

The Distributed Application Debugger

Michael Quinn JONES¹ and Jan Bækgaard PEDERSEN

Department of Computer Science, University of Nevada, Las Vegas, NV, USA

Abstract In this paper we introduce a tool for debugging parallel programs which utilize the popular MPI (message passing interface) C library. The tool is called The Distributed Application Debugger and introduces distinct components around the code being debugged in order to play, record, and replay sessions of the user's code remotely. The GUI component incorporates the popular GDB debugger for stepping through code line by line as it runs on the distributed cluster as well as an analysis tool for stepping through the executed portions of code after the session has completed. While the system is composed of multiple components, the logistics of coordinating them is managed without user interaction; requiring the user only to provide the location of the program to debug before starting.

Keywords. distributed, debugger, debugging, parallel, MPI, TCP

Introduction

Anyone who has developed software before will likely agree that bugs will be encountered along the way. No matter how careful you are, logic errors, memory mismanagement, race conditions, inaccurate execution path assumptions and a whole host of other issues will be encountered from time to time. These issues are acceptable, understandable and expected when designing software. As a result of the popularity of sequential programs, and the common understanding that debugging will always be part of their development, many exceptional debugging resources have been created to help the developer analyze and step through sequential code.

Developing parallel programs which run on distributed computer clusters introduces additional challenges to those present in traditional sequential programs. When debugging parallel programs, one needs to be able to inspect both the sequential code executing on each node and track the flow of messages being passed back and forth between them in order to infer where a problem actually lies. One such framework for distributed programming is called MPI [1]. It stands for Message Passing Interface (MPI) and is a C library which utilizes the power of distributing work across processors which communicate with each other by passing messages.

The tool introduced in this paper, and described in greater detail in [2], known as The Distributed Application Debugger, was developed to debug code which runs in parallel within all versions of MPI including MPI-1, MPI-2, and MPI-3. Among other things, the tool gives the user a centralized and organized space to examine the output of each node while being able to also inspect and match the messages being passed between them. It provides them with a way to not only replay a recorded session but also to halt a session in order to step through each node's execution while inspecting all the variables of a system. Finally, because clusters of computers are often housed at universities or super computer centers around the world,

¹Corresponding Author: *Michael Quinn Jones, University of Nevada Las Vegas, 4505 Maryland Parkway, Las Vegas, NV, 89154, United States of America.* E-mail: mjones112000@gmail.com.

the tool is fundamentally structured to run remotely and behind any number of impeding computer servers that needed to be logged into first before being able to access the actual computer cluster.

1. Related Work and Motivation

The Distributed Application Debugger bases much of its foundation from two sets of related work. First was the results from two surveys given to two different sets of graduate students at the University of Nevada, Las Vegas over the course of two years. The purpose of the surveys was in response to research by Pancake [3] who suggests that tools for parallel programming and debugging are often found to be unhelpful for users because their developers do not spend enough time trying to understand what the root problems that their users really need addressed are. The surveys, reported in [4] and [5], attempted to classify bugs found by students learning to program with MPI into 5 distinct categories. The first four categories were based on the the *Partitioning* and *Communication* aspects of the parallel construction model known as PCAM [6].

Partitioning deals with the task of partitioning both the data and functionality of the algorithm being implemented. It can be further sub-categorized as *Data Decomposition*, which covers developing code structured to deal with managing memory, modeling data structures etc., and *Functional Decomposition* which deals with the organizational side of defining the responsibilities of each node and establishing roles for architectures such as pipelining and master/slave relationships. The second category, *Communication*, is the task of implementing interprocess communication. It can also be broken down into two sub-categories: *Messaging* which deals with correctly addressing individual send and receive calls between destination and source nodes, and *Protocol Specifications* which deals with the overall communication pattern.

The survey asked the students to report each time they encountered a bug while developing their projects for the semester. The bug report asked them to estimate the time they spent on it and to classify it as a *Data Decomposition*, *Functional Decomposition*, *Messaging*, *Protocol Specifications* or a fifth category, *Sequential*, type bug. In both surveys, the *Sequential* type, which deals with the traditional bugs found in sequential programs such as mistakes with conditionals, method calls, pointers, pre and post conditions, and algorithm modeling to name a few, made up the vast majority of bugs encountered while writing parallel programs as displayed in Figure 1.

The second motivational related work were previous tools presented by Tribou [7] and Basu [8]. These tools, built to debug PVM [9] and LAM-MPI [10] parallel programming libraries respectively, both focus on debugging from a multilevel approach as introduced in [11]. This approach focuses on allowing the user to first inspect the code from a *Sequential* level by organizing print statements together by issuing node, as shown in Figure 12, and allowing the user to attach GDB to their program as they would debug any sequential program. Secondly, users are able to inspect their code from a *Message* level which allows users to inspect data that was sent in a message and then compare it to the data that was received. Finally, the tool introduced a third level of debugging, the *Protocol* level which concerns itself with the messaging system as a whole by matching send and receive pairs and alerting the users when a message was not matched at all.

While the graduate student surveys and the tools introduced by Tribou and Basu influenced the layout and features included in the Distributed Application Debugger, the need for a new tool for debugging MPI programs became apparent when the results of the survey of graduate students showed that most of their debugging was done using print statements. Although commercial debuggers, described in Section 2, are available that can monitor dis-

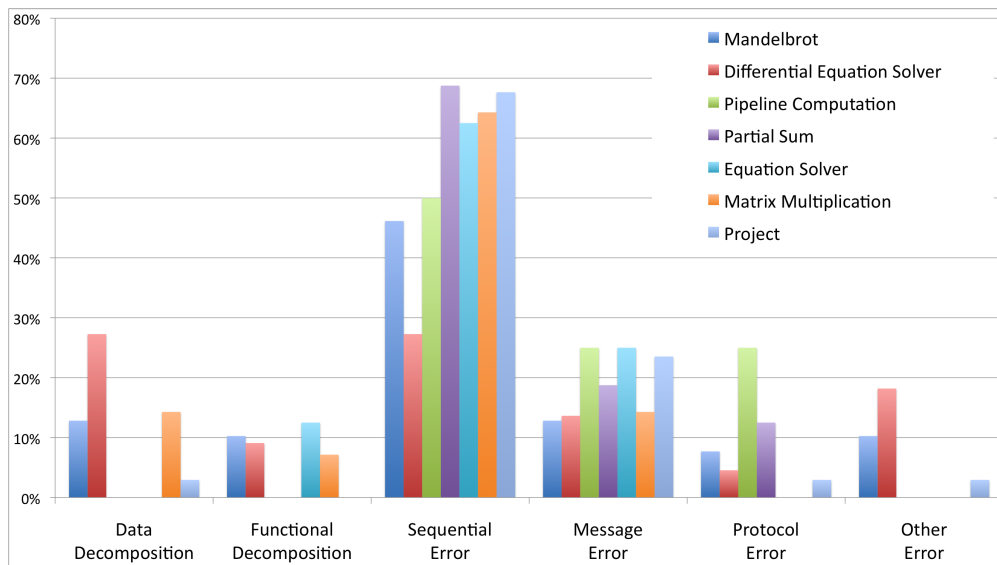


Figure 1. Error classifications from a survey [5] of graduate students taking a parallel programming class.

tributed processes at the petascale level, research by [12] found that 80 percent of developers used less than 4 processes when debugging their code. With this in mind we felt that a debugging tool focused on the common debugging needs of its target audience, rather than on extreme scalability, could still be very effective. The tool consists of debugging messages, detailed at great detail in [2], sent between the components detailed in Section 3. It includes features allowing the users to play, record, and replay their code running in a distributed system with a front end which allows them to analyze it in a sequential way.

2. Other Tools

The Open MPI Project's frequently asked questions website [13] characterizes how to debug applications running in parallel as a difficult question to answer. In their words

"Debugging in serial can be tricky: errors, uninitialized variables, stack smashing, ... etc. Debugging in parallel adds multiple different dimensions to this problem: a greater propensity for race conditions, asynchronous events, and the general difficulty of trying to understand N processes simultaneously executing – the problem becomes quite formidable."

The project recommends two enterprise level debuggers, *DDT* (short for the Distributed Debugging Tool) by Allinea Software [14] and *TotalView* by Rogue Wave Software [15], to aid in the complicated task of debugging MPI programs. Because of this endorsement, we felt that it was important to touch on some of their features and how they may be a better tool of choice at times than the Distributed Application Debugger.

DDT and *TotalView* have many features in common. Both GUIs are very *debugger-centric* in the way that the view is always focused on source code and the node specific view is simply the line of the source code currently being executed for any specific node. Because of this, other features, such as a graphical views of the messages being sent, are done with pop up windows. The Distributed Application Debugger took great lengths to not have views presented in pop ups so that the user is not tasked with juggling them. The Distributed Application Debugger by contrast, is more of an analysis tool which integrates a debugging feature when needed.

Both *DDT* and *TotalView* offer great features that allow inspection of the code at both the process level and the thread level. They display their message stacks in graphical form,

whereas the Distributed Application Debugger displays its messages in tabular form. Also, both not only are able to record and replay MPI sessions, but also allow the user to 'rewind' a session, whereas the Distributed Application Debugger always goes forward. Most impressively, they both can scale to over 100,000 processes which the Distributed Application Debugger would not be able to presently. This scalability comes with a large price, however, with DDT costing over \$600 for an academic or government workstation license, and TotalView costing in the range of \$1,000.00.

The Distributed Application Debugger's strengths lie in its simplicity. It can be configured to run remotely, and does not require any applications to be running on the remote machines prior to the session beginning. It copies, compiles and launches any component needed in the communication line outlined in Figure 6 as part of its start-up process. It also leverages the extremely popular and powerful GDB debugger with which most students are already familiar. This cuts down on its learning curve and further assists the students in focusing on their parallel programs as just a set of running sequential programs. The Distributed Application Debugger's layout presents a useful layout for viewing 2 or 3 nodes at a time because their data are displayed side by side without have to switch between them, while DDT and TotalView display the main code and require the user to request from which node they want to see data. The Distributed Application Debugger's weakness is in its scalability of processing messages across many processes. The processing time for a sample program which distributed 800 messages across 8 nodes each passing 100 messages, for instance, took only 3 seconds to process. The processing time for the same sample program distributed between 80 nodes each passing just 10 messages, however, took 37 seconds.

Figures 2 and 3 display the main front end GUI for DDT and TotalView respectively.

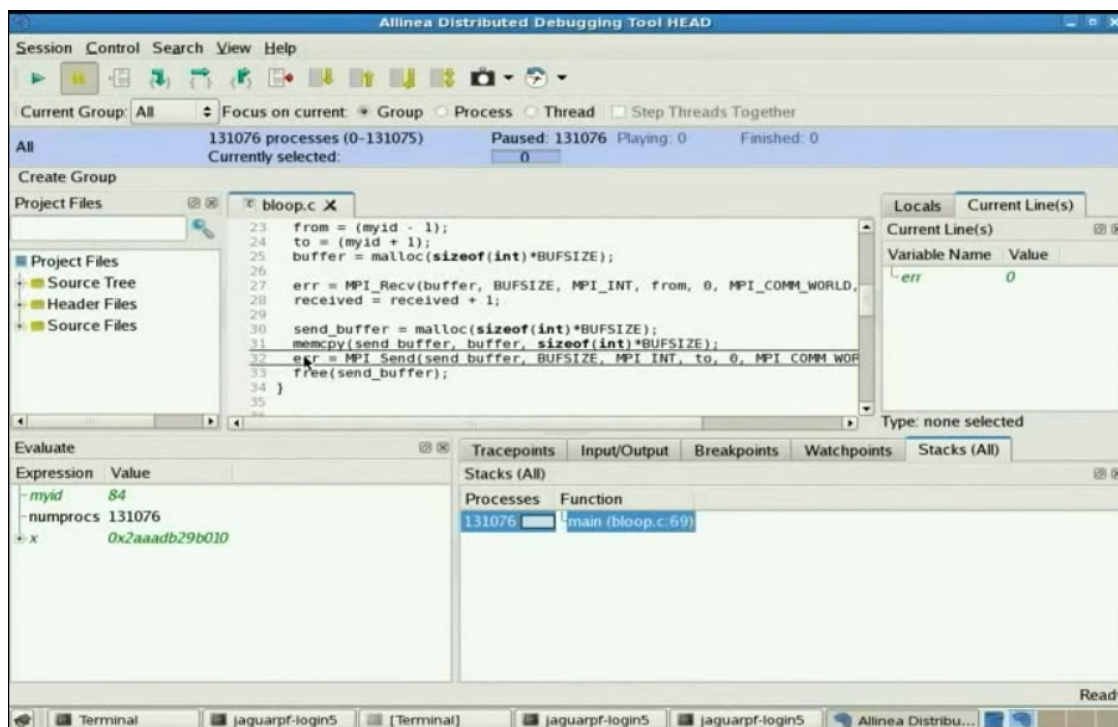


Figure 2. The GUI display of Allinea Software's DDT application.

3. Architecture

This section focuses on the architecture of the Distributed Application Debugger which is broken into 4 distinct components focused on giving the ability to instantiate and debug MPI

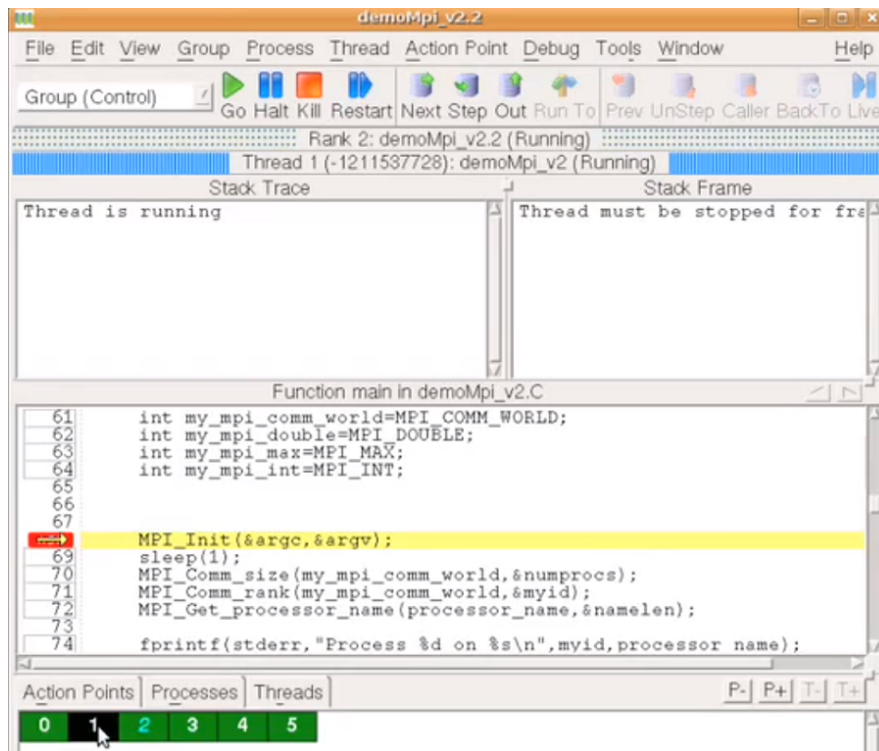


Figure 3. The GUI display of Rogue Wave Software's TotalView application.

programs remotely. The first of the components is a graphical user interface known as The Client. The Client, displayed in Figure 4, allows the users to input their remote credentials, instantiate a connection to a machine within a distributed cluster, and then initiate debugging sessions on their parallel programs until they choose to disconnect from the cluster. Each node of the distributed system conveys debugging data in its own panel which consists of a Console tab displaying output to that node's standard out, a Messages tab displaying all sent and received messages by that node, and an MPI tab which displays every MPI command run on that node along with its parameter values (as illustrated in Figure 5). More details regarding the debugging information presented in these panels are described in Section 5.

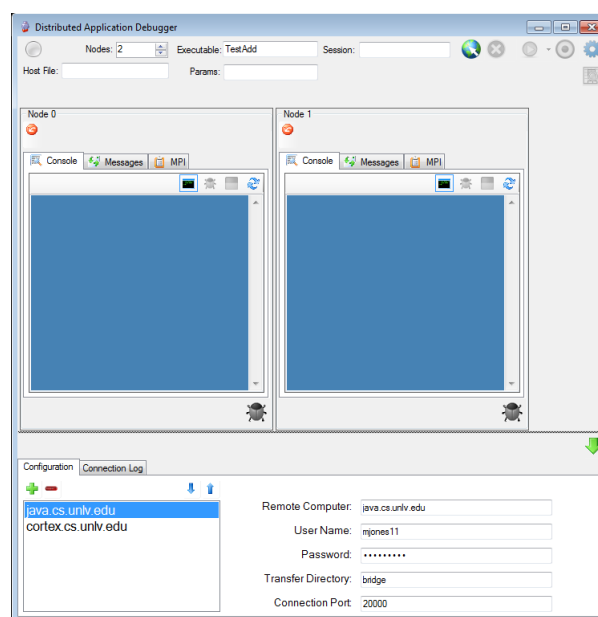


Figure 4. A debugging session configured for two nodes with both displayed.

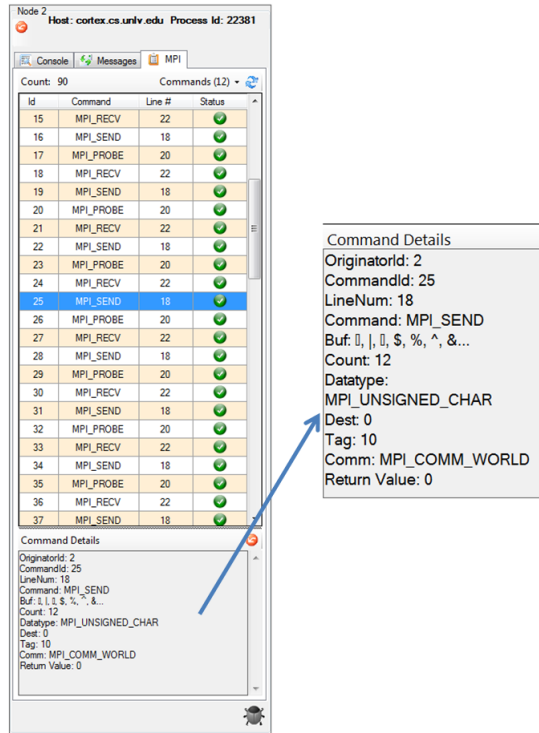


Figure 5. The extra information displayed in the Command Details panel of each node.

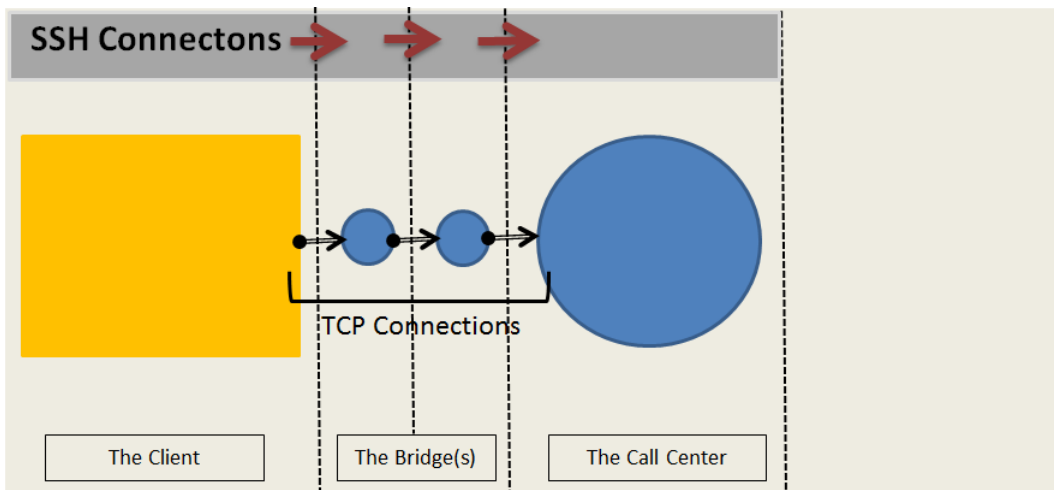


Figure 6. The system connected via TCP sockets.

The second and third components of the system are The Bridge and The Call Center which are depicted in Figure 6. The Bridge is a relay component meant to move data without inspecting it. The Bridge reads from one TCP port, the 'A' side, and writes to a second TCP port, the 'B' side. It also reads from the 'B' side and likewise writes what is read to the 'A' side. It is important to note that The Bridge is not necessary in the case that the parallel cluster is accessible directly from The Client but, in the case that the cluster is housed on a machine whose network is not accessible publicly, becomes crucial in connecting The Client to the cluster. The third component, The Call Center, however, is a crucial component in order to make debugging possible. The Call Center is a program which runs outside of the MPI system and must be run on a computer within the cluster. Its listens for commands from The Client in order to start a debugging session, and is responsible for starting and retrieving debugging sessions for The Client and relaying status information back to it. The connection from The Client to the remote Call Center is done in a two step process. First,

```

#include "mpi.h"

int main(int argc, char *argv[]){
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numProcs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myId);

  if(myId == 0){
    //This is the master node.
    //Send the whole buffer to the middle index
    MPI_Send(transferBuffer, numSlaves, MPI_INT,
             startingNodeId, numSlaves, MPI_COMM_WORLD);

    //Sync and then distribute the initial values
    MPI_Barrier(MPI_COMM_WORLD);

    //Wait for the result
    MPI_Recv(&finalResult,1, MPI_INT,
             numSlaves, TAG, MPI_COMM_WORLD, &stat);

    //Send result and wait for everyone to get it
    MPI_Send(&finalResult, 1, MPI_INT,
             startingNodeId, TAG, MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);
  }
  else{
    //Distribute and process the partial sums
    DistributeInitialValues();
    ProcessPartialSums();

    if(myId == numSlaves){
      //Send the result back to the master
      MPI_Send(&partialSum, 1, MPI_INT,
               0, TAG, MPI_COMM_WORLD);
    }
  }

  MPI_Finalize();
  return 0;
}

#include <mpi.h>
#include "debug.h"
#include "mpidebug.h"

int main(int argc, char *argv[]){
  _MPI_Init(&argc,&argv);
  _MPI_Comm_size(MPI_COMM_WORLD,&numProcs);
  _MPI_Comm_rank(MPI_COMM_WORLD,&myId);

  if(myId == 0){
    //This is the master node.
    //Send the whole buffer to the middle index
    _MPI_Send(transferBuffer, numSlaves, MPI_INT,
              startingNodeId, numSlaves, MPI_COMM_WORLD);

    //Sync and then distribute the initial values
    _MPI_Barrier(MPI_COMM_WORLD);

    //Wait for the result
    _MPI_Recv(&finalResult,1, MPI_INT,
              numSlaves, TAG, MPI_COMM_WORLD, &stat);

    //Send result and wait for everyone to get it
    _MPI_Send(&finalResult, 1, MPI_INT,
              startingNodeId, TAG, MPI_COMM_WORLD);

    _MPI_Barrier(MPI_COMM_WORLD);
  }
  else{
    //Distribute and process the partial sums
    DistributeInitialValues();
    ProcessPartialSums();

    if(myId == numSlaves){
      //Send the result back to the master
      _MPI_Send(&partialSum, 1, MPI_INT,
                0, TAG, MPI_COMM_WORLD);
    }
  }

  _MPI_Finalize();
  return 0;
}

```

Figure 7. Compile time changes made from including The Runtime's `mpi.h` file.

after the user has supplied credentials of which machines to connect to (as shown in the configuration section of Figure 4), The Client uses the order in which these credentials are entered to SSH into each destination machine in sequence, copy component libraries to the directory specified as the 'Transfer Directory' in the configuration section, compile them and then launch the appropriate component (Another Bridge or the Call Center). The last destination gets the Call Center launched on it and all of the other ones get a Bridge. After each of these nodes has either a Bridge or The Call Center running, The Client makes a TCP connection to the first node, which makes a connection to the second node etc., until The Client and The Call Center are connected via TCP either directly or through a sequence of Bridges as illustrated in Figure 6.

The last component of the architecture is The Runtime. The Runtime is a C library that redirects the user's MPI code to a set of intermediary methods which connect back to the Call Center and relay debugging information on behalf of the code being debugged. In order to incorporate The Runtime, the user needs to include a local MPI header file that contains macros which change the file's MPI calls at compile time from the true MPI functions to The Runtime's methods (This is illustrated in Figure 7). When the user starts a new MPI debugging session from The Client, The Bridges relay data to The Call Center, which launches an MPI session and then relays debugging data from The Runtime as displayed in Figure 8. The system's Bridge and Call Center applications also clean themselves up upon detecting that the program that they are reading from or the program that they are writing to has closed their socket connections. This reliably helps the system close in series when the user disconnects The Client from the session or if one of the components should crash.

It should be noted that the concept of each component cleaning itself up is one reason why The Bridge component was developed to relay messages which could have essentially been implemented by network administrators configuring port forwarding on their routers. When The Client closes its TCP connection, each Bridge detects that its input connection has closed and then properly closes their outgoing TCP connections in series until The Call Center detects that its incoming TCP connection has closed and can then properly clean itself up as well. If port forwarding has been used, when The Client closes its outgoing TCP

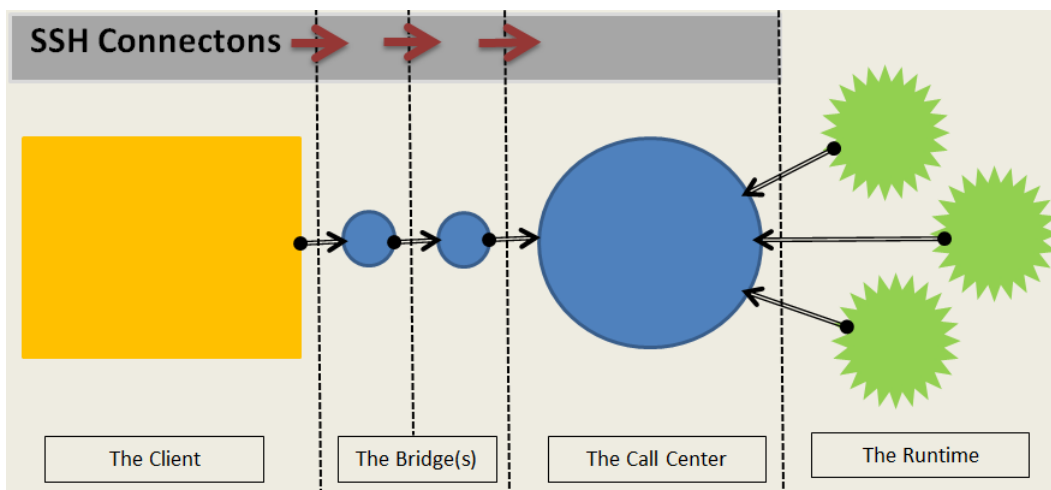


Figure 8. The system connected fully connected from The Client to the MPI nodes via The Call Center.

connection to an intermediary node, The Call Center would not get any more messages, but also would not know that it had been disconnected, and thus not know that the session is over and it should terminate itself.

4. Supported Commands

The Distributed Application Debugger does not support all of the 300 commands found within the MPI library [16], but does support 12 core commands making analyzing the initializing, message passing, synchronizing and finalizing of a typical MPI program possible. Table 1 lists the commands available for debugging within the Distributed Application Debugger along with their signatures.

Table 1. The MPI commands supported by the Distributed Application Debugger.

Command	Description
MPI_Init(int *argc, char ***argv)	Initializes the MPI environment
MPI_Comm_rank(MPI_Comm comm, int *rank)	Determines the rank of the process in the cluster
MPI_Comm_size(MPI_Comm comm, int *size)	Determines the size of the cluster
MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)	Performs a blocking send
MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)	Blocking receive for a message
MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)	Begins a nonblocking send
MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)	Begins a nonblocking receive
MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)	Blocking test for a message
MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)	Nonblocking test for a message
MPI_Wait(MPI_Request *request, MPI_Status *status)	Waits for an MPI request to complete
MPI_Barrier(MPI_Comm comm)	Blocks until all processes in the communicator have reached this routine.
int MPI_Finalize(void)	Terminates MPI execution environment.

5. Features

The Distributed Application Debugger introduces several features for inspecting data collected from The Runtime. First a (debugging) session is run in one of three modes: PLAY, RECORD, or REPLAY. PLAY is a standard MPI session in which The Runtime executes each line of the real MPI code and sends back the line number, though without the source file name, the parameters, and the result of each command run on each command.

Although one may think that PLAY mode would be the only mode necessary in order to run the debugger in order to inspect results, research by Leblanc and Mellor-Crummey [17] prompted us to add two other modes: RECORD and REPLAY. Leblanc and Mellor-Crummey illustrated that since parallel programs often include nodes passing messages asynchronously, that the execution behavior of a parallel program in response to a fixed input can be non-deterministic. Given this information we felt that it was crucial that the Distributed Application Debugger be able to record the execution of an MPI session and allow the user to both inspect it by hand and replay it by re-executing the code. During RECORD, the Call Center runs like it does in PLAY but also records an XML log file of all the data sent back to The Client during the session (illustrated in Figure 10). The data contains the name of the RECORD session, along with the number of nodes on which then program was executed, the host file used (which nodes on the cluster participated in the execution), the location of the executable, and the parameters passed to it. As seen in Figure 9, these recorded session are available to the user and can be recalled to run in the third and most useful debugging mode, REPLAY.

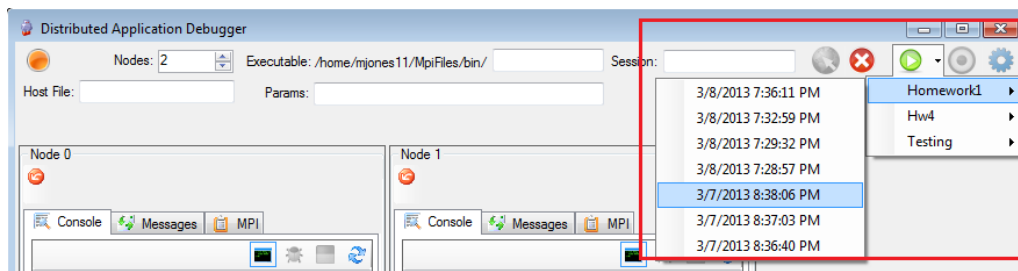


Figure 9. The names of the recorded sessions available for replay.

Nodes run in REPLAY mode do not execute MPI commands, but rather reload the results of each MPI command from the log files saved during a RECORD session. Running in REPLAY mode has several advantages. First, while in REPLAY mode, the user is able to playback the results of a session that has been recorded in exactly the same way as when it was executed (and recorded) before. This feature becomes especially helpful for users who are investigating exceptional cases which may be difficult to reproduce due to race conditions or other asynchronous factors. Secondly, the user has the ability to run a 'mixed mode' replay, in which designated nodes actually invoke the MPI framework (and send and receive actual messages) while the other nodes continue recalling their values from the log files. This allows the user to systematically narrow down which node or nodes may be causing the error being investigated. When a session is running in mixed mode replay, however, each node must consider if it will just relay back the results from the recorded log file, or if it must execute the actual MPI commands. As an example, the simple command `MPI_Comm_size` command does not affect any other nodes and thus does not have to ever be issued to the real MPI framework from a node running in REPLAY mode. `MPI_Recv`, however, must consider if the sender is replaying from a log file, or if it is a node that is invoking the real MPI framework. In the case that they are both replaying, the results from the `MPI_Recv` may just be read from the log file and sent back to The Call Center. If, however, the sender is actually issuing real MPI commands the receiving node must execute the `MPI_Recv` command so that the buffers of the MPI system do not fill up and cause the application to freeze. If a node running in REPLAY

```

<MPI_SIZE rank="0" commandId="3" dateTime="Mon Mar 04 08:55:15 2013 ">
  <parameters>
    <comm>1140850688</comm>
  </parameters>
  <returnvalue>0</returnvalue>
</MPI_SIZE>

<MPISEND rank="0" commandId="4" dateTime="Mon Mar 04 08:55:15 2013 ">
  <parameters>
    <buf>
      <value>0</value>
      <value>-2147483648</value>
      <value>2147483647</value>
      <value>356456</value>
      <value>765</value>
      <value>68378376</value>
      <value>67787</value>
      <value>17636</value>
      <value>585356</value>
      <value>253636</value>
    </buf>
    <count>10</count>
    <datatype>MPI_INT</datatype>
    <dest>1</dest>
    <tag>10</tag>
    <comm>1140850688</comm>
  </parameters>
  <returnvalue>0</returnvalue>
</MPISEND>

<MPI_RECV rank="0" commandId="5" dateTime="Mon Mar 04 08:55:15 2013 ">
  <parameters>
    <buf>
      <value>H</value>
      <value>e</value>
      <value>l</value>
      <value>l</value>
      <value>o</value>
      <value> </value>
      <value>W</value>
      <value>o</value>
      <value>r</value>
      <value>l</value>
      <value>d</value>
      <value></value>
    </buf>
    <count>12</count>
    <datatype>MPI_CHAR</datatype>
    <src>1</src>
    <tag>10</tag>
    <comm>1140850688</comm>
    <status>
      <MPI_SOURCE>1</MPI_SOURCE>
      <MPI_TAG>10</MPI_TAG>
      <MPI_ERROR>0</MPI_ERROR>
    </status>
  </parameters>
  <returnvalue>0</returnvalue>
</MPI_RECV>

```

Figure 10. A portion of the values recorded within a log file from a RECORD session.

Node 0
Host: cortex.cs.unlv.edu Process Id: 6476

Console Messages MPI

Count: 13 Commands (12)

Id	Command	Line #	Status
0	MPI_INIT	14	✓
1	MPI_RANK	15	✓
2	MPI_SIZE	16	✓
3	MPI_RECV	34	✓
4	MPI_RECV	34	✓
5	MPI_RECV	34	✓
6	MPI_RECV	34	✓
7	MPI_RECV	34	✓
8	MPI_RECV	34	✓
9	MPI_RECV	34	✓
10	MPI_RECV	34	✓
11	MPI_RECV	34	✓
12	MPI_RECV	34	?

Command Details

OriginatorId: 0
CommandId: 12
LineNum: 34
Command: MPI_RECV
Buf:
Count: 1
Datatype: MPI_INT
Src: 1
Tag: 0
Comm: MPI_COMM_WORLD


Figure 11. A message for which no POST has been received.

mode does issue a real MPI command, it will compare the results of the command to that of the values recorded in the log file: if a discrepancy is found, a warning will be relayed back to The Client, which will display a special symbol in the Status column of the MPI tab (see Figure 11).

Within each node's 'node panel', The Client organizes the debugging data sent back from The Runtime among three distinct tabs: the Console tab, the Messages tab, and the MPI tab. The Console tab displays a dedicated area for monitoring the standard out of each node. This allows the user to use print statements when needed. Such output is presented in the order which it was produced by the node. This is illustrated in Figure 12.

The Messages tab displays each message sent and received and matches sends and received as shown in Figure 13. When the user is in RECORD or REPLAY mode, the Messages tab also provides the option of recalling the buffer values which were sent or received in order to further investigate the message content during the debugging session.

The MPI tab, which displays every MPI command executed during the session, shows the order in which the MPI commands were executed while the code ran. An especially useful detail featured on this tab is that The Runtime reports each MPI command with the combination of a PRE message and a POST message. The PRE message indicates that an MPI command is about to be executed and what parameters were passed to the method. When the MPI command completes, The Runtime sends back a POST message which contains the results of the MPI command. The MPI tab displays an entry for each PRE message received. This informs the user about which MPI command is about to be executed, but the Status will be shown as 'Incomplete' until a corresponding POST message is received. This allows a user who is debugging a program which is not completing, to view the line number and parameters passed into the last command which started execution in order to investigate why it did not finish (no POST message was ever received, so the MPI command never finished).

The Distributed Application Debugger offers two different types of filters to cut down on the overwhelming amount of data that is present within each tab. The first type pertains to identifying mismatched messages as described above. The messages tab offers a filter button, , which removes all matched messages from the node's messages tab. (A matched message

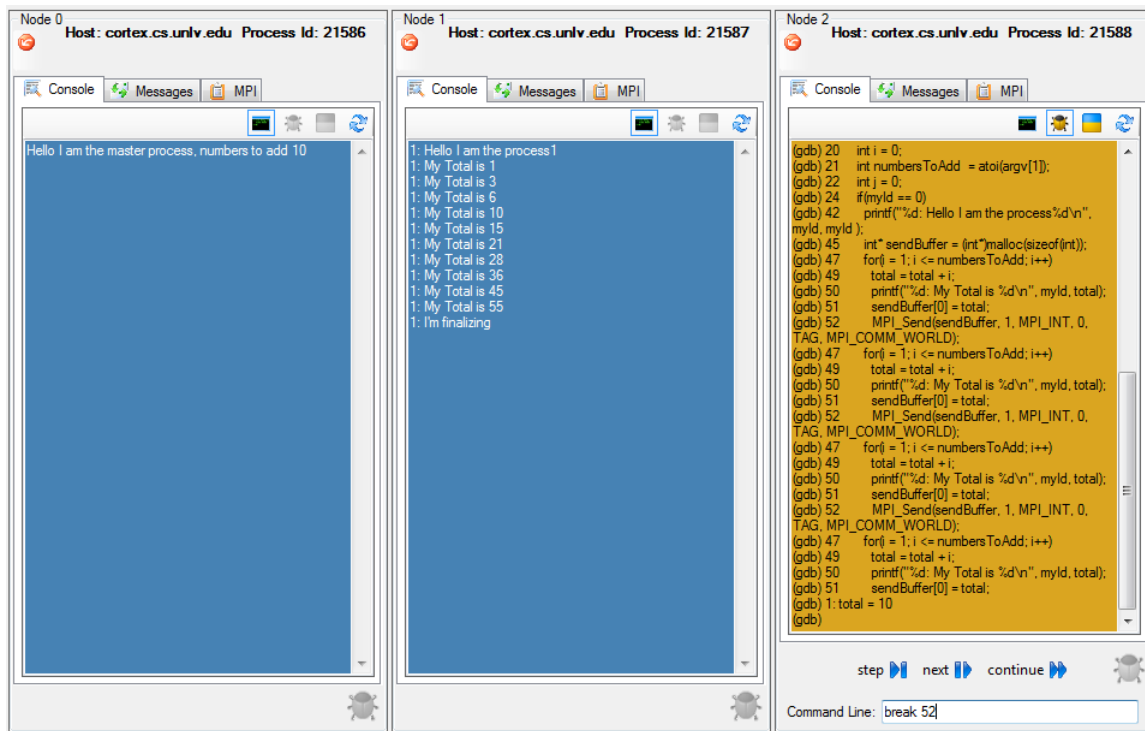


Figure 12. An MPI debugging session where GDB is attached to node 2.

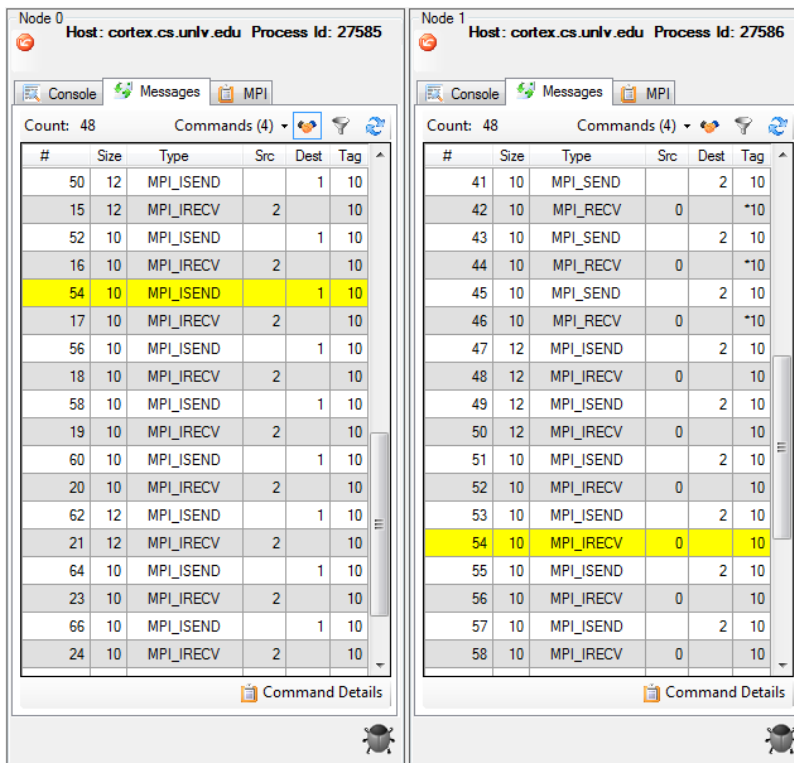


Figure 13. Two nodes displaying automated message matching.

is a message for which The Client has received POST messages from both the sender and receiver). Invoking this filter leaves the user with just the mismatched messages to investigate. The second type is aimed at helping the user cut down on the number of commands displayed within the Messages and MPI tabs. It is a drop-down which allows the users to check or uncheck the commands that they want displayed within the tab (see Figure 14).

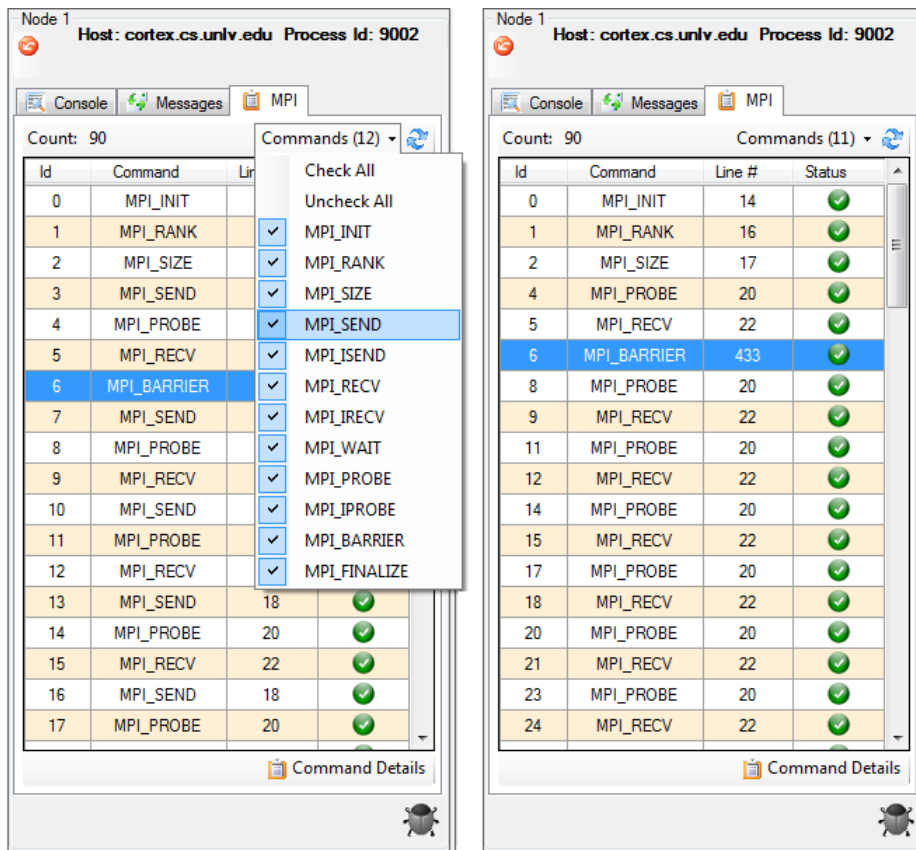


Figure 14. The MPI panel before and after a command filter was applied.

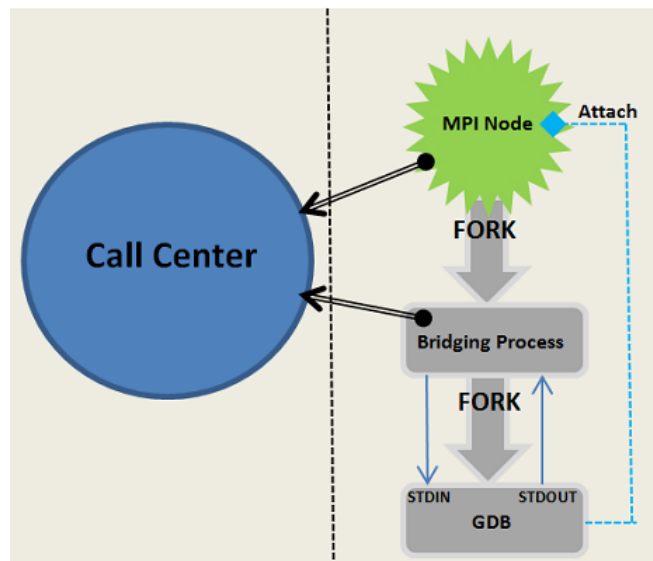


Figure 15. A Runtime node with an extra connection to the Call Center from GDB.

Finally, the Distributed Application Debugger incorporates the popular sequential command line debugging tool *GDB* [18]. The Client allows the user to attach *GDB* to any of the running nodes regardless of the mode in which the session is running. When the Call Center detects that a node is to be run with *GDB*, it instructs The Runtime to attach *GDB* to the selected nodes through a series of process forkings, and then waits for an additional connection back from the Bridging nodes controlling *GDB*. This is illustrated in Figure 15. The Client can then supply command line instructions to the Call Center which will route these commands to the Bridging Process controlling the *GDB* process. The Bridging Process then

writes the commands to the GDB's stdin (standard in). The Call Center routes, in addition to the standard data described, additional messages back from The Runtime. This additional data is the stdout (standard out) from all GDB processes (which contains other debugging data such as line numbers, variables' values and breakpoint information) which The Client can present to the user as displayed in Figure 12.

6. Conclusion

The Distributed Application Debugger has many features for debugging distributed MPI applications. It works remotely from home, even when the parallel cluster is not accessible directly. It graphically displays the nodes of a cluster in a way that represents both the sequential nature of the code being executing within each process, as well as the parallel nature of the messages being passed between them. It reliably handles all of the messages which are passed back within any given session to give valuable debugging data to the user about the MPI commands that are starting and completing as well as all of the data being printed to standard out. It offers features for recording and replaying MPI sessions which can help programmers focus on a problem that may be hard to recreate. It offers buffer value inspection to aid in debugging common sequential bugs along with message matching to cut down on the time for message error debugging. Finally it seamlessly integrates GDB to encourage alternatives to inefficient print statements.

7. Future Work

Some work should be done to extend the number of commands supported by the Distributed Application Debugger in the future. While the 12 commands it currently supports, presented in Table 1, are a start, MPI programs which implement commands outside of this initial scope will not get the full debugging features of the application at this time.

The next logical progression for the Distributed Application Debugger would be to integrate a development environment within it to compliment its debugging features. Integrated Development Environments (IDE), like Eclipse [19] or Microsoft Visual Studio [20] have become very popular with developers as a place that they feel comfortable both developing and debugging their programs in. In the current system, the user is expected to develop their code as they would normally do so, and to attach the debugger when they encounter bugs and need more data to resolve them. The system should offer a command line feature to take advantage of the persistent SSH session maintained with the cluster so that the user could add, update and delete file and directory names. In the future an interface with popular source control software such as Subversion [21] would be useful as well to let users index the various versions of their project files.

References

- [1] Jack Dongarra. MPI: A Message Passing Interface Standard. *The International Journal of Supercomputers and High Performance Computing*, 1994.
- [2] Michael Q. Jones. The Distributed Application Debugger. *M.Sc. Thesis, University of Nevada, Las Vegas, United States*, May 2013.
- [3] Cherri M. Pancake. Why Is There Such a Mis-Match between User Need and Parallel Tool Production? *Keynote address, 1993 Parallel Computing System: A Dialog between Users and Developers.*, April 1993.
- [4] Jan B. Pedersen. Classification of Programming Errors in Parallel Message Passing Systems. *In Proceedings of Communicating Process Architectures 2006 (CPA'06) IOS Press*, September 2006.
- [5] Jan B. Pedersen and Michael Q. Jones. Error Classifications for Parallel message Passing Programs: A Case Study. *Proceedings of Parallel and Distributed Processing techniques and Applications (PDPTA'12)*, July 2012.

- [6] Ian Foster. Designing and Building Parallel Programs: Concepts and tools for parallel software engineering. Addison Wesley, 1995.
- [7] Erik H. Tribou. Millipede: A Graphical Tool for Debugging Distributed Systems with a Multilevel Approach. *M.Sc. Thesis, University of Nevada, Las Vegas, United States*, August 2005.
- [8] Hoimonti Basu. Interactive Message Debugger for Parallel Message Passing Programs Using LAM-MPI. *M.Sc. Thesis, University of Nevada, Las Vegas, United States*, December 2005.
- [9] Parallel Virtual Machine, Oak Ridge National Laboratory. <http://www.csm.ornl.gov/pvm>, 2011.
- [10] LAM/MPI Parallel Computing. <http://www.lam-mpi.org>, 2012.
- [11] Jan B. Pedersen. Multilevel Debugging of Parallel Message Passing Systems. *PhD Thesis, University of British Columbia, Vancouver, British Columbia, Canada*, June 2003.
- [12] Susanne M. Balle and Robert T. Hood. Global Grid Forum User Program Development Tools Survey. *Global Grid Forum*, 2004.
- [13] The Open MPI Project. FAQ: Debugging applications in parallel. <http://www.open-mpi.org/faq/?category=debugging>, 2013.
- [14] Allinea Software. DDT product page. <http://www.allinea.com/products/ddt>, 2013.
- [15] Rogue Wave Software. TotalView product page. <http://www.roguewave.com/products/totalview.aspx>, 2013.
- [16] Mathematics Argonne National Laboratory and Computer Science Division. Web pages for MPI Routines. <http://www.mcs.anl.gov/research/projects/mpi/www/www3>, 2013.
- [17] Thomas J. Leblanc and John M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, April 1987.
- [18] GDB - The GNU Debugger. <http://www.gnu.org/directory/gdb.html>, 2013.
- [19] The Eclipse Foundation. <http://www.eclipse.org/>, 2013.
- [20] Microsoft Visual Studio. <http://msdn.microsoft.com/vstudio/>, 2013.
- [21] Apache Subversion. <http://subversion.apache.org>, 2013.

