An Automatic Logfile Analyzer for Parallel Programs

Sanzheng Qiao
Computing and Software Department
McMaster University
Hamilton, ON L8S 4L7
Canada

Haitong Zhang
Computing and Software Department
McMaster University
Hamilton, ON L8S 4L7
Canada

Abstract *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 and the Wrapper 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 analyzing. Currently, the tool supports all major MPI functions. The tool is implemented in C++. It is extensible and reusable.*

Key words: Parallel Programming, Message Passing, Debugging, MPI.

1 Introduction

Parallel programming using message passing is prone to errors such as communication inconsistency, for example unmatched send and receive. In this paper, we present a new approach to check the communication consistency in parallel programs. It allows us to improve program reliability, detect errors in communication, and also get an overall picture of the execution of a program. Our tool supports most of the major MPI [1] communication functions. Currently, it handles two kinds of communications: 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 communication 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 way. The tool also supports the non-blocking completion functions such as wait and test. Although the current version of the tool does not support all MPI functions, it is designed to be easily extended to support other communication functions.

The basic idea is the use of a logfile to record intermediate communication events occurred during an execution of a parallel program. Then we use the analyzer to process the logfiles automatically, check communication sequences, and detect errors. The whole work is done in two stages, execution stage and post execution stage. In the first stage, the logfiles are generated. In the second stage, the logfiles are analyzed, which is the major purpose of the tool. There are some existing profiling tools using logfiles, such as upshot [2], but they focus on different issues, such as performance tuning, rather than debugging. Although we implemented our tool on top of MPI, the ideas are applicable to other message-passing communication libraries. We implemented our tool in C++ because it provides good modularity. Thus the functions in this tool can be easily extended or reused. The MPI functions can be easily replaced by other similar communication library functions.

Our tool consists of three essential parts and one add-on part. The essential parts are the Logger, the Wrapper, and the Analyzer. The add-on part is the Preprocessor. In the following sections, we describe the functions performed by the four parts and discuss their design issues. The three essential parts are described in the following three sections. The Preprocessor is illustrated in Section 5. Finally, in Section 6 we give a summary by presenting an overall structure of the tool.

2 Logger

The logger records communication events occurred during an execution, hides the implementation details, and provides the interface to the Wrapper. The functionality of the Logger includes writing the log buffer, controlling the frequency of flushing the buffer, and creating and writing the logfiles. During an execution, each process produces a logfile containing communication events.

The logger is implemented as a class. A logger object is created when the parallel program calls MPI_Init, which means the program starts logging communication events. The logger is destroyed when the program calls MPI_Final. The user can choose how often the buffer is flushed. To minimize the lost information due to an unexpected exit of the program, the user can increase the flush rate. The safest way is 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 flush rate to improve program speed.

3 Wrapper

The original MPI functions do not have the ability of tracing communication events. In order to keep track of communication events for later analysis, the functionality of the original MPI functions needs to be extended. The wrapper functions are such extended MPI functions. The Wrapper is also responsible for creating and destroying logger object to control the logging process. We use the layering structure for the wrapper functions. They are designed to have 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.

The first issue is the information structures provided by the Wrapper, since different communication events require different information structures. Our selection criterion for the information structure of an MPI function is that the information should be just enough to identify the communication event for the later analyzing process. For example, the information structure of the blocking point-to-point communications contains the event name, event sequential ID, source rank, destination rank, message tag, communicator, and source code line number. The non-blocking point-to-point communications have additional information about the request.

The second issue is the locations where the logging functions are inserted. Our criterion for the insertion positions is that we can log sufficient information for the later analysis while avoiding unnecessary log messages. For example, for the blocking send and receive, we only insert the logging function before (pre-log) the MPI function call. Whereas for non-blocking completion function wait, we insert the logging function both before (pre-log) and after (post-log) the MPI function call.

4 Analyzer

The Analyzer is the core part of our tool. It works at the post execution stage. It takes logfiles as inputs regardless they are complete or partial, which may be as the result of an abnormal exit during an execution. The Analyzer either proves the communication consistency or detects communication errors.

Since MPI supports both synchronous and asynchronous communications, deadlock and message pending occur quite often. A parallel program may have very different communication patterns in different executions, as the result of the variance in computation capacity, system 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 Section 2 to log all communication events. One logfile is generated by each process. Our analysis is based on the sequence of event entries 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 analysis. In addition, our tool can detect not only the errors, such as deadlocks, which cause the execution hang or fail, but also potential errors, such as incomplete operations in non-blocking communication, which may or may not cause problems in an execution. In this section, we present the main algorithm used in the Analyzer. Before describing the algorithm, we introduce the terms and notations used in the algorithm.

First we divide the MPI functions into the following six categories based on their communication patterns.

- 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 (Wait)
  MPI_Wait, MPI_Waitany, MPI_Waitsome, MPI_Waitall
- Partial non-blocking completion (Test)
  MPI_Test, MPI_Testany, MPI_Testsome, 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

Second we list the terms used in the algorithm.

- Logfile

The data structure represents the actual logfile obtained by a process.

- Event

An action that performs an MPI communication such as MPI_Send. It is represented by an entry in a logfile plus some state information. There are six types of events based on the six classes of MPI functions: Blocking event; Non-blocking event; Non-blocking complete event; Collective event; Test event; Persistent event. A null event is a dummy event.

- State Vector

An array of events, one from each logfile, representing communication state. Each entry is updated sequentially.

- Event Rank

The process rank where the event is logged.

- Matched Event

An event is matched by one or more events. There are three types of matched events: Send and Receive matched events when their source and destination ranks as well as tags match; Non-blocking and Test or non-blocking complete matched events when their request handles match; Matched collective events when they are of same type (e.g., Bcast) throughout the state vector at a specific point.

- Matched Pair

Two matched events one of which is send and the other is receive.

- Complete List

A list of successfully completed events.

- Pending List

A list of all pending events.

- Request Table

A hash table containing all request handles generated by the non-blocking events. The table uses event sequential ID as the key.

The major task of the Analyzer is to search for matched events in a predetermined manner. The following algorithm describes how the Analyzer works. It takes an event from the state vector as an input. If the input event is matched, then the matched events are moved to either the complete list or pending list and the state vector is updated. If eventually all events are moved to the complete list and no events left in either the state vector or the pending list, then the program has no communication errors. Otherwise, the program has communication inconsistency.

Algorithm 1: Check and update the 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 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 complete.

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 complete list and update the state vector, if events are not matched and the state vector can not be updated anymore, generate error message.

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

Algorithm 2: Find a matched event in the pending list

Compare the input event with all events in the pending list. In case when

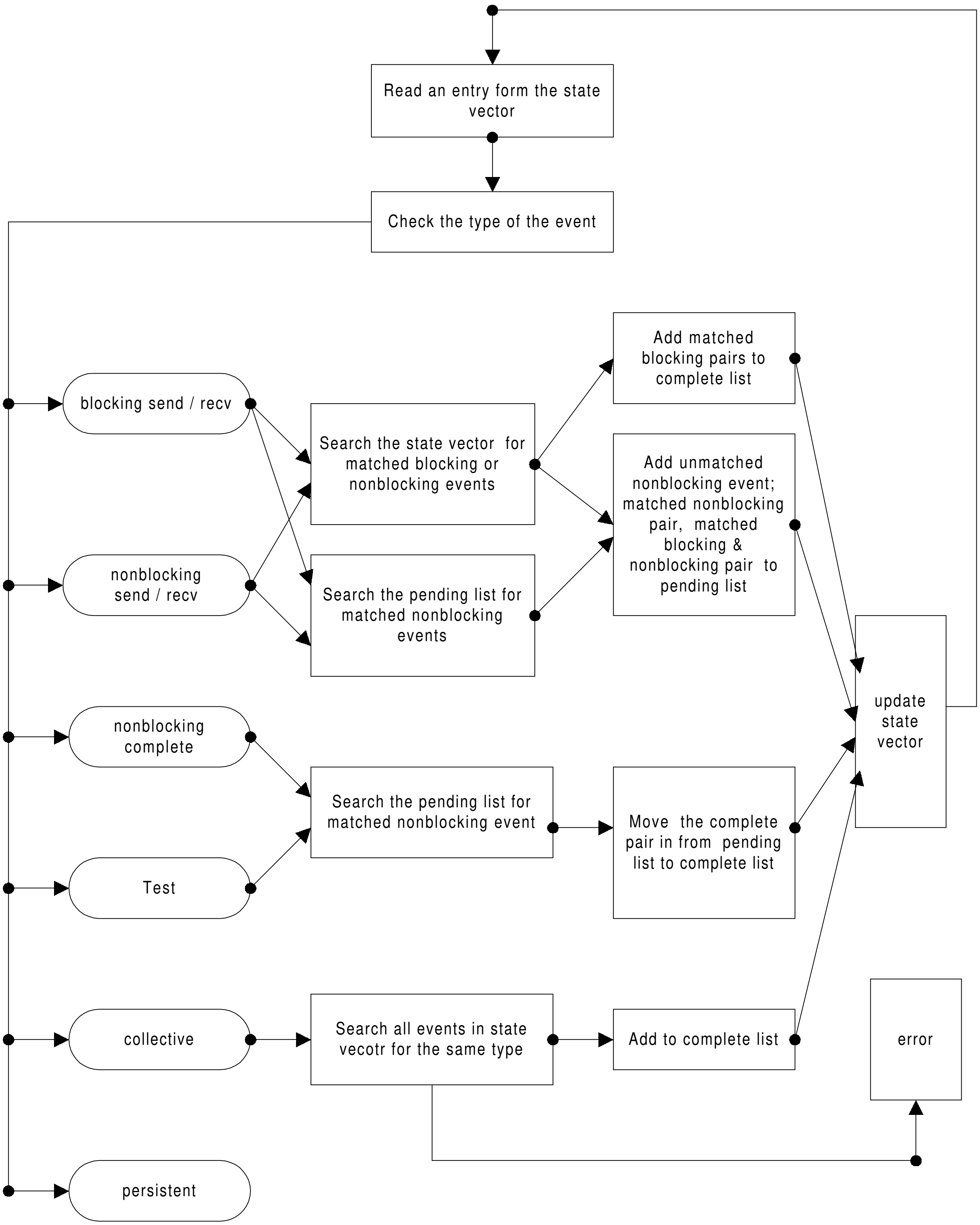
1. Input is either blocking or non-blocking, and the event to be compared is non-blocking

   - 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.

Figure 1 gives a simplified checking flow in the Analyzer. A state vector must go through all checking before being updated. The error message and final successful communication sequences are the results of the Analyzer.



Simplified checking flow in the Analyzer

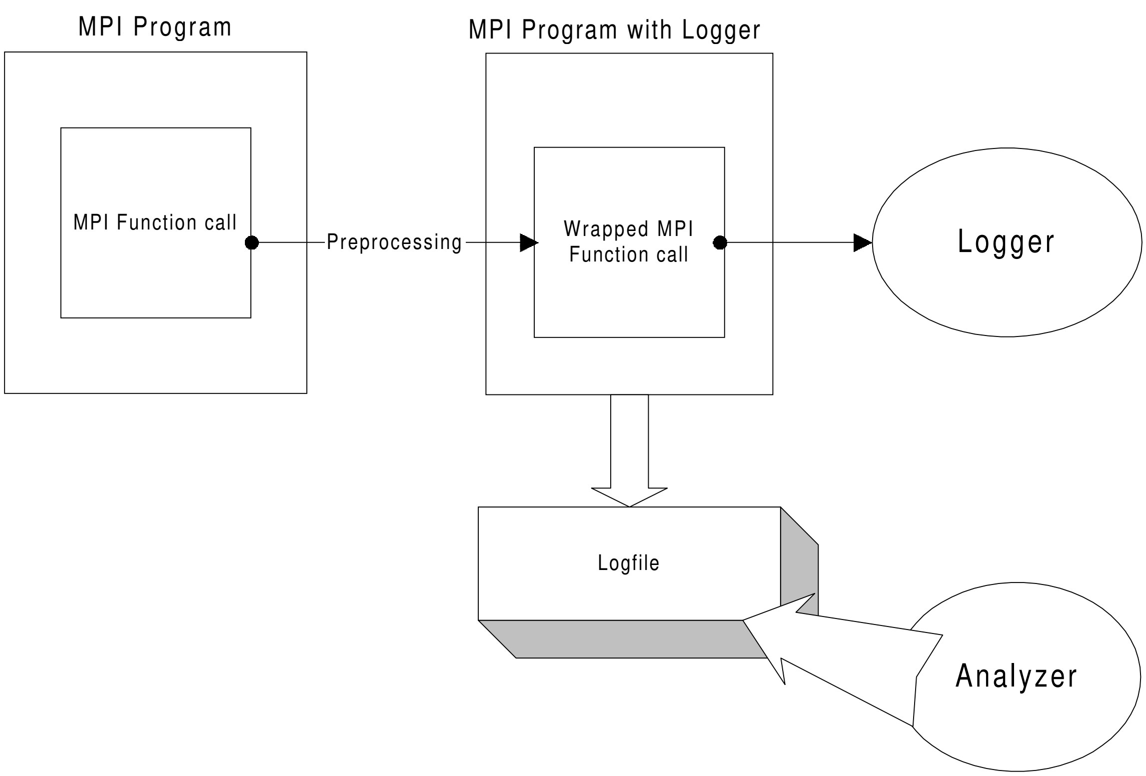
5. Add-on Preprocessor

The add-on preprocessor provides a blackbox to hide the event-logging mechanism. The user can activate the logger by simply launching this preprocessor, which searches the MPI keywords and replaces the MPI communication functions with the wrapper functions.

6. Summary

We have presented a tool, Automatic Logfile Analyzer, for checking communication consistency of parallel programs using MPI. Our Logfile Analyzer provides a simple and intuitive way of detecting communication problems in MPI programs. It is especially beneficial for beginners in parallel programming. This tool can be adapted to message-passing libraries other than MPI with minor changes. The algorithm used in the Analyzer can be easily changed and extended. Figure 2 depicts the structure of the tool and relation among the parts.

Although, at this time, our tool handles the default communicator MPI_COMM_WORLD, it can be extended to support communicator subdomains by adding some checking rules. The information needed for this purpose is already recorded by the logger object.



Firgure 2. The relation among the four parts. In preprocessing, the original MPI functions in user program are replaced by the wrapped MPI functions. Then the Wrapper calls Logger interface to log the communication events and produce logfiles. After logfiles are generated, the Analyzer processes the logfiles and returns the messages to users.

References

[1] MPI: A Message-Passing Interface Standard, Message Passing Interface Forum, June 12, 1995.

[2] Edward Karrels and Ewing Lusk. Performance analysis of MPI programs. In Jack J. Dongarra and Bernard Tourancheau, editors, Environments and Tools for Parallel Scientific Computing, pages 195--200. SIAM, 1994.