)
datatype	Input	Datatype of each receive buffer element (handle)
source	Input	Rank of source (integer)
tag	Input	Message tag (integer)
comm	Input	Communicator (handle)
request	Output	Communication request (handle)

Table 5. Prototype for Non-blocking Receive

The non-blocking send MPI_Isend or receive MPI_Irecv initiates the send or receive operation, but does not complete them, so, they also can be called the non-blocking start calls. A non-blocking start operation will return before the message is copied out of the send buffer. At the same time, it requires a communication request object and associates it with the request handle (the request argument). The request can be used later to query the status of the communication or wait for its completion. We will discuss this in the section 3.5.

3.3 Communication Modes

In section 3.2, we mentioned that the blocking send does not return until the message data and envelope have been safely copied out of the user send buffer, so that the sender can be free to modify the send buffer. There are two places where the message could go: one is directly moving to the receive buffer; the other place is moving into a temporary system buffer. If the system can provide a large enough buffer to store the message before they can be copied into the receiver's buffer, this will improve the program performance because the blocking send can return without waiting for the matching receive to be initiated. On the other hand, message buffering can be very expensive, as some systems cannot offer large buffer space. Considering the effect of system buffering, the MPI offers different communication modes that allow the user to control the choice of communication protocols. In the following table, we provide brief information about each mode.

Mode	System Buffer Availability	Function calls
Standard	System buffer is not guaranteed	MPI_Send, MPI_Recv, MPI_Isend, MPI_Irecv
Buffered	User needs to allocate the system buffer before and release the buffer after the send	MPI_Bsend, MPI_Ibsend
Ready	No system buffer needed, receive must be started before matching send	MPI_Rsend, MPI_Irsend
Synchronous	No system buffer available	MPI_Ssend, MPI_Issend

Table 6. Communication Mode

3.4 Non-blocking Completion Functions

The non-blocking completion operations are used to complete the non-blocking operations, such as non-blocking send and receive. The completion of send operation indicates that the sender is now free to modify the send buffer. The completion of a receive operation indicates that the receive buffer contains the received message, the receiver is now free to access it. The completion of a synchronous send indicates the message has been received by the receiver. A non-blocking communication without a proper completion may become a pending message. The pending message is a potential error which could cause unpredictable errors.

The function Wait and Test are completion functions. They belong to blocking and non-blocking operation, respectively. According to the number of non-blocking operations they can complete, they also can be divided into two classes: the single completion which only completes one non-blocking operation and the multiple completion which can complete several non-blocking operations. In tables 7 and 8, we list their prototypes.

MPI_Wait (MPI_Request *request, MPI_Status *status)		
Request	Input and Output	Requst object handle
Status	Output	Status object (handle)
MPI_Test (MPI_Request *request, MPI_Status *status)		
Request	Input and Output	Request handle (integer)
Status	Output	Status object (handle)

Table 7. Single Completion

MPI_Waitany (count, array_of_request, index, status) MPI_Waitsome (incount, array_of_request, outcount, array_of_indices, array_of_statuses) MPI_Waitall (count, array_of_requests, array_of_statuses) MPI_Testany (count, array_of_requests, index, flag, status) MPI_Testsome (incount, array_of_request, outcount, array_of_indices, array_of_statuses) MPI_Testall (count, array_of_requests, flag, array_of_statuses)		
Count	Input	Lists length (integer)
Array_of_requests	Input and Output	Array of requests (array of handles)
Index	Output	Index of handle for operation that completed (integer)
Status	Output	Status object (Status)
Incount	Input	Length of array_of_requests (integer)
Outcount	Output	Number of completed requests (integer)
Array_of_indices	Output	Array of indices of operations that completed (array of integer)
Array_of_status	Output	Array of status objects for operations that completed (array of status)
Flag	Output	Logical, return true if all requests have been set to null. Else return false

Table 8. Multiple Completion

3.5 Collective Communications

Collective communications are initiated by all the processes in the communicator within which the communication takes place. The MPI collective communication procedures are blocking. The function will not return until all the processes in the communicator have invoked the same collective function. The major

types of collective functions include barrier, broadcast, reduce, gather and scatter. We give the function definitions and describe how they operate as below.

• Barrier

A barrier call has no effect on the local memory of processes. Its sole purpose is to return after having been called by all processes associated within the same communicator.

• Broadcast

A broadcast sends the same message from one process, called the root, to all other processes within the same communicator. The following figure illustrates the broadcast.

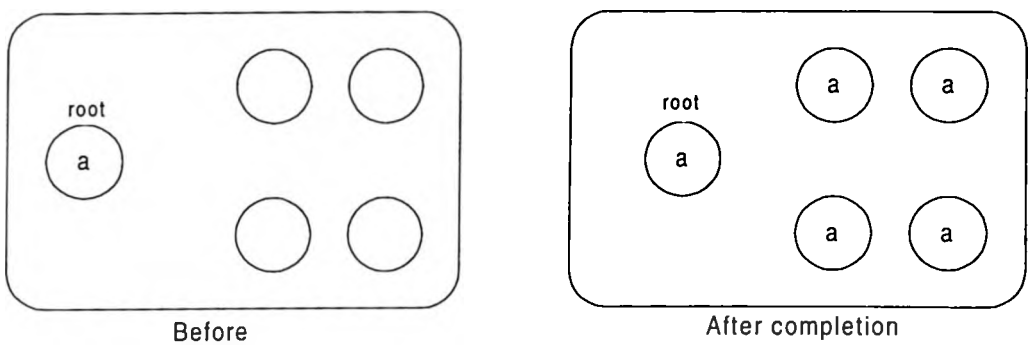

Figure 1. Broadcast

• Scatter and Gather

The scatter and gather operations are inverses of each other. In a scatter operation, a root process sends different messages of the same MPI data type to each of the other processes within the same communicator. In a gather operation, the root process

receives and concatenates different messages of the same MPI data type from all processes within the same communicator. MPI also provides a function called allgather. The only difference between gather and allgather is that, besides the root, all the other processes in the communicator can receive the same message as the root. The following figure illustrates these operations.

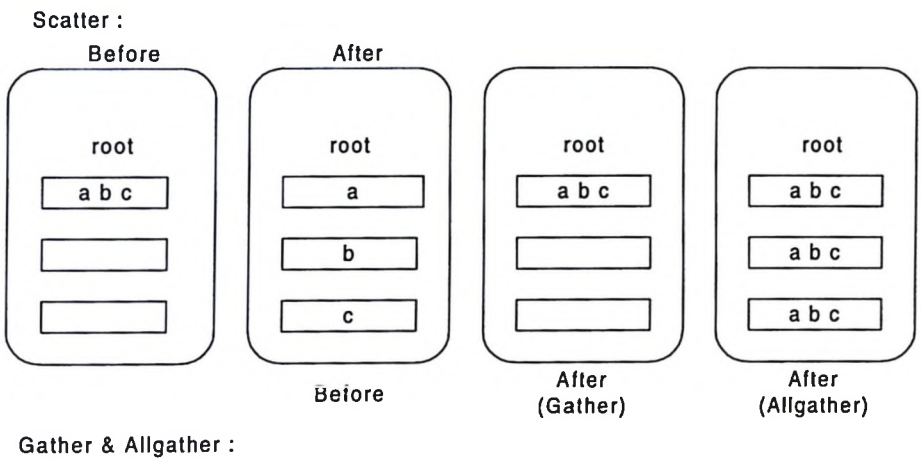

Figure 2. Scatter and Gather (Allgather)

• Reduce and Allreduce

All the elements in the same position in the input sequence of every process within the same communicator are reduced together. The result sequence is stored in the output sequence of the root process (reduce) or stored in the output sequence of every process within the communicator (allreduce). The following figure illustrates these two functions where the reduce operation is sum. Reduce and allreduce operations include sum, max, min, and, or and etc.

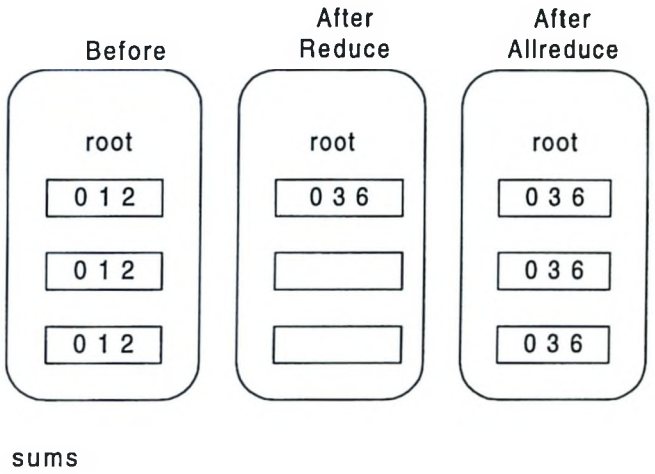

Figure 3. Reduce and Allreduce

3.6 The Typical Errors in Parallel Programming

As we discussed in the introduction, parallel programming is prone to errors. Some of the communication errors happen dynamically and cannot be detected by regular compilers. In this section, we present three major types of programming mistakes and show what kind of potential errors could happen during run time. In the chapter of Case Study, we will discuss these errors in more detail.

A. Disordered Communication Sequences

The programmer writes all the sends and corresponding receives in the program. If the communication is synchronized, the order of sends and receives are very important for parallel programs. If the order is not proper, deadlock could occur. The following table shows a simple example. This sequence may cause deadlock because the dependency

between the sends and receives. P0 will not return until P1 receives the message from P0. Meanwhile P1 will not return until P0 receives the message from P1. Since they use blocking operations, deadlock could happen.

P0	P1
Blocking send m1 to P1	Blocking send m2 to P0
Blocking receive m2 from P1	Blocking receive m1 from P0

Table 9. Disordered Communication Sequence

B. Unmatched Communication Events

The programmer writes more sends than receives, or more receives than sends, or some sends or receives use wrong parameters. So, some sends/receives cannot find matched receives/sends. If these sends or receives are blocking operations, deadlock could happen. If these sends or receives are non-blocking operations, message pending could happen. Tables 10 and 11 show two examples of unmatched communication events. The sequence in table 10 could cause deadlock. The sequence in table 11 could cause message pending.

P0	P1
Blocking send m1 to p1	Blocking receive m1 from p0
Blocking send m2 to P1	

Table 10. A Communication Sequence
with More Sends than Receives

P0	P1
Non-locking send m1to p1	Non-blocking send m2 to p0
Non-blocking receive m3 from p1	Blocking receive m1 from P0

Table 11. A Communication with Incorrect Parameters

C. Non-blocking Function without Proper Completion

A reliable parallel program requires every non-blocking send or receive being completed properly. The non-blocking events without completion could become pending messages. Pending messages are a very serious source of potential errors; they could affect the running results or cause unpredictable system errors. We list below three programming errors causing message pending:

- non-blocking sends or receives without matched waits
- using test to complete the non-blocking send or receive
- more waits than non-blocking sends or receives

Tables 12, 13 and 14 give three communication sequences to show the detail about this type of error. In Table 12, m2 could become a pending message because there is no completion function being called on P1. In Table 13, the P0 calls test to complete the non-blocking send. Since the flag returns false, the non-blocking operation is still active and it could become a pending message. Tests do not guarantee the completion of the non-blocking operations. In Table 14, the P0 starts two waits to complete the non-blocking operations. The event which the second wait is trying to complete does not exist. Since the wait is a blocking operation, it will cause deadlock.

P0	P1
Non-blocking send m1 to P2	Non-blocking send m2 to P0
Wait send m1 complete	

Table 12. A Non-Blocking Communication without Completion

P0	P1
Non-blocking send m1 to P1	Non-blocking send m2 to P0
Test send m1 with return flag = false	Wait send m2 complete

Table 13. A Non-Blocking Communication with a Test

P0	P1
Non-blocking send m1 to P1	Non-blocking send m2 to P0
Wait send m1complete	Wait send m2 complete
Wait send m3 complete	

Table 14. A Communication Sequence with More Waits

We have discussed some typical errors in parallel programming. These errors dramatically decrease the reliability of parallel programs. Because of different MPI implementations, the system buffer sizes are variable. Hence, some programs with these errors may run successfully on some systems, but fail on others. For example, consider the communication sequence in table 9; if the system provides a large buffer size, the deadlock may not occur in some executions. But a program depending on system buffer size to solve deadlock is very unreliable. Our tool tries to detect all these errors and finally improve the program reliability.

Chapter 4

The Architecture of the Automatic Logfile Analyzer

4.1 An Overview of ALA Architecture

As mentioned in the Chapter 1, the Automatic Logfile Analyzer (ALA) logically consists of three essential parts and one add-on part. Figure 4 illustrates the relation among the four parts. In pre-processing, the original MPI functions in the user program are replaced by the wrapped MPI functions; this step is done before the compilation of the source code. Then the Wrapper calls the Logger interface to log the communication events and produces the logfiles; this step is done during the program execution. After the logfiles have been generated, the Analyzer processes the logfiles and returns the messages to the user - which is the last and most important step and is done after the program execution.

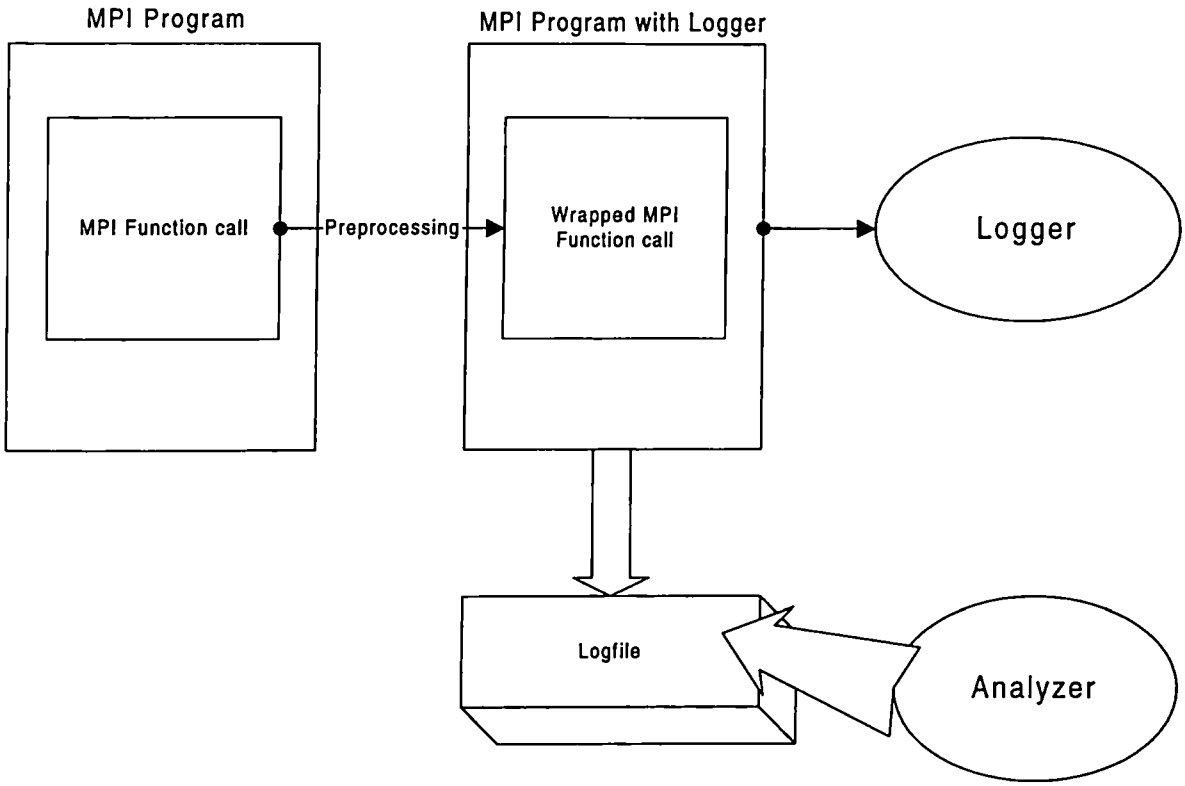

Figure 4. The System Overall Structure

4.2 The ALA Major Components

4.2.1 The Wrapper

The original MPI functions do not have the ability of tracing communication events. In order to keep track of communication events for analysis, the functionality of the original MPI functions needs to be extended. The wrapped functions are those MPI functions that encapsulate the original MPI functions as well as the Logger interface. The Wrapper is also responsible for creating and destroying the logger object during the whole logging process. We use the layering structure for the wrapper functions. They were designed to have very similar look in terms of the function prototypes as the original MPI library. They are built on top of the MPI library and the Logger. In this section, we discuss two major issues for the Wrapper.

The first issue is to determine what type of the information should be logged. Since the different communication events require different information, our selection criteria for the information structure should hold enough data for the analyzing purpose. In the current version, we have defined an information structure that contains the following fields - an event name, an event sequence ID, a source rank, a destination rank, a message tag, a communicator handle, the source line number and an array of request handles. Obviously, it is unnecessary to fill out all fields of the information structure for certain MPI functions. For example, MPI_Send needs to fill out all the fields except the array of request handles; MPI_Isend needs to fill out every field;

MPI_Bcast does not need to fill out a source_rank, a destination_rank, a message tag and array of request handles.

The second issue is to determine to which locations the logging messages should be interpolated. The criterion for the insertion positions is that we can log sufficient information for the later analysis while avoiding unnecessary log messages. We can either insert logging messages immediately before (pre-log) or after (post-log) the MPI function calls; in this way we can obtain one logging message for each function call. We also can insert logging messages both before and after the MPI function calls. This provides two logging messages for each function call. In the current version, we need to log blocking point-to-point communications, non-blocking point-to-point, collective communications and non-blocking completion functions. If the information which we log does not change after the function call, such as blocking point-to-point communication, source, destination, communicator and tag, only this information is placed into the pre-log message. If same information is only available after the function call, such as the request in non-blocking communication, the system assigns a request handle during the function call and returns this handle as an output parameter, it needs to be inserted into the post-log message. If the information is modified by the function call, such as non-blocking completion function, the request passed to the function may be set to Null after the function returns, then these data need to be inserted into both pre-log and post-log messages in order to monitor the change.

4.2.2 The Logger

The Logger records communication events which occurred during an execution and provides the interface that can be called from the wrapped MPI functions. The functionality of the Logger includes creating the log file, writing and formatting output, filling the log buffer, keeping track of communication event sequence, and controlling the frequency of flushing the log buffer to disk logfiles. During an execution, each process produces its own logfile containing communication events.

A logger object is created when the parallel program calls MPI_Init, which means the program starts to log communication events. The logger object is destroyed when the program calls MPI_Final. The user can choose how often the buffer is flushed. To minimize the loss of information due to an unexpected exit of the program, such as deadlock, the user may increase the flush rate. To increase a MPI program executing speed, the user may reduce the flush rate. The safest way to keep track of all communication events, it is necessary to flush the buffer every time a log message is generated. This feature is especially useful in detecting deadlocks. When there is no need to pay too much attention to details, the user can reduce the flush rate to improve program speed.

4.2.3 The Preprocessor

Although the wrapped MPI functions are very similar to their original counterparts, their prototypes and interpolations are different. The major difference is that most of wrapped MPI functions have the source line number information in their prototypes. Obviously, to input the source line number manually is tedious. The pre-processor translates the supported MPI functions into the wrapper functions automatically, so that the compiler can then link to the Wrapper runtime library to activate the Logger during execution. This process makes our event-logging mechanism more user friendly. Since this preprocessing hides all rules of interpolation and calling logger interface, a user may just write MPI programs as usual without worrying about how to call the wrapper functions.

4.2.4 The Analyzer

During the MPI program execution, the Logger writes all events into logfiles. After the execution, the Analyzer takes the logfiles as inputs, keeps track of the complete communication events and pending events, these are done by checking and analyzing the state vector and finally the Analyzer gives results that indicate whether the communications in the MPI programs are consistent or potential problems exist. We will explore the Analyzer in more details in the next chapter.

4.3 The Implementation Design

In order to maintain the good modularity and to improve code reusability and extensibility, we divide the whole system into three modules: MPILOG, MPIWRAP and MPICHECK. These modules can be implemented independently and can communicate with each other through clearly defined interfaces. Each module provides a set of access functions for the user which hide the implementation details. We will discuss more detail about each module in the following sections. We can easily adapt our system to a similar environment with minimum modifications to the code and effect on other parts of system.

The three modules implement the Pre-processor, the Wrapper, the Logger and the Analyzer: MPICHECK implements the Analyzer, MPILOG implements the Pre-processor and MPIWRAP implements the functionality of the Wrapper and the Logger.

In Appendix 1, we provide detailed usage of these three components.

4.4 The Implementation of ALA

The functionality of ALA is achieved by the aid of two executables plus a runtime library. For illustration purposes, we use four major MPI functions: MPI_Init, MPI_Isend, MPI_Recv, MPI_Waitsome. These examples represent most of the MPI functions in our system.

1. MPILOG

   a) This executable is responsible for interpolating the Logger into MPI source code by replacing the supported MPI functions with the wrapped MPI functions. _MPI_Init replaces MPI_Init; _MPI_Isend replaces MPI_Isend; _MPI_Recv replaces MPI_Recv; _MPI_Waitsome replaces MPI_Waitsome and so forth; all supported MPI functions are replaced with their wrapped counterparts by adding "_" before their function names.

   b) The MPI header file "mpi.h" is replaced by "mpiwrap.h".

   c) Most of the wrapped MPI functions require a source line number argument. This number is traced and put into the wrapped MPI functions automatically by the Logger.

   d) If any piece of source code appears as a MPI function supported by our tool, but it does not follow the syntax that should be used, a comment will be added after that MPI function which indicates this function cannot be logged. In most cases, these are typing errors. The user will also get an error message from the compiler. For example, the following line missed a "(" after MPI_Test:

```
MPI_Test &request100, &flag, &status); /* Line 60 can't be logged */
```

2. MPIWRAP

   This runtime library includes the Wrapper functions and the Logger interface as well as the Logger implementation. The Wrapper functions call the Logger interface

during the executions of the MPI programs and generate the logfiles. They also determine when, where and how to log communication events.

The Logger interface is provided as a set of four functions InitLog(), TermLog(), WriteLog() and WriteLogCompleted(), where

- InitLog() and TermLog() are responsible for construction and destruction of the Logger object and creating and managing logfiles.
- WriteLog() is used for filling the log buffer and formatting output to log files.
- WriteLogCompleted() is a variant of WriteLog(), and essentially performs the same functionality as the WriteLog().

The following three examples illustrate some wrapped MPI functions. Please note:

a) WriteLog() function is put before MPI function in _MPI_Recv, after MPI function in _MPI_Isend and both before and after MPI function in _MPI_Waitsome. The function type determines the logging scheme used by each MPI wrapper function. In addition, a source line argument "line" is passed to the wrapper function while there is no such argument in their original counterparts.

b) In _MPI_InitLog(), the Logger construction interface is called according to the rank of the calling process.

Pre-Log Case:

```
Int _MPI_Recv( unsigned long line,
               Void* buf,
               Int count,
               MPI_Datatype datatype,
               Int source,
               Int tag,
```

```
                MPI_Comm comm,
                MPI_Status* status)
{
     WriteLog( line, NAME_RECV, 0, comm, source,
                RANK_ROOT, tag, 0, NULL);
     MPI_Recv( buf, count, datatype, source,
                Tag, comm, status );
}

Remarks:
1) Applicable to the other blocking events
2) Source line information is passed in.
```

Post-Log Case:

```
Int _MPI_Isend(  unsigned long line,
                 Void*          buf,
                 Int            count,
                 MPI_Datatype   datatype,
                 Int            dest,
                 Int            tag,
                 MPI_Comm       comm,
                 MPI_Request*   request )
{
     MPI_Request index[1];
     MPI_Isend(buf, count, datatype, dest,
                Tag, comm, request );
     index[0]= *request;
     WriteLog( line, NAME_ISEND, 1, comm,
                RANK_ROOT, dest, tag, 1, index);
}

Remarks:
1) Applicable to most non-blocking events.
2) Source line information is passed in.
```

Composite Case:

```
Int _MPI_Waitsome(  unsigned long line,
                    Int count,
                    MPI_Request*,
                    array_of_requests,
                    Int* complete_count,
```

```
                        Int* index,
                        MPI_Status* status)
{
     WriteLog( line, NAME_WAITSOME, 0, NULL, RANK_NULL,
               RANK_NULL, NULL, Count,
               Array_of_requests);
     MPI_Waitsome( count, array_of_requests,
                   Complete_count, index, status);
     WriteLogCompleted( line, NAME_WAITSOME,
                        *complete_count, index);

}
```

Remarks:
1) Only applicable to Wait and Test events.
2) The log interface WriteLog() or WriteLogCompleted() has been called twice both before and after MPI_Waitsome.
3) Source line information is passed in.

Special Case:

```
Int _MPI_Init( int* argc_ptr, char** argv_ptr[] )
{
      int myrank;
      char filename[16];
      MPI_Init( argc_ptr, argv_ptr );
      MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
      Sprintf(filename, "p%d.log\0",myrank);
      InitLog(filename, myrank);
}
```

Remarks:
1) Only applicable to MPI_Init and MPI_Finalize events.
2) Construct the Logger object according to the rank of the calling process.
3) No source line information is passed in.

3. MPICHECK

This executable implements the Automatic Logfile Analyzer. It runs against logfiles generated from the MPI program execution. We will discuss the data structures, algorithms and implementations concerning MPICheck in Chapter 5.

Chapter 5

The Analyzer

5.1 Program Overview

In this chapter, we are going to discuss the Analyzer, the key component of the system. The Analyzer runs independently from the Logger, the Wrapper and the Pre-processor. It takes logfiles as inputs regardless of whether they are complete or partial. The latter may occur as the result of an abnormal exit during an execution. The running results from the Analyzer either indicate the communication consistency or detect the possible communication errors.

As we discussed in chapter 3, since MPI supports synchronous and asynchronous communications, deadlock and message pending would occur quite often. A parallel program may have very different communication patterns in different executions, as the result of the variance in computation capacity, computer load and network traffic, etc. This kind of randomness introduces difficulties in analyzing parallel programs. To overcome this difficulty, we use the Logger described in chapter

4 to log all communication events. Each process generates one logfile. Our analysis is based on the sequence of events in each logfile. It is unnecessary to know the chronological difference between entries from different logfiles. Therefore, the execution randomness has no effect on our analyzing results. In addition, our tool can detect not only errors, such as deadlock, which causes execution hang or fail, but also potential errors, such as incomplete operations in non-blocking communication, which may not cause any problem in one execution. In this chapter, we present the major algorithms used in the Analyzer.

5.2 The Communication Category

We have already discussed the MPI communication functions that are supported by our system in the Chapter 3. We divided these MPI functions into six categories, which are blocking send or receive, non-blocking send or receive, non-blocking completion, partial non-blocking completion, collective and persistent communications. We list all the functions according to their category in the following table.

COMMUNICATION CATEGORY	MPI FUNCTION NAME
Blocking Point-to-Point	MPI_Send, MPI_Bsend, MPI_Rsend, MPI_Ssend, MPI_Recv
Non-blocking Point-to-Point	MPI_Isend, MPI_Ibsend, MPI_Irsend, MPI_Issend, MPI_Irecv
Non-blocking Completion	MPI_Wait, MPI_Waitany, MPI_Waitsome, MPI_Waitall
Partial Non-blocking	MPI_Test, MPI_Testany, MPI_Testsome,

Completion	MPI_Testall
Collective	MPI_Bcast, MPI_Reduce, MPI_Allreduce, MPI_Gather, MPI_Allgather, MPI_Scatter
Persistent Communications	MPI_Send_Init, MPI_Bsend_Init, MPI_Rsend_Init, MPI_Ssend_Init, MPI_Start

Table 15. Communication Category

5.3 Data Structures

In this section, we address some data structures used in the Analyzer. The Log File, the Event, the State Vector, the Event Pair, the Complete Communication List, the Pending List and the Request Table are the key data structures. First, we discuss some new terminologies used in the Analyzer. We also use these terms frequently in the algorithms.

Terminology

- Event Rank

The process rank where the event is logged. For example, if an event rank equals to 0, it means that process 0 logged this event.

- Matched Pair

A matched pair that has only one send event and one receive event. The send event can be blocking or non-blocking in any modes. The receive event can be blocking or non-blocking only in standard mode. For example, MPI_Send and MPI_Irecv is a matched

pair, MPI_Isend and MPI_Irecv is a matched pair. The following table shows how event pairs are matched.

1	Matched blocking send and blocking receive	MPI_Send & MPI_Recv, MPI_Ssend & MPI_Recv, MPI_Rsend & MPI_Recv, MPI_Bsend & MPI_Recv
2	Matched blocking send and non-blocking receive	MPI_Send & MPI_Irecv, MPI_ Bsend & MPI_Irecv, MPI_Irsend & MPI_Irecv, MPI_Issend & MPI_Irecv
3	Matched non-blocking send and blocking receive	MPI_Isend & MPI_Recv, MPI_Issend & MPI_Recv, MPI_Irsend & MPI_Recv, MPI_Ibsend & MPI_Recv
4	Matched non-blocking send and non-blocking receive	MPI_Isend & MPI_Irecv, MPI_Issend & MPI_Irecv, MPI_Ibsend & MPI_Irecv, MPI_Irsend & MPI_Irecv
5	Non-blocking send or receive and Null	MPI_Isend & Null, MPI_Ibsend & Null, MPI_Irsend & Null, MPI_Issend & Null, MPI_Irecv & Null

Table 16. Event Pairs

• Matched Events

An event is matched by one or more events. There are three types of matched events. The first kind is Send and Receive matched events when their source and destination, as well as tag, match. If there are only one Send and one Receive, they are also represented as a matched pair. If the Receive uses a wild card (MPI_ANY_SOURCE or MPI_ANY_TAG), the Receive event will be matched by several Send events. The second kind of matched events are non-blocking send or receive with test or wait when their request handles match. The last kind of matched events are collective events when they are in the same type (e.g. Bcast) throughout the state vector at a specific point.

Data Structures

- Log File

This object represents the physical logfile obtained from a process.

- Event

An action which performs an MPI communication function such as MPI_Send is represented by an entry in a logfile plus some state information, such as if this event is going to be removed from the state vector and if a non-blocking event is completed or not. There are six types of events based on the six categories of MPI functions: Blocking event; Non-blocking event; Non-blocking complete event; Collective event; Test event; Persistent event (not implemented). A null event is a dummy event. For the details, please refer to the Communication Category in the Table 15.

- State Vector

An array of events, the dimension of the array is the number of processes used by the MPI program. For example, if a program runs four processes, the dimension of the state vector is four. Each event of the array comes from each logfile, representing the execution state. The logfile records the events according their execution order. The events in the same logfile have chronological differences. If we say each logfile only provides the local state of communication, then the state vector can provide the global state of the whole system including all of the processes. Due to the running randomness of a parallel program, the events in the same state vector are not required to occur at the

same time in the real execution. However, it is one instance of all possible occurrences. Each entry of the state vector can be updated sequentially. For example, updating an event of the state vector means the replacement of the event with the next entry of the corresponding logfile. The State Vector gives us the possibility to trace the communication state transitions of the whole system.

- Event Pair

A pair of events. It is used as a node in the Pending Event List and Complete Event List. An Event Pair may appear as a Matched Pair. An Event Pair may contain the null events.

- Pending Event List (Pending List)

A list contains all events that have been started, but not have been completed. For example, a non-blocking send event (MPI_Isend) in the pending list means that the program has called MPI_Isend, but has not called non-blocking completion function (e.g. MPI_Wait) to complete it. The Pending Event List is initialized to be empty when the Analyzer starts to run. The Pending Event List shows which non-blocking events are still waiting for completion at the current point.

- Complete Event List (Complete List)

A list contains all the events that have successfully completed. For example, there are a blocking send and a blocking receive in the same state vector, and they are matched.

We can move this pair into the complete event list. There are two sources for the events in the Complete Event List. One is directly from the State Vector, the other one is from the Pending Event List. The Complete Event List provides complete information about what communications have been successfully finished in this execution.

- Request Table

A hash table to contain all request handles generated by the non-blocking events. The table uses event sequential ID as the search key. The non-blocking completion events use the information in the Request Table to search their matched non-blocking events in the Pending List.

5.4 The Interface

The Analyzer consists of a User Interface and an Analyzer Engine, or Engine in abbreviation. The following code fragments show the interface between the User Interface and the Engine.

1) The User Interface

It takes the arguments passed in by the user, validates these arguments, initializes the Engine, runs the Engine and returns the results to the users.

```
Main(argc, argv[])
{
     Evaluates the arguments taken from the user;
     If (the arguments are invalid)
     {
          Print error message;
          Exits the program;
```

```
        }
        Initialize the Engine object;
        Add log files to the Engine object;
        Run the Engine;
        Print the Engine running results;
}
```

2) How the Engine runs

After the Engine has started, it takes event entries from each logfile respectively, continues to check if the state vector can be updated. If the state vector is updateable, it runs the updating process ValidateEvent(n), otherwise, it generates an error code and returns.

```
Engine::Run(…)
{
      While (the State Vector is updateable)
      {
            Scan events in state vector;
            if (Event n in the state vector is updateable)
                  ValidateEvent(n);
      }
      If ( NOT all log files have been processed ||
            The any events stays in the pending list)
            Generate error code;
      Return;
}
```

3) ValidateEvent(n) in the (2) contains most of the algorithms to update events in the state vector and to add events to the pending list or complete list. We discuss these algorithms in detail in the following sections.

5.5 Algorithms

The major task of the Analyzer is to search for matched events in both the state vector and the pending event list in a predetermined manner. If the communication sequence is consistent, the state vector should be kept updating until all log files become empty. At the very last step, the pending event list should also be empty. All communication events have been paired and put into the complete communication list sequentially. If the communication sequence has an inconsistency, the state vector updating either stops in the middle, or completes with some events left in the pending list. Hence, the major issues focus on the following three points:

- When the state vector can be updated;
- When the events can be moved from the state vector to the pending list;
- When the pending events can be moved from the pending list to the complete list;

Algorithm 1: Check and Update State Vector

```
In case when the input event is

1. Blocking Send/Receive Event (pre-log)
- Find the matched blocking event in the state vector. If
  a matched pair is found then put it into the complete
  list and update the state vector.
- Use Algorithm 2 to find the matched non-blocking event
```

in the pending list, if found, put the input event and the matched event into the pending list.

2. Non-blocking Send/Receive Event (post-log)
- Find the matched non-blocking event in the state vector. If a matched pair is found put it into the pending list And update the state vector.
- Use Algorithm 2 to find the matched non-blocking event in the pending list. If an event is found put the input event and matched event into pending list; otherwise put the input event paired with a null event into the pending list and update the state vector.

3. Non-blocking Complete Event (pre- and post-log)
- At the pre-log stage, put the event into the request table, using the event sequential ID as the key.
- At the post-log stage, find the event from the request table by the event sequential ID, then use both pre-log and post-log data to get some extra information. It will be useful in the cases of WaitAny or persistent communications.
- Use Algorithm 2 to find the matched event in the pending list

• In case that one event is a blocking and the other is a non-blocking, then put the pair into the complete list and update the blocking event entry of the state vector.
• In case that the both events are non-blocking, if the both events are marked as complete, put the pair into

```
   the complete list; otherwise mark the matched event as
   incomplete.

4. Test (pre-log and post-log)
- If the communication is complete then follow the rules
   in "Non-blocking Complete Event" and update the state
   vector if there is one blocking event.

5. Collective Events (pre-log)
-  Find the matched events in the state vector. If all
   Events are matched then put the events into the
   Complete Event List and update the state vector, if
   events are not matched and the state vector can not
   be further updated, generate error message.

6. Persistent Event (pre- log and post-log)
-  To be implemented
```

Algorithm 2: Find a matched event in the pending list by Comparing the input event with all events in the pending list.

```
In case when

1. Input is either blocking or non-blocking, and the event
   to be compared is non-blocking
- Compare source, destination and tag.
- Return TRUE if they are matched, otherwise continue to
  compare with the next event.

2. Input is a non-blocking complete event (implies there is
   at least one non-blocking event in the pending list)
- Compare the request handle with each of non-blocking
  events in the pending list.
- If a matched non-blocking event is found, return TRUE.
  Otherwise, this should generate an error message.

3. Input is a Test event
- Compare the request handle with each of non-blocking
  events in the pending list.
- If a matched non-blocking event is found, return TRUE.
  Otherwise, return false.
```

The following figure provides us a simplified checking flow in the Analyzer. It shows a state vector must go through all checking procedures before it can be updated. The error message and final successful communication sequences are the result of the Analyzer.

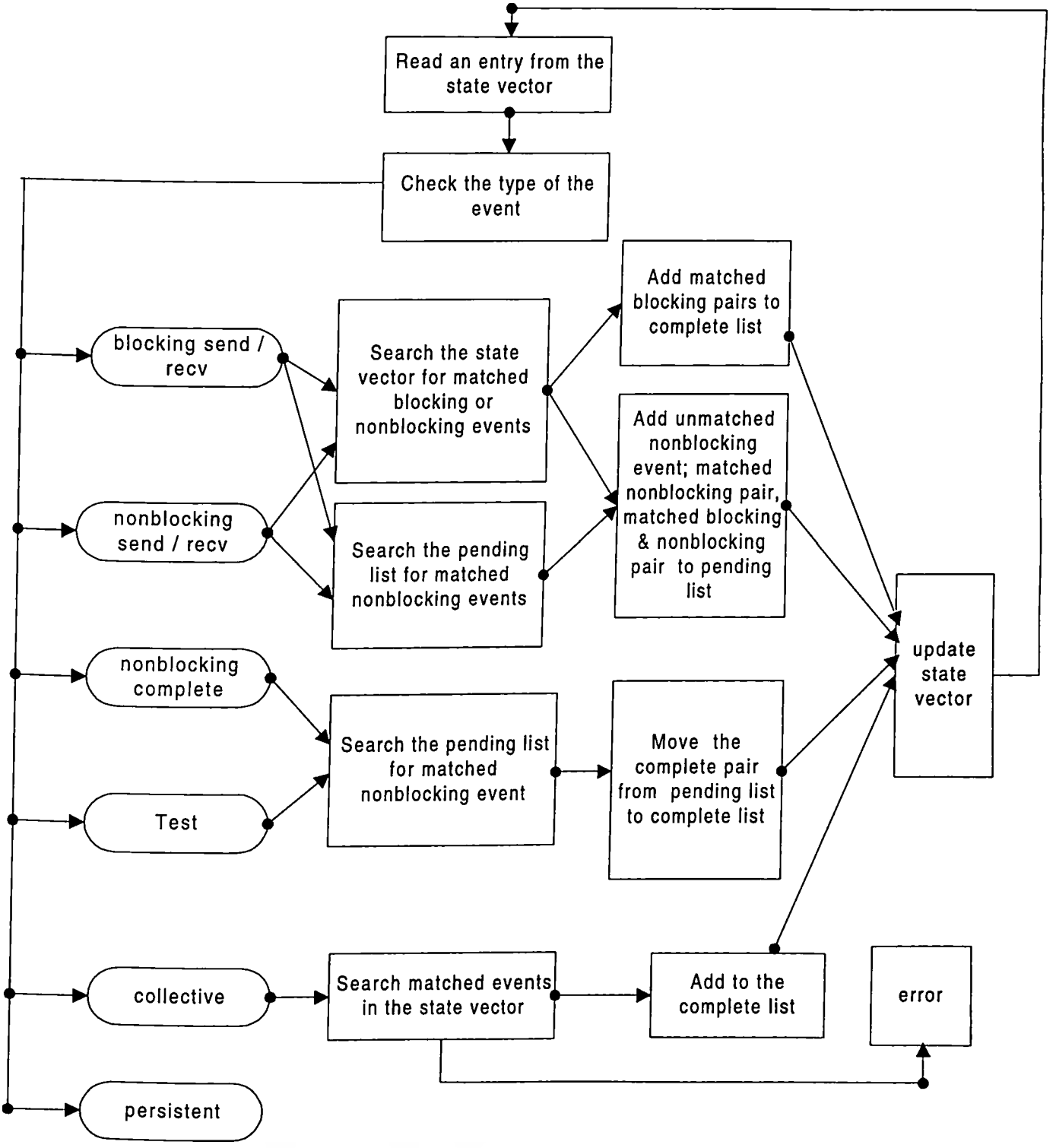

Figure 5. Simplified Checking Flow in the Analyzer

5.6 Implementation Details

All the functionalities of the Analyzer are implemented in the executable — MPICheck. It takes log entries line by line as the inputs to calculate the state vector. An event entry is usually represented by one line. However, two consecutive lines with the same sequence ID are used to provide enough information in cases of Wait or Test events. As shown in the following example, lines starting with 11, 12, 13, 15 represent one event per line, where two lines started with 14 represent one MPI_Wait event.

```
...
11,72,296,0,91,,,,()
12,82,770,1,91,0,2,1,(2324)
13,92,514,0,91,0,2,3,()
14,102,1059,0,,,,,(2324)
14,102,1059,1,,,,,(0)
15,112,545,0,91,2,0,5,()
...
```

Figure 6. A Segment of a Logfile

a) It generates one possible communication trace and shows which events have completed or are still remaining in the pending list. It also indicates whether the communication path is in the current program or not.

Rank 0 Rank 1 Rank 2 Rank 3

Completed Events:
[Ln #10] Send(1) [Ln #10] Recv(0)
[Ln #11] Send(3) [Ln #20] Recv(2)

[Ln #21] Send(3) [Ln #40] Recv(2)
[Ln #20] Recv(3) [Ln #41] Send(0)
.....

[Ln #105] Wait(0)

State vector cannot be updated anymore!

The Last State Vector:
[Ln #105] Wait(0) [Ln #28] Testall(2,3) [Ln #31] Recv(0) [Ln #51] Recv(2)

Pending Events:
[Ln #101] Issend(2,1126)

[Ln #30] Ibsend(3,3126) [Ln #51] Recv(1)
[Ln #22] Isend(3,2129)

[Ln #25] Isend(2,2131)

[Ln #26] Isend(3,2132)

MPICHECK stopped!

REMARKS:

The output parameters for each MPI function are :

Blocking Send/Receive – Destination/Source rank

Non-blocking Send/Receive – Destination/Source rank + Request handle

Waits and Tests – Index of request handle

Collectives – None

Detailed information can be found in state vector log file.

Figure 7. A Segment of MPICHECK Output.

b) It generates a state vector log associated with the communication trace in a) and indicates which event triggers the state vector update. All detailed information can be found in the file.

```
.....
[Ln #40] Isend(1,1124)*          [Ln #13] Send(2)   [Ln #23] Irecv(1,3124)*   [Ln #43] Ssend(0)
[Ln #50] Isend(2,1125)*          [Ln #13] Send(2)   [Ln #24] Recv(0)          [Ln #43] Ssend(0)
[Ln #60] Recv(3)*                [Ln #13] Send(2)   [Ln #24] Recv(0)          [Ln #43] Ssend(0)*
[Ln #70] Testany(1124,1125,)*    [Ln #13] Send(2)   [Ln #24] Recv(0)          [Ln #44] Ssend(1)
[Ln #70] Testany()*              [Ln #13] Send(2)   [Ln #24] Recv(0)          [Ln #44] Ssend(1)
[Ln #80] Waitall(1124,1125,)*    [Ln #13] Send(2)   [Ln #24] Recv(0)          [Ln #44] Ssend(1)
[Ln #90] Waitall(0,1,)*          [Ln #13] Send(2)   [Ln #24] Recv(0)*         [Ln #44] Ssend(1)
.....
```

Figure 8. A Segment of State Vector Updating Log

(The events with "*" are those events which trigger state vector updates)

c) In addition, it provides some options to format and control output.

MPI Function Name Coding Convention

All MPI function names are coded according to their event names. The coded name provides flexibility in our system. The coded event names are used in both the Logger and the Analyzer.

An event name is coded as a 12-bit bitmap that describes the event type, the event mode and the event function. Figure 9 is the layout of such a bitmap. Tables 17, 18 and 19 show detailed information about each field of the bitmap.

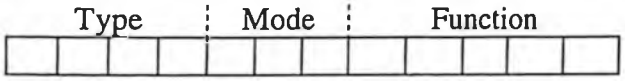

Figure 9 Event Bitmap

Value	Mode
0	Standard mode
1	Synchronized mode
2	Ready mode
3	Buffer mode

Table 17 Mode Values

Value	Type
0	Null Event
1	Collective Event
2	Blocking Event (Send / Receive)
3	Non-blocking Event (Send/Receive)
4	Non-blocking Complete Event (Wait)
5	Partial Non-blocking Complete Event (Test)
6	Persistent Event
7-15	Reserved

Table 18 Type Values

Value	Function Type
0	Null Function
1	Send
2	Receive
3	Wait
4	Test
5	Persistent Start
6	Collective Broadcast

7	Collective Reduce
8	Collective Gather
9	Collective Scatter
10-31	Reserved

Table 19 Function Values

For example, a non-blocking ready-mode send function can be described as Type = 3, Mode = 4, Function = 1, so that MPI_IRSend = 3 * 2^8 + 4 * 2^5 + 1 = 897.

In this chapter, we addressed the detail design and implementations about the Analyzer. In the chapter 6, we will do case study in order to prove the tool utility.

Chapter 6

Case Study

In order to evaluate the practicality and effectiveness of the methods described in the previous chapter, and to gain an appreciation of their strengths and weaknesses, we designed various cases to fully test every part of the tool. The test cases are classified into two categories, hypothetical cases and real applications. The hypothetical cases focus on testing the core part — the Analyzer. The real applications focus on testing every part of the tool, including the Pre-processor, the Logger, the Wrapper and the Analyzer. Also through testing these applications, we will show how the tool participates in parallel program developing and what role it plays in checking communication consistency. In the Section 6.1 and 6.2, we will discuss the hypothetical testing cases and a real application, respectively.

6.1 The Hypothetical Testing Cases

The hypothetical cases focus on testing the core part – the Analyzer. The Analyzer takes the logfiles as inputs, goes through all the checking procedures and returns results

to the user. Most of the crucial algorithms are implemented in this part. The accuracy and reliability of this part will affect the evaluation of the whole tool. Therefore we put more emphasis on testing this part.

In order to test the Analyzer intensively, we need various logfiles containing some sequences simulating real-time communications during parallel program executions. The advantage of hypothetical testing cases is we can design any kind of communication sequences purposely. It gives us more flexibility and control to generate various logfiles, from simple ones to complicated ones. If we only rely on the execution of programs to generate logfiles, it will restrict logfiles' variety, because we have to change programs in order to get different sequences. Since the communication is dynamic, it would be difficult for us to get some critical sequences. From the testing point of view, we need more control to the logfile sample resources. The hypothetical cases can solve this problem easily and ideally.

In the following, we will give some hypothetical testing cases. For each case, we will describe what we want to test, what we expect to happen and what actually happens. We generated some logfiles containing consistent communication sequences and some logfiles containing inconsistent communication sequences. We gave these logfiles to the Analyzer and expected those inconsistent ones could be detected. In Table 20, we list the testing numbers of the typical case. We also chose some general cases which were not included in this chapter. Due to the space restrictions, we only give a few typical examples. For readability reasons, we use a table to describe that part of the logfiles where the problems are located.

The Testing Purpose	The Number of Testing Cases
Consistent sequences	10
Deadlocks caused by mis-ordered blocking events	5
Deadlocks caused by mis-matched blocking events (passing by wrong parameters, no corresponding events called)	7
Message pending caused by no completion events called (Test, Wait)	10
Message pending caused by only calling test	5

Table 20 Testing Cases

CASE NO. 1

PURPOSE:

Detecting deadlocks or potential deadlocks due to mis-ordered sends and receives. Also detecting the potential pending events in the sequence.

DESCRIPTION:

We added a ring structure in the logfiles. Since blocking communication was used, the dependency between the senders and receivers might cause serious deadlock. We assume the deadlock occurred, and that the logfiles were partial. The communication sequence is shown in the following figure.

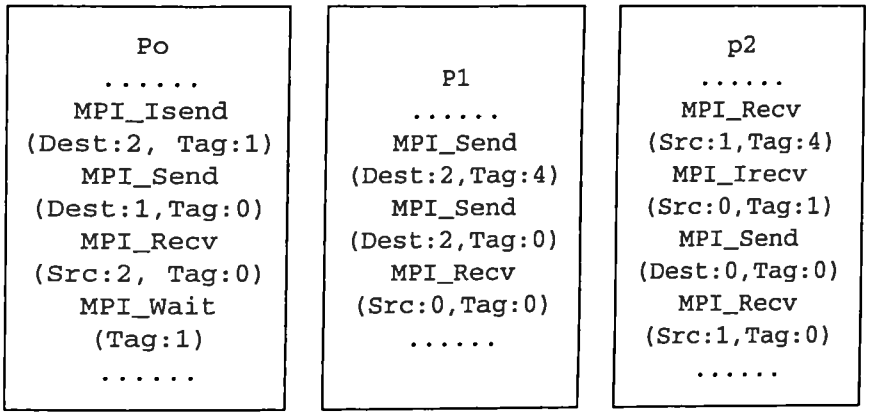

Figure 10 Case No.1

EXPECTED RESULT:

After running the Analyzer, we should get the following results from the standard output, also a state vector transaction log file:

1. Final State Vector:

.....

[Ln #] MPI_Send(1) [Ln #] MPI_Send(2) [Ln #] MPI_Send(0)

2. Pending Events:

.....

[Ln #] MPI_Isend(2) [Ln #] MPI_Irecv(0)

3. Complete Events:

.....

[Ln #] MPI_Send(2) [Ln #] MPI_Recv(1)

TESTING RESULT:

Same as the expected results. This case was based on the assumption that the system did not provide a large system buffer. In the real application, the same MPI function calls might go through successfully if the system could provide enough buffer space. If no

deadlock happens and the logfiles are complete, our tool still returns the same final state vector warning there is potential error in this State Vector. It did not occur this time, but it may occur in the future.

CASE NO. 2

PURPOSE:

Detecting deadlocks caused by improper use of "non-blocking completion operations".

DESCRIPTION:

In this case, the logfiles contained some non-blocking communication events. We used "Wait" to complete these non-blocking events. We assumed the deadlock occurs due to the improper "Waits" called.

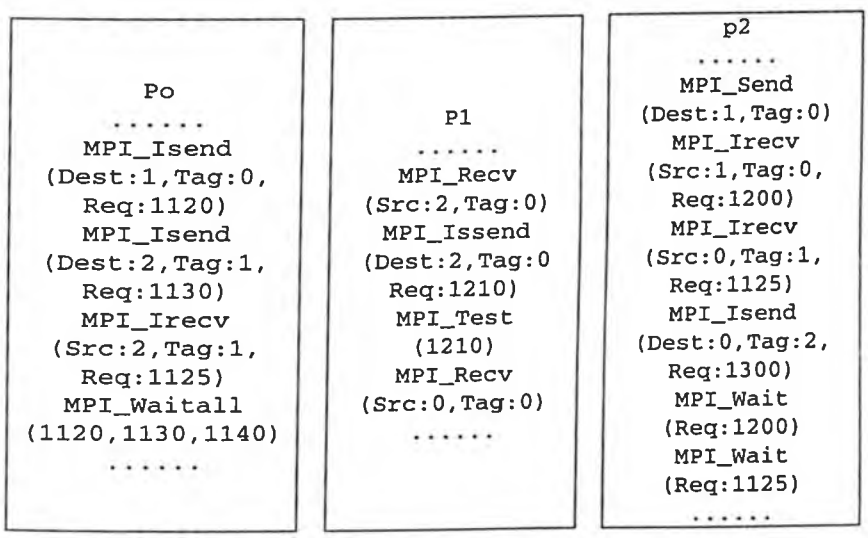

Figure 11 Case No. 2

EXPECTED RESULT:

After running the Analyzer, we should be able to obtain the following outputs:

1. Final State Vector:

[Ln #]MPI_Waitall(1120,1130,1140) [Ln #]MPI_Recv(0) [Ln #] MPI_Wait(0)

2. Pending Events:

.....

[Ln #] MPI_Isend(1,1120) [Ln #] MPI_Recv(0)
[Ln #] MPI_Isend(2,1130) [Ln #] MPI_Irecv(0,1125)
[Ln #] MPI_Irecv(2,1125) [Ln #] MPI_Isend(0,1300)

3. Complete Events:

.....

[Ln #] MPI_Recv(0) [Ln #] MPI_Send(1)
[Ln #] MPI_Issend(2,1210) [Ln #] MPI_Irecv(1,1200)

TESTING RESULT:

We compared the testing results (output and state log). They were same as the expected results. The "MPI_Wait" belongs to local blocking operation. One of the "Requests" issued by the "MPI_Waitall" on P0 failed because none of the non-blocking operations used this handle. Thus, the "Waitall" was blocked and caused a serious deadlock. If P1 and P2 did not need to communicate with P0, they would continue executing.

CASE NO. 3

PURPOSE:

Detecting the potential pending events left after program execution.

DESCRIPTIONS:

There were some non-blocking events contained in the logfiles. Some of them were not completed or cleaned up before the program terminated. Too many pending events could exhaust system resources and cause serious unpredictable errors.

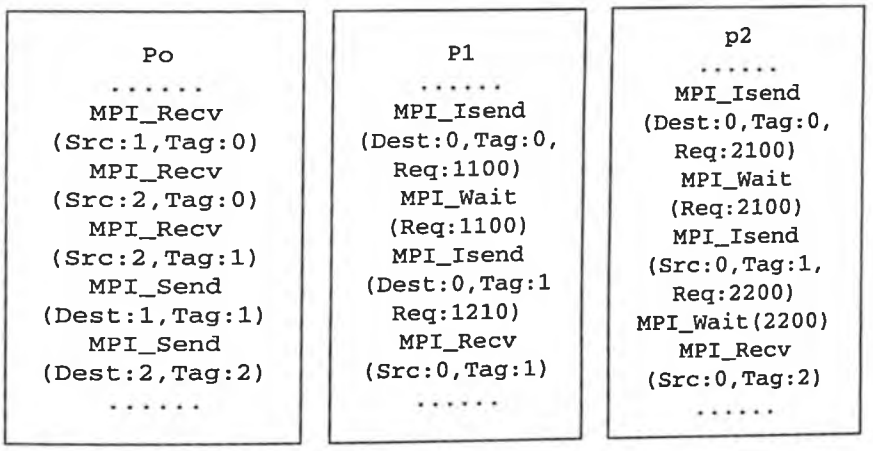

Figure 12. Case No. 3

EXPECTED RESULTS:

After running the Analyzer, we should be able to get the following outputs:

1. Since no Final State Vector is reported, no deadlocks occurred.
2. Pending Events:

.....

[Ln #] MPI_Isend(0,1210)

3. Complete Events:

.....

[Ln #] MPI_Recv(1) [Ln #] MPI_Isend(0,1100)
[Ln #] MPI_Recv(2) [Ln #] MPI_Isend(0,2100)
[Ln #] MPI_Recv(2) [Ln #] MPI_Isend(0,2200)
[Ln #] MPI_Send(1) [Ln #] MPI_Recv(0)
[Ln #] MPI_Send(2) [Ln #] MPI_Recv(0)

.....

TESTING RESULTS:

Checking the outputs and state log, the testing results were same as the expected results. P1 sent a message to P0, but P1 did not issue a completion operation to complete this blocking send. Hence, that message could become a pending message.

6.2 Real Application

In the last section, we focus on testing the core part — the Analyzer. In this section, we do integrated testing, which includes testing the pre-processor, the logger, the wrapper and the Analyzer. In order to address the problem clearly, we present a real application: adaptive quadrature using the rectangle rule. We will describe the whole debugging procedure to show how our tool assists the programmer to detect the communication errors and improve program reliability.

6.2.1 Program Overview

• PURPOSE

Using MPI to develop a parallel software version for adaptive quadrature using the rectangle rules. In this application, we need to cover the issues such as dynamic load balance, communication structures and program termination.

• **COMPUTING NUMERICAL INTEGRATION**

In this program, we chose the rectangle rule to compute the numerical integration. For each level, we divide the interval [a,b] into half, as long as the error is greater than the tolerance, or stop dividing if either the level is greater than max_level, or the error is less than the tolerance. The interval, tolerance and max_level were controlled by the user interface.

• **DYNAMIC LOAD BALANCING**

When there are multiple processes joining the computation, maintaining the work load balance is very important for improving the performance. More simply, try to keep everyone busy all the time. In our program, distributing and managing jobs are two major issues to control load balance. We follow the principle that communication is always more expensive than the computation. So, we should keep the job executing as locally as possible, and increase the size of sending messages so as to decrease the communication times.

• **COMMUNICATION STRUCTURE**

We use Figure 13 and 14 to describe the major communication patterns we designed for the system.

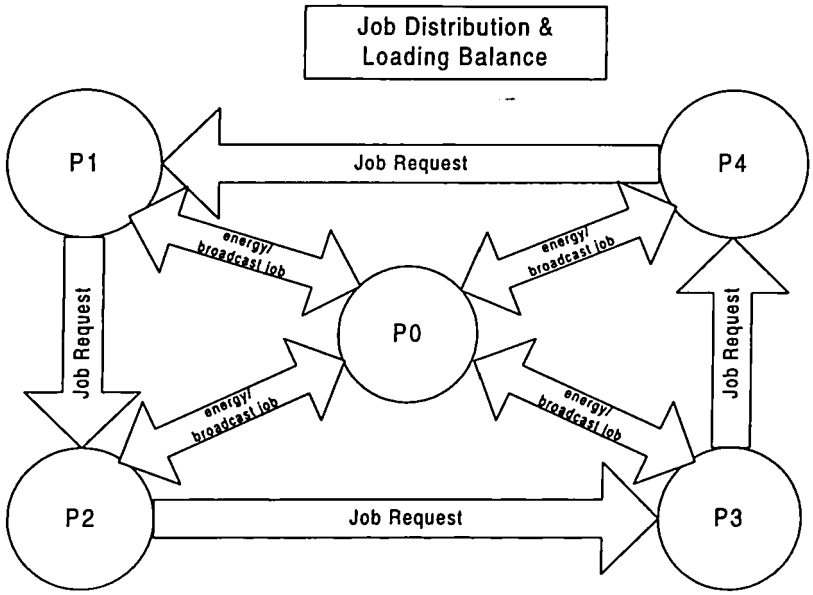

Figure 13. Communication Pattern One

1. P0 obtains the whole work at the beginning. It distributes the approximately equal parts of the work to the other processes.
2. Each process maintains a local work stack. If all the local work has been finished, the process sends the "Job Request" message to the process on its right. If it receives the "Reject Request" message, it increments the rank of the target by adding one each time (wrapping around). If it receives the "Work" from the other processes, it pushes the work into its local work stack.
3. When the process receives the "Work Request", it always sends half or less than half of its local work to the other process. If it does not have enough work to send, it will send the "Reject Request" message to the requesting process.

4. Each process sends the finished work in terms of energy to the P0.

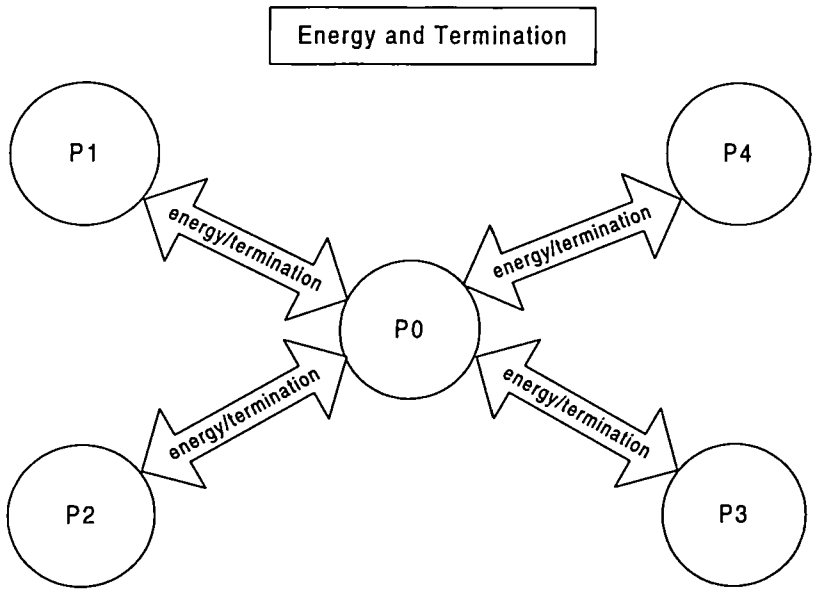

Figure 14. Communication Pattern Two

1. P0 decides when the program should terminate by adding the energy received from all the other processes. When the total energy equals to one, all the work has been finished. Next P0 broadcasts the termination message to all the processes, then shuts itself down too.
2. Each process computes the local energy after they finish the current local work, then sends the energy to P0. When they receive the "Termination" message, they terminate immediately.

• **MODULE DESCRIPTION**

The system has six modules: input module, message passing module, work_stack module, local energy module, work module and result module. The input module provides the user interface. The user can interactively input parameters such as the interval, tolerance, etc. The work_stack module provides the local work management. The work module is responsible for calculating the local energy and numerical integration value. The most crucial part is the message passing module. All communications are implemented in this module. It handles work distribution, sending and receiving work, requests, rejects and energy. It also takes care to signal the system to terminate.

• **MPI EVENTS**

The MPI events used in the message passing module involve four communication categories as we mentioned in chapter 3. They are blocking communication, non-blocking communication, non-blocking completion operation and collective communication. The communications occurred in the message passing module cover most of the typical MPI functions, such as "MPI_Send", "MPI_Isend", "MPI_Bcase" and "MPI_Wait".

6.2.2 Problem Solving with the Tool

In this section, we describe two errors which occurred during the testing and how the Automatic Logfile Analyzer assisted us to detect them. These two errors are

deadlock and message pending. We skipped the work spent on debugging the sequential errors, and only discuss how to detect the programming errors involving communications.

•DEALING WITH COMMUNICATION ERRORS WITHOUT THE TOOL

In order to test the communication consistency, we run the program on different number of processes with different architectures and work loads. The testing results were not stable. Some executions were successful and returned the correct answers (we compared the results with MatLab and assumed the results from MatLab were correct). Some executions encountered serious deadlock, the whole program hung and nothing was returned. The message in the core file could not provide us with enough information to indicate where and what the problem could be. The deadlock happened so unpredictably that it was very difficult to trace.

• DEALING WITH THE COMMUNICATION ERRORS WITH THE TOOL

After we introduced the Automatic Logfile Analyzer, the deadlock was detected quickly. The procedure is shown in the following:

A. Including the header file "mpiwrap.h" into the message passing module program which is "message_passing.c".
B. Running the pre-processor to replace the original MPI functions with the wrapped MPI functions, and meantime generating the wrapper function library.
C. Compiling the program to get the executable.

D. Executing the program, generating the logfiles.

E. Running the Analyzer to process the logfiles, obtaining the results.

F. Checking the outputs and state transaction log to locate the problems.

The logfiles were generated by two executions, one was successful and one failed. For the first one, the logfiles were complete. For the second one, the logfiles were partial. We used the tool to analyze these logfiles, and obtained the similar results. The checking stopped in the middle, and final state vectors were returned. The final state vector showed there were two blocking send operations in the same state. They were trying to send different messages to each other. From a logic perspective, this dependency could cause deadlock if the system could not provide enough buffer size. The analyzing results were reasonable, because we tested our program on different machines with different architectures and system configurations. We used blocking send operations, that were not placed in proper order. Whether an execution was successful or not totally depended on the runtime system buffer size. This was the reason why the deadlock happened dynamically. According to the line number, we located the source code where the events were called, then we changed the blocking send (MPI_Send) to non-blocking send(MPI_Isend) or (MPI_Bsend).

After doing this, the deadlock did not occur. The checking went through all the logfiles without stopping. But we found after the checking finished, that there were still some events left in the pending list. We knew if the communication sequence was consistent, all the pending events should be paired and moved to the completed list. We found these pending messages were the very last "Request Work" messages sent by

different processes. The reason was that one process such as P1 sent out a "Request Work" message to another process such as P2, but P2 received the "Termination" message before the "Request Work", then it terminated. P1 also received the "Termination" message from P0 after it sent the "Request Work", then it terminated immediately without cleaning up the messages, which were already sent, but have not been received. Both the sender and receiver for the "Request Work" message terminated, resulting in the message "Request Work" becoming a pending message. After this realization, we added the clean up operations for every process, then the problem was solved.

The application we described shows that the Automatic Logfile Analyzer is very helpful to detect communication errors. It can detect not only the errors appearing in executions, but also the potential ones. Without this help, we will have very hard time to locate the first error, and have very small chance to detect the second one. So, the tool can reduce the time spent on debugging the communication problems and give the users more confidence about their programs. Finally, it can improve the parallel program reliability.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this last chapter, we will draw some conclusions to highlight the advantages of our tool. Also we will discuss what improvements are expected to be completed in the future.

In the thesis, we have addressed background knowledge and the new ideas behind this tool, the detailed design and implementation and the testing results. Now, we are confident that this tool is useful. It provides a simple and intuitive way of detecting communication errors in MPI programs. The user can save time spent on detecting the communication problems. It provides clear indications, which serve to locate the problem quickly. One of the good things about the tool is that it can detect not

only existing errors, but also potential ones. Thus, it can help the user to deliver more reliable parallel programs.

We find this tool is especially beneficial for beginners to learn parallel programming. From the state vector transition record, they can get the big picture about what is really going on in parallel programs. It provides an overview of the whole system including all processes, not just part of it. With the help of the tool, the user can gain more parallel programming experience quickly. They learn where are the critical points, what are the common communication errors and what communication operations should be chosen.

The tool is designed to be user friendly, it provides complete functionality including pre-processing and post processing. All the mechanisms are hidden from the user. The user interface is simple and easy.

All functions included in the tool are designed to be easily extended and reused. The MPI functions can be easily replaced by other similar communication library functions with small changes. For example, since our implementation is based on an MPI program in C, there would be no big change to make our system work for the MPI program in FORTRAN. The only change should take place in the Pre-processor because of the function prototypes and the calling convention.

7.2 Future Works

Due to time restriction, we cannot cover everything desired in the current version. There still are some improvements expected to be finished in the future in order to make the tool more powerful and complete. We list a few of them below,

1. Supporting communication sub-domains

   Our current version only supports one communication domain which includes all of the processes called MPI_Comm_World.

2. Supporting persistent communications

   Our current version supports blocking, non-blocking and collective communication. We need to add some new rules in order to support persistent communications.

3. Error Levels

   Our current system does not implement error level detection. All potential communication errors are considered as fatal. The Analyzer will not continue unless the communication error has been resolved. To add some error level detection may reveal the potential problem in the program as a whole, so that it would improve the code efficiency.

4. Code Optimization

The current code is optimized at the compiler level. The code size is relatively large because of using the Standard Template Library (STL). The code size may be dramatically reduced if only C along with some optimized library code is used.

Appendix 1. Reference Manuals

MPICHECK (1.0)

Purpose:

To perform a deadlock check against an MPI* program written in C.

Syntax:

mpicheck [-s*filename*] [**-c**] [**-p**] [**-v**] [**-u**] [**-t#**] [**-w#**] [**-h**] [**–l#***logfile*]...

Description:

This program takes input logfiles generated by the MPI programs that are interpolated with the MPIWRAP runtime routines, then runs the deadlock-check algorithm against the MPI program. It formats event updating status to a file, the last state vector and the complete event pair list to stdout. Finally it produces an analysis result for the MPI program.

Flags:

-l#*logfile*

To specify the processor rank and its associated logfile. Up to 8 processors/logfiles can be defined by this version. # ranges from 0 to 7. logfile must be a valid path to the logfile produced on the corresponding processor #.

-s*filename*

To specify the output file for the state updating status. (default = states.log)

-c

To suppress creating the complete event pair list (default = not)

-p

To suppress creating the pending list (default = not)

-v

To suppress creating the last state vector if state vector cannot be updated (default = not)

-u

To suppress creating the state vector updating file (default = not)

-t#

To define the size of a tab stop. This flag only affects the output formatting. (default = 4)

-w#

To define the number of tabs per column that represents a processor. (default = 5). This flag only affects the output formatting.

-h

To display a help screen for command line usage

Exit Status:

0 – Successful

1 – Invalid command line switch

2 – Invalid rank number

3 – Logfile open error

4 – Logfile is either corrupted or in an invalid format

5 – No logfile specified

6 – The state vector cannot be updated anymore, potential deadlock

7 – The pending list is not cleaned up even though the MPI program terminates normally.

Examples:

1) Run mpicheck on two logfiles p0.log and p1.log, which were generated on processor 0 and 1 respectively, use tab size 8 and 5 tabs per processor rank.

```
mpicheck -l0p0.log -l1p1.log -t8 -w5
```

```
MPICHECK Version 1.0 Copyright (c) 1999 by Haitong Zhang
*** (Email: hzhang@church.cas.mcmaster.ca) ***
Rank 0                                  Rank 1

Completed Events:
[Ln #38] BCast(...)                     [Ln #38] BCast(...)
[Ln #39] Gather(...)                    [Ln #39] Gather(...)
[Ln #40] Allgather(...)                 [Ln #40] Allgather(...)
[Ln #44] Scatter(...)                   [Ln #44] Scatter(...)
[Ln #45] Reduce(...)                    [Ln #45] Reduce(...)
```

```
[Ln #46] Allreduce(...)                [Ln #46] Allreduce(...)
[Ln #53] Isend(1,793232)               [Ln #71] Recv(0)
[Ln #61] Testall(0,1,2)
[Ln #54] Issend(1,793340)              [Ln #72] Irecv(0,713824)
                                       [Ln #73] Wait(0)
[Ln #55] Irsend(1,793448)              [Ln #74] Irecv(0,713824)
                                       [Ln #82] Waitall(0,1,2)
[Ln #62] Issend(1,793556)              [Ln #75] Irecv(0,713976)
                                       [Ln #82] Waitall(0,1,2)
[Ln #64] Irsend(1,793664)              [Ln #76] Irecv(0,714128)
                                       [Ln #82] Waitall(0,1,2)

MPICHECK run successfully!
```

Files:

Mpicheck

Remarks:

* Supports most blocking and non-blocking functions defined in MPI 1.0

MPILOG (1.0)

Purpose:

To interpolate the event-logging enabled code into MPI programs written in C. This is the preprocessor for MPICHECK.

Syntax:

mpilog [–*spath*] [–**i***ext*] [-**o***ext*] [-**b**] [-**c**] [-**p**] [-**h**] *file1*, *file2*, ...

Description:

MPICHECK requires "events" must be logged in order to make further analysis. Since the default MPI logging mechanism cannot fit this purpose, some special logging routines must be interpolated into source code. Then these routines can link to a runtime library "libwrap.a" after compiling. The users' MPI program can log all the information required by MPICHECK.

MPILOG takes C source files *file1*, *file2*, ... as input, which must have extension ".mpc" or a name specified in [-iext], then creates output ".c" files. All files that implemented MPI routines must also include the header "mpiwrap.h" to declare the prototypes for the wrapped MPI routines.

MPILOG does not replace any MPI functions that contain syntax errors. It puts a comment at the end of source line.

Flags:

-*spath*

To specify the source file directory (default = ".")

-**i***ext*

To specify input file extension (default = ".mpc")

-**o***ext*

To specify output file extension (default = ".c")

-b

To disable source file backup : overwrite existing ".mpc" files if they are in the same path as output files. (default = enabled)

-c

To disable continue on error (default = enabled)

-p

To supress print statistics.

-h

To display a help screen for the usage.

Examples:

1) To interpolate logging runtime routines to a MPI C source file "mpitest.mpc". A file "mpitest.c" is generated after running mpilog. Statistics about how many times each MPI function has occurred in this file are also output to stdout.

```
mpilog mpitest.mpc
MPILOG Version 1.0 Copyright (c) 1999 by Haitong Zhang
*** (Email: hzhang@church.cas.mcmaster.ca) ***

MPI_Init        : 1
MPI_Finalize    : 1
MPI_Bcast       : 1
MPI_Reduce      : 1
MPI_Gather      : 1
```

```
MPI_Scatter    : 1
MPI_Allreduce  : 1
MPI_Allgather  : 1
MPI_Send       : 1
MPI_Recv       : 1
MPI_Irsend     : 2
MPI_Issend     : 2
MPI_Irecv      : 4
MPI_Wait       : 1
MPI_Waitany    : 1
MPI_Waitsome   : 1
MPI_Waitall    : 1
MPI_Test       : 2
MPI_Testany    : 1
MPI_Testsome   : 1
MPI_Testall    : 1
```

MPIWRAP Runtime Library (1.0)

Purpose:

To log MPI events at runtime.

Usage:

Link the libray "libwrap.a" with –lwrap switch during linking executable.

Description:

The runtime library combines both wrapped MPI functions and the event logging functions. The wrapper functions have the similar look as their real counterpart. Since at the preprocessing phase, MPILOG already changes the real MPI function prototypes to the wrapper function prototypes. During compiling phase, the wrapper and logger work together to log MPI events.

Examples:

1) to link library "libwrap.a" which was installed in the path

"/u0/grad/hzhang/MPIWrap"

-L/u0/grad/hzhang/MPIWrap –lwrap -llmpi -lmpe -lm -lX11 -lpmpi -lmpi

Files:

libwrap.a

References

1. V. Herrarte and E. Lusk. Studying parallel program behavior with upshot. Technical Report ANL-91/15, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1991.
2. A Message-Passing Interface Standard. Message Passing Interface Forum.
3. Al Geist, Adam Beguelin, Jack Dongarra, Robert Manchek and Vaidy Sunderam. PVM: Parallel Virtual Machine. MIT Press, 1994.
4. R. Hempel, H.-C. Hoppe, and A. Supalov. PARMACS 6.0 library interface specification. Technical report, GMD, Postfach 1316, D-5205 Sankt Augustin 1, Germany, 1992.
5. S. Qiao and H. Zhang. An Automatic Logfile Analyzer for Parallel Programs, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Vol. III, Editor: H.R. Arabnia, pp. 1371-1376, Las Vegas, Nevada, USA, June 28 - July 1, 1999.
6. A. Gottlieb, R. Grishman, C.P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer: Designing a MIMD, shared memory parallel computer. IEEE trans. Computs., C-32(2): 175-189, 1983.
7. Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. MPI: The Complete Reference. Cambridge, MA: MIT Press, 1996.

8. High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston., Tex., 1993.

9. Ian Foster. Designing and Building Parallel Programs. Addison-Wesley Publishing Company, 1994.

10. Building an advanced climate model: Program plan for the CHAMMP climate modeling program. U.S. Department of Energy

11. J. Worlton. Characteristics of high-performance computers. In Supercomputers: Directions in Technology and its Applications. National Academy Press, 1989.