IDLI: An Interactive Message Debugger for Parallel Programs using LAM-MPI

Hoimonti Basu
School of Computer Science
University of Nevada, Las Vegas
Las Vegas, NV 89154
Email: hoimonti@yahoo.com
Ph: 702-443-1538, Fax: 702-895-2639

Jan B. Pedersen
School of Computer Science
University of Nevada, Las Vegas
Las Vegas, NV 89154
Email: matt@cs.unlv.edu
Ph: 702-895-2557, Fax: 702-895-2639

Abstract

Many complex and computation intensive problems can be solved efficiently using parallel programs on a network of processors. One of the most widely used software platforms for such cluster computing is LAM-MPI. To aid development of robust parallel programs using LAM-MPI we need efficient debugging tools. However, the challenges in debugging parallel programs are unique and different from those of sequential programs. This paper introduces IDLI, a parallel message debugger for LAM-MPI, designed on the concepts of multilevel debugging. Through its customizable query mechanism, data abstraction, granularity, and user-friendly interface IDLI provides an effective environment for debugging parallel LAM-MPI programs. It has a novel technique to simultaneously replay and sequentially debug one or more processes from a distributed application.

Keywords: Distributed computing, LAM-MPI, multilevel debugging, message debugger

1. Introduction

Parallelism introduced in a computation brings new dimensions for errors and unexpected behavior. Consequently debugging parallel programs is a challenging job. The most often used technique is to insert print statements in the programs to track its behavior [1]. Often debuggers for sequential programming like the GNU Debugger (GDB) [2] or the Data Display Debugger (DDD) [3] are used to debug the sequential part of a parallel program at a particular node. But the limitation of these processes is that when a user has to debug a parallel application running on many nodes where both functionality and data has been distributed among various nodes. In such a scenario the user needs to understand the whole picture since the state of the entire application is dependent on the states of all the involved nodes. Focusing on debugging programs at particular nodes with a sequential debugger without understanding the big picture does not go a long way in finding and correcting intricate and complex bugs [4].

Another issue which makes debugging quite intricate is the fact that in parallel programs, the cause and effect of an error can be separated by great distance in time and code making it difficult to locate and debug. Also, this difficulty is enhanced when the cause and effect do not occur in the same process [5]. A parallel system is much larger and complex than a single process and many available tools deluge the programmer with information overload, making it nearly impossible to zero down on useful information pertinent to debugging [6]. Often, existing parallel debugging tools are criticized precisely for this reason [7]. IDLI has been designed to avoid information overloading by providing requisite abstractions and views. At the same time customized views can be generated by the user thus making the tool flexible. IDLI provides both local and global context debugging information thus making it suitable to be used for sequential debugging as well as understand the overall big picture. Our goal is to develop a simple, yet effective, debugger based on the principles of multilevel debugging [6] which expedites the process of locating errors due to faulty message passing in parallel programs using LAM-MPI [8].

2. Related Work

For the last two decades a great deal of research effort has been directed at developing tools for improving the development of parallel applications and significant progress has been made [9]. However, the reason for not having highly popular and standardized debuggers in the parallel domain, akin to sequential debuggers like GDB, is that they are extremely difficult to implement. Tool developers must cope with an inherently unstable environment where it may be impossible to reproduce program events or timing relationships [10]. Moreover, it is often difficult to find a comprehensive debugging tool capable of handling of all types of errors that arise in the parallel programming domain [8].

At present, quite a few tools are available for debugging MPI [11] programs. Most of them can be broadly classified into three categories based on the functionalities they provide. These categories are source level, graphical visualization and post processing debugging tools. Some of these tools with

their versatile features might belong to more than one of the above mentioned categories.

2.1. Source Level Debuggers

Source level debugging tools being, the lowest of the above mentioned three categories, are extensions of traditional sequential debuggers like GDB. Some tools simply instantiate a copy of a standard sequential debugger for each process, while others may be more sophisticated and have a sequential debugger integrated in an Integrated Development Environment (IDE). Each sequential debugging window is capable of providing substantial information about the attached process. The information is localized in context and is best used for debugging the sequential part of the code. Since these debuggers are based on the sequential style of programming they do not fit well into the paradigm of parallel programming which has quite a few new classes of errors related to message passing and protocol conformance [6].

Firstly, they typically operate at the level of source and assembly code. Such a fine level of granularity makes it difficult to debug an MPI program running on hundreds of nodes. Often source level debuggers do not have views or data pertaining to the big picture, containing all the nodes, which help a developer, analyze and locate the exact source of bugs. Though the technique of debugging a parallel MPI program usually starts at the local context, it eventually requires information pertaining to a global view of the whole application. Moreover, in an MPI program some of the most common type of errors arises due to faulty message passing involving several processes [4]. As the number of processes increase, it becomes nearly impossible for the developer to manually manage, issue commands, monitor the output, and control each process in a separate window. Further, such a process is quite prone to human errors. Examples of source level debuggers are Classic Guard [12], DDT [13], Etnus Total View [14], KDevelop [15], and PGDGB [16].

2.2. Graphical Visualization Debuggers

Graphical visualization debug tools attempt to assist the developer in a top-down debugging approach. They graphically present snapshots of the whole system indicating current states of processes, message queue, message route, pending messages and other relevant system features of the parallel machine. Their primary strength lies in depicting the complete system status at different points of time during execution of the program using various graphical charts and diagrams. They help developers in understanding the overall system behavior.

Unfortunately, these debuggers are at the other end of the spectrum compared to source level debuggers. Most of these debuggers lack sufficient granularity to aid a developer pin-point the location of the errors. Typically, in an MPI program the bulk of the code is sequential. Hence having no source level debugging capability at all seriously cripples the usability of these tools. The mapping of an activity at the global level to its causal context at the local level is left entirely to the user. Another limitation is that most graphical visualization debuggers typically have a predefined set of views. In other words, they are not flexible or adaptable to a user's customized needs. Examples of graphical visualization debuggers are Inter Trace Collector [17], MQM [18], Panorama [19], Paradyn [20], and XMPI [21].

2.3. Post Processing Debuggers

Post processing debugging tools provide post-mortem debugging capabilities. The principle behind such tools is logging program execution in sufficient detail which enables replaying a part or whole of the program later on. Most of these debuggers fail to provide sufficient granularity when needed. Generally they posses no integrated sequential debuggers and perform a replay based on past data stored in log files. As a result, on-the-fly data manipulation cannot be supported by these tools thereby seriously limiting their debugging capabilities. For example, a tool that was designed to record only message passing events would obviously lack any debugging capability for bugs in the sequential part of the code. Examples of post processing debuggers are Buster [22] and PVaniM [23].

2.4. Summary

Though there many tools available to aid in parallel programming only a handful of them are debuggers. Others belong to various tool classes like static or dynamic error checkers, profilers, event tracers and code analyzers. Primarily, the usability and effectiveness of available debuggers are severely reduced due to the following reasons:

- Most of them are designed and built to satisfy only one end of the spectrum for debugging tools, that is, they are either good at providing localized or global contexts but not both.
- A large amount of data which is often not of much relevance to the user makes debugging extremely challenging and time consuming by causing information overload.
- A serious shortcoming of existing tools is that they offer very little flexibility in creating customizable views or altering existing views. This often makes debugging extremely complicated or at worst impossible.
- In order to debug MPI programs effectively, users often need to trace a particular message or a group of related messages. Most existing debuggers do not support such queries. Few tools do offer tracing of messages but they require instrumentation of the users' applications.

Arguably, for debugging a parallel program, a user may choose to use a combination of currently available tools. But such a combination is often hindered by the following obstacles: (a) high learning curves for each tool, (b) lack of (seamless) integration since each product is from a different vendor, (c) different user interfaces and design philosophies for each tool, and (d) variances in compliance to standards, reliability, portability and levels of available support.

3. Multilevel Debugging

In contrast to the top down approach used in most parallel debuggers and visualization tools, multilevel debugging [6] was developed as a bottom up approach to debugging. Instead of providing a global view of a program and allowing the user to look for all errors using just one tool, the bottom up approach of multilevel debugging provides not only tools for the creation of error hypothesis but specialized tools for handling different error classes. These tools assist in verifying a hypothesis and refine it if necessary [6]. Also, such tools are equipped with the ability to help the user track the error back to its source code and fix it.

As discussed earlier, in contrast to sequential debugging there are many new types of errors that arise in parallel programming. To handle these new types of errors, new tools specific to each type which will provide detailed information on locating and debugging the error are needed. The bottom up approach of multilevel debugging is very well suited for development of these tools since it not only provides information for hypothesis of an error but also helps in locating the source of the bug. In multilevel debugging, errors are classified into three classes, namely sequential, message passing and protocol level errors. Each class has bugs that are specific to it. We shall be focusing on debugging the second class of errors, that is, message passing, using IDLI.

4. Introducing IDLI

The concept of multi level debugging was developed in [6] and demonstrated on PVM with a debugger named Millipede. Later, a Java GUI was added to Millipede to improve its cross platform compatibility and incorporate new debugging features [4, 24]. IDLI is a message debugger designed to extend the concepts of multilevel debugging to LAM MPI [8] which has become a popular message passing library for parallel computations. In the subsequent sections we shall explore the architecture and features of IDLI.

4.1. Architecture and Overview

Our multilevel message debugger, IDLI, operates with a high quality implementation of the MPI standard from LAM which is associated with Open Systems Laboratory (OSL) of Indiana University [8]. The architecture of IDLI consists of three layers: (a) a distributed relational SQL database which is used for persistent data storage comprises the back-end, (b) the middle layer is a native C [25] library which has wrappers for MPI functions and, (c) a simple shell user interface forms the front-end. These three layers aid in logging, displaying and analyzing the debugging information gathered from calls made to the MPI routines during the execution of a parallel application in a distributed environment.

4.1.1. Back-end: Distributed Relational Postgre SQL Database

We have used the free open source SQL database system PostgreSQL [26] for persistent storage. PostgreSQL is a well tested, widely used powerful and distributed object-relational database management system. It has an efficient and safe concurrency transaction management. The SQL database need not be installed on any of the nodes of the network where the user's application is executing. This is demonstrated in Figure 1. The architecture of IDLI enables processes to insert data into the database in a distributed manner. Each process executing a program with MPI calls, stores its data by opening a dedicated connection to the database server through a TCP/IP network connection. This data comprising of meta data from MPI calls and messages exchanged is used for debugging the program later on.

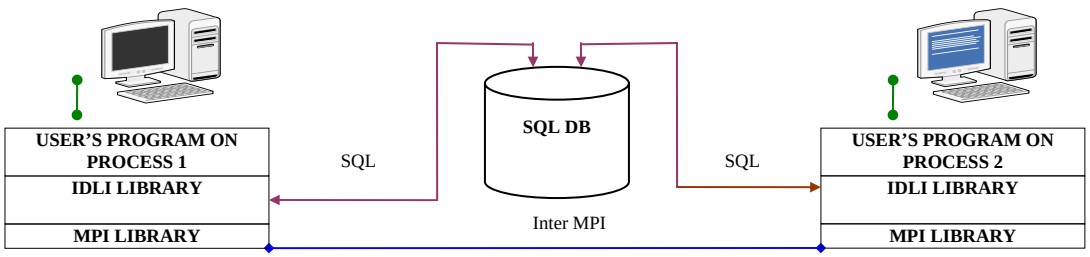

Figure 1: Overview of execution of an application in debug mode with IDLI.

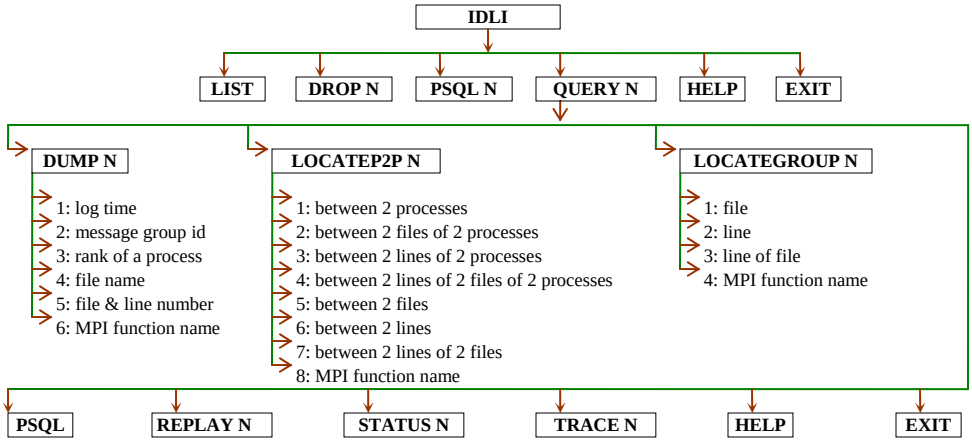

Figure 2: Menu Navigation Map of IDLI.

4.1.2. Middle Layer: Wrappers for MPI functions in the native C library of IDLI

During a debugging session, when a user's application is compiled in debug mode with IDLI, it is linked with the native C library of IDLI instead of the standard MPI library. This is done by means of functions present in IDLI's native C library known as wrappers. Currently, IDLI supports twenty four commonly used MPI functions [27]. Consequently the native C library has wrapper functions for each of these MPI routines. These wrapper functions intercept the MPI calls placed in the user's program. In addition, the wrappers possess intelligence for MPI function specific (a) initialization, (b) database processing, (c) lock management for database transactions like writes and updates (more than one process might try to simultaneously write or update the same database tables) and (d) determining the mode for processing, that is, whether the user wants to sequentially debug an application or replay the application at selected nodes. When a wrapper function intercepts an MPI call from an application program, it stores debugging information specific to the MPI routine in the SQL database at the back-end. The stored data is furnished by the wrappers whenever required by the user for analysis, and debugging of erroneous scenarios. As a result, IDLI's native C library has a two way communication with the SQL database as shown in Figure 1.

4.1.3. Front-end: Shell User Interface of IDLI

A simple shell user interface acts as IDLI's front-end. When a user begins a session with IDLI he gets a welcome screen with a list of menus. The complete menu navigation map of IDLI is shown in Figure 2. The front-end, which is a shell user-interface, possesses useful features like command history, command completion with tabs, prompts showing the selected user database for current session. Data is predominantly displayed as a set of rows. Each row has columns for different types of data. Each column has a header row which has a suitable name for that column. To easily discern the criteria that the data was sorted by, columns of sorted data are generally displayed in colors which are different from the rest of the data.

4.2. Features of IDLI

IDLI can be used to replay, analyze, as well as view, the contents of communication messages exchanged by MPI routines in an application. It can also be used for debugging source code of a program. Thus, IDLI enables a user to do *post-processing* as well as *source level* debugging or a combination of both.

4.2.1. Query Manager

IDLI, as a message debugger, can be used to view details of messages exchanged by MPI calls through a Query Manager. The Query Manager has a front-end, which is the shell user interface that interacts with a SQL database at the back-end. It is equipped with a feature that enables a user to write customized SQL queries. This helps a user to create customizable views of data from the global to local context of the application and vice versa. An example of the execution of a customized query is shown in Figure 3. A set of well defined built-in queries (as shown in Figure 2) are provided by the Query Manager to aid the user in retrieving debugging data for analysis and hypothesis formation of errors [6].

An example of the execution of the built-in query *locategroup N* is shown in Figure 4. The built-in features have been designed not to overwhelm the user with huge amounts of irrelevant data. The Query Manager also aids the user in tracing a message to its origin at a particular line number of the requisite file. In addition, a user can do post mortem analysis using IDLI's Query Manager.

```
IDLI=> psql 1
Welcome to psql 7.4.8, the PostgreSQL interactive terminal.

amma_1=> select
amma_1->        myrank, msggroupid, mpifuncname, mpifuncreturnmsg
amma_1-> from
amma_1->        loginfo, mpifuncsigid
amma_1-> where
amma_1->        opdone = 1 AND
amma_1->        myRank = 0 AND
amma_1->        loginfo. mpifuncid = mpifuncsigid.mpifuncid
amma_1-> order by
amma_1->        logtime;
 myrank | msggroupid |  mpifuncname  |    mpifuncreturnmsg
--------+------------+---------------+------------------------
      0 |          1 | MPI_Init      | MPI_SUCCESS: no errors
      0 |          3 | MPI_Comm_size | MPI_SUCCESS: no errors
      0 |          5 | MPI_Comm_rank | MPI_SUCCESS: no errors
      0 |          7 | MPI_Send      | MPI_SUCCESS: no errors
      0 |          8 | MPI_Finalize  | MPI_SUCCESS: no errors
(5 rows)
```

Figure 3: Example of execution of a customized SQL to get specific data.

```
amma_2=> locategroup 1
name of file? 1_TC_MPI_Bcast.c
msgId  src dest myRank mpiFuncName          ok fileName          line logTime                    returnMsg
   13    0    0      0 MPI_Bcast            1 1_TC_MPI_Bcast.c     28 2005-10-26 18:56:13.100651 MPI_SUCCESS: no errors
   13    0    1      1 MPI_Bcast            1 1_TC_MPI_Bcast.c     28 2005-10-26 18:56:13.107979 MPI_SUCCESS: no errors
   13    0    2      2 MPI_Bcast            1 1_TC_MPI_Bcast.c     28 2005-10-26 18:56:13.15367  MPI_SUCCESS: no errors
   13    0    3      3 MPI_Bcast            1 1_TC_MPI_Bcast.c     28 2005-10-26 18:56:13.18487  MPI_SUCCESS: no errors

amma_2=> locategroup 2
value of line? 28
msgId  src dest myRank mpiFuncName          ok fileName          line logTime                    returnMsg
   13    0    0      0 MPI_Bcast            1 1_TC_MPI_Bcast.c     28 2005-10-26 18:56:13.100651 MPI_SUCCESS: no errors
   13    0    1      1 MPI_Bcast            1 1_TC_MPI_Bcast.c     28 2005-10-26 18:56:13.107979 MPI_SUCCESS: no errors
   13    0    2      2 MPI_Bcast            1 1_TC_MPI_Bcast.c     28 2005-10-26 18:56:13.15367  MPI_SUCCESS: no errors
   13    0    3      3 MPI_Bcast            1 1_TC_MPI_Bcast.c     28 2005-10-26 18:56:13.18487  MPI_SUCCESS: no errors

amma_2=> locategroup 3
name of file? 1_TC_MPI_Bcast.c
value of line? 28
msgId  src dest myRank mpiFuncName          ok fileName          line logTime                    returnMsg
   13    0    0      0 MPI_Bcast            1 1_TC_MPI_Bcast.c     28 2005-10-26 18:56:13.100651 MPI_SUCCESS: no errors
   13    0    1      1 MPI_Bcast            1 1_TC_MPI_Bcast.c     28 2005-10-26 18:56:13.107979 MPI_SUCCESS: no errors
   13    0    2      2 MPI_Bcast            1 1_TC_MPI_Bcast.c     28 2005-10-26 18:56:13.15367  MPI_SUCCESS: no errors
   13    0    3      3 MPI_Bcast            1 1_TC_MPI_Bcast.c     28 2005-10-26 18:56:13.18487  MPI_SUCCESS: no errors

amma_2=> locategroup 4
name of MPI function? MPI_Bcast
msgId  src dest myRank mpiFuncName          ok fileName          line logTime                    returnMsg
   13    0    0      0 MPI_Bcast            1 1_TC_MPI_Bcast.c     28 2005-10-26 18:56:13.100651 MPI_SUCCESS: no errors
   13    0    1      1 MPI_Bcast            1 1_TC_MPI_Bcast.c     28 2005-10-26 18:56:13.107979 MPI_SUCCESS: no errors
   13    0    2      2 MPI_Bcast            1 1_TC_MPI_Bcast.c     28 2005-10-26 18:56:13.15367  MPI_SUCCESS: no errors
   13    0    3      3 MPI_Bcast            1 1_TC_MPI_Bcast.c     28 2005-10-26 18:56:13.18487  MPI_SUCCESS: no errors

amma_2=>
```

Figure 4: Example of execution of the built-in query locategroup N in IDLI's Query Manager.

These features of the Query Manager lend the qualities of adequate data abstraction and granularity to IDLI.

4.2.2. Replay

With IDLI's Replay feature, an application's execution can be replayed simultaneously at a number of selected processes using a sequential debugger of the user's choice. Figure 5 shows the overall architecture of IDLI's Query Manager and Replay at two nodes. During replay all data related to MPI calls are fetched from stored data of a previous run of the application.

Since the MPI communication layer is not invoked during Replay, the debugging process for a parallel application running on large networks is considerably faster. A user may run a Replay of the application in a sequential debugger of her choice for a selected set of processes from any machine on the

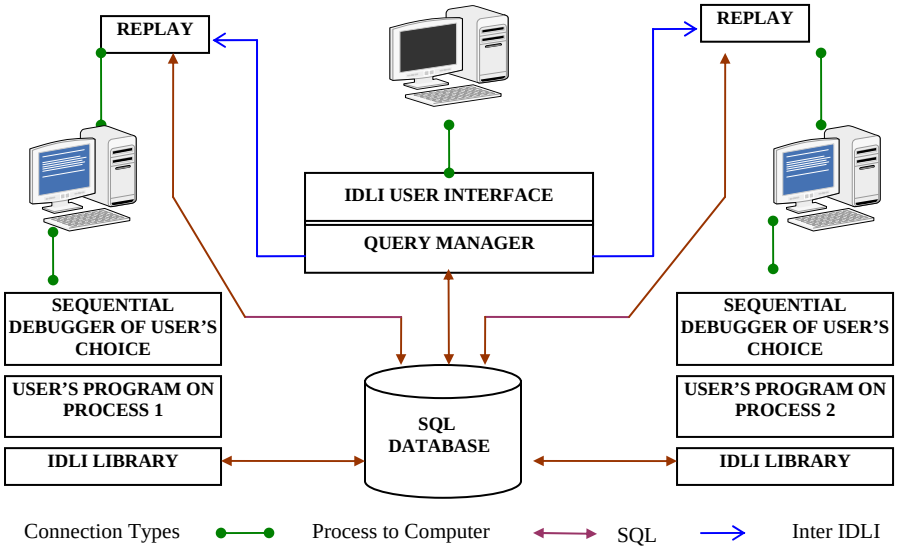

Figure 5: Overall architecture of IDLI's Query Manager and Replay.

network which is enabled with TCP/IP connectivity. Figure 6 shows Replay in action with the sequential debugger DDD.

IDLI creates individual connections for each specified process. During replay of a program, IDLI creates a connection to the chosen process running on a node using the Secure Shell 2 Protocol (SSH2) [28] over TCP/IP and invokes the user's selected sequential debugger. The original session of the Query Manager, from which the replay was initiated, is the parent while the replays on different nodes are its children.

The Replay feature can be used simultaneously for any number of processes of distinct ranks. IDLI automatically runs the Replay on the same node on which the process was originally run during an application's execution. We do so to simulate the exact hardware and software environment for the parallel application in which the errors arose. This is particularly relevant for processes running in a heterogeneous network where each node might have a different machine architecture and operating system.

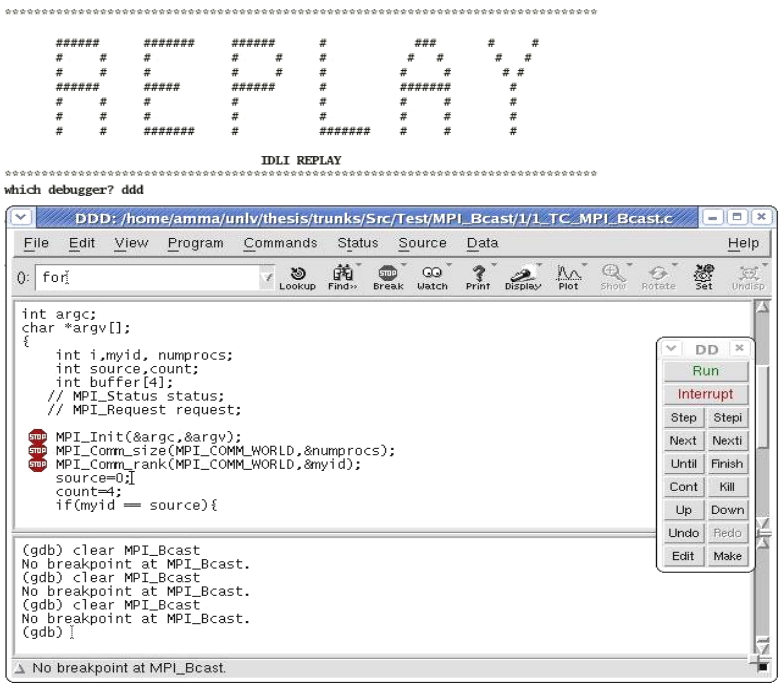

Figure 6: IDLI's Replay feature in action with sequential debugger DDD.

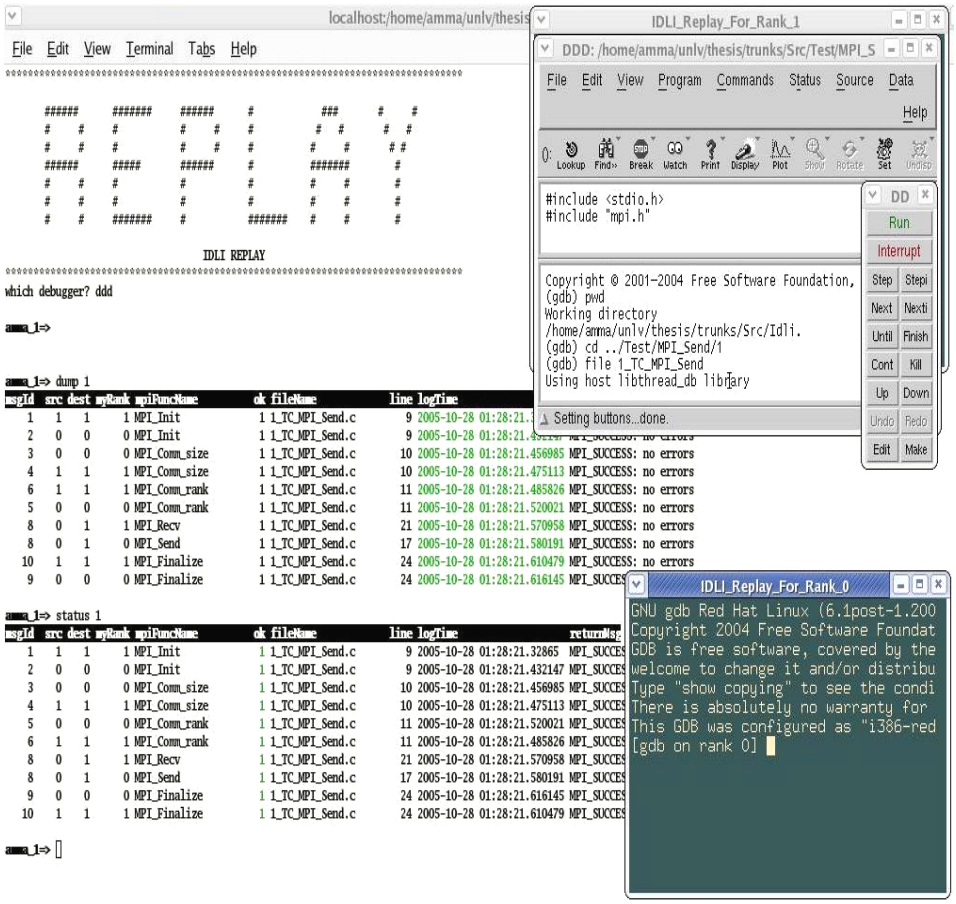

Figure 7: Replay on two different nodes with two different debuggers (DDD & GDB) along with simultaneous use of Query Manager.

It is noteworthy that multiple processes might have run on the same node. IDLI is capable of handling multiple such Replay sessions on the same node. Further it has built-in error checking to prevent a user from simultaneously invoking more than one Replay for the same process rank. IDLI has a robust Replay session management functionality. During exit from Replay, IDLI checks if there are any replays in action on any of the nodes. If that is the case, the user is notified to exit only when all nodes have quit replays.

An application can be replayed multiple times in the same session on any number of processes simultaneously. Also, a programmer can replay an application on different nodes of the network using different sequential debuggers simultaneously for each distinct process. This feature provides the flexibility to use GDB at node 1 or DDD at node 2 depending on the availability and need at a specific remote machine. The debugger is opened in an xterm window with display set to the local machine where IDLI is running. In addition, Replay and Query Manager can be run simultaneously. These features are demonstrated in Figure 7.

5. Conclusion

The guiding philosophy behind the development of IDLI was to implement a message debugger for the LAM-MPI environment based on the principles of multilevel debugging. This concept was designed to avoid several limitations prevalent in current parallel debugging tools. Some of the limitations prevalent in current parallel debuggers have been transcended in the following manner:

- Partial view of the debugging spectrum: IDLI provides both global and local context debugging information and is flexible to a user's specific needs.
- Information Overloading: Various levels of data abstraction are provided in IDLI thus enabling it to display relevant information according to a user's needs. The built-in queries have various options that help to trim down the debugging data and retrieve specific information. Moreover IDLI provides the flexibility to replay an application simultaneously on a chosen number of processes. This enables a user to choose as many processes as he is comfortable debugging simultaneously. Thus the user is in total

control of the amount of the data she wants to simultaneously view and process for debugging.

- Inability to alter or create custom views: A user has complete freedom to take advantage of the entire range of Postgres SQL commands to virtually create any desired view of the available data in the user database using IDLI.
- Lack of querying features at message level: IDLI's Query Manager has a whole set of built-in queries that cater to fine granularity at the message level. To provide detailed information on the exchanged messages IDLI does not require any modification or instrumentation of the user's application source code.

An interesting fact, that demonstrates IDLI's utility and convenience, is that it was used to debug its own software during the development cycle. In the implementation phase, considerable time and effort were saved while debugging a number of complex bugs, by using IDLI's Query Manager. To summarize, IDLI provides: (a) specific debugging information through sufficient levels of data abstraction, (b) connects global data with local context and vice-versa, (c) has a simple front-end user interface, (d) has built-in queries for querying messages and viewing details of executions of MPI routines, (e) allows custom SQL queries to be written by a user, and (f) enables fast and multiple simultaneous replays on any process (at its physical machine) with a sequential debuggers of the user's choice. We believe that these features based on multilevel debugging make IDLI a novel parallel debugging tool.

6. Future Work

We would like to enhance the features of our message debugger IDLI to include protocol conformation as defined in [5] and deadlock detection. Protocol conformation would allow a user to write specifications of the behavior of the protocol. Then using information from the actual messages, IDLI would automatically check that the messages satisfy the given specifications. Another great utility would be automatic detection of deadlocks based on an algorithm that provides automatic suggestions for a deadlock induced state, given a protocol specification [29]. Implementation of this algorithm would also add automatic correction to automatic detection of deadlocks.

At present IDLI is designed to work with message passing in one communication world denoted by the communication handle, MPI_COMM_WORLD. This functionality can be extended to multiple communication worlds which are used in many large real time applications for parallel computing.

Also, currently IDLI provides global level views of the whole system through data abstraction. This feature can be extended to include a graphical display of the entire system complete with pictures of active processes at various nodes, their executions and contents of messages exchanged [30]. We can also add profilers to view the performance of the system as a whole. These future features would transform IDLI into a complete debugger for LAM-MPI parallel programs.

7. References

[1] Pancake, Cherri. M. et al. *"Results of User Surveys Conducted on Behalf of Intel Supercomputer Systems Division, Two Divisions of IBM Corporation, and CONVEX Computer Corporation"*, 1989-1993.

[2] GNU DeBugger. http://www.gnu.org/directory/gdb.html

[3] Data Display Debugger. http://www.gnu.org/software/ddd/

[4] Tribou, Erik H. *"Millipede: A Graphical Tool for Debugging Distributed Systems with a Multilevel Approach"*, Masters Thesis, University of Nevada Las Vegas, Las Vegas, Nevada, USA, August 2005.

[5] Eisenstatdt, M. *"My Hairiest Bug War Stories", The Debugging Scandal And What To Do About It*, Communications of the ACM, April 1997.

[6] Pedersen, Jan B. *"Multilevel Debugging of Parallel Message Passing Systems"*, PhD Thesis, University of British Columbia, Vancouver, British Columbia, Canada, June 2003.

[7] Pancake, Cherri. M. *"Why Is There Such a Mismatch between User Need and Parallel Tool Production?"*, Keynote address, 1993 Workshop on Parallel Computing Systems: A Dialog between Users and Developers, April 1993.

[8] LAM / MPI Parallel Computing. http://www.lam-mpi.org/

[9] Pancake, Cherri. M. *"Performance Tools for Today's HPC: Are We Addressing the Right Issues?"* in Parallel Computing, Vol. 27, pp. 1403-1415, 2001.

[10] Pancake, Cherri. M. *"Applying Human Factors to the Design of Performance Tools"*, Proceedings of Euro-Par '99, pp. 440-457, 1999.

[11] Message Passing Interface (MPI). http://www.llnl.gov/computing/tutorials/mpi/

[12] Classic Guard. http://www.guardsoft.com/classicguard.html

[13] DDT. http://www.allinea.com/?page=48

[14] EtnusTotalView. http://www.etnus.com/TotalView/index.html

[15] KDevelop. http://freshmeat.net/projects/mpiplugin/

[16] PGDGB. http://www.pgroup.com/products/pgdbg.htm

[17] InterTraceCollector. http://www.intel.com/cd/ids/developer/asmona/eng/95656.htm

[18] MQM. http://web.engr.oregonstate.edu/~pancake/ptools/mqm/flyer.html

[19] Panorama. http://www-cse.ucsd.edu/users/berman/panorama.html

[20] Paradyn. http://www.cs.wisc.edu/~paradyn/

[21] XMPI. http://www.lam-mpi.org/software/xmpi/

[22] Buster.http://166.111.68.162/web/gelato/gelato-3-Buster.htm

[23] PVaniM. http://www.cc.gatech.edu/gvu/softviz/parviz/pvanimOL/pvanimOL.html

[24] Tribou, Erik H. & Pedersen, Jan B. *"Millipede: A Multilevel Debugging Environment for Distributed Systems"*, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'05), pp. 187-193, June 2005.

[25] Kernighan, B. & Ritchie, D. *"The C Programming Language"*, Prentice Hall, 1988.

[26] PostgreSQL. http://www.postgresql.org/

[27] Wilkinson, B. & Allen, M. *"Parallel Programming Techniques and Applications using Networked Workstations and Parallel Computers"*, 2nd Edition, Pearson Prentice Hall, 2005.

[28] SSH2. http://www.ssh.com/

[29] Pedersen, Jan B. & Wagner, A. *"Correcting Errors in Message Passing Systems"*, High-Level Parallel Programming Models and Supportive Environments, 6th international workshop, HIPS 2001, San Francisco, LNCS 2026, Springer Verlag, April 2001.

[30] Pancake, Cherri. M. "*Exploiting Visualization and Direct Manipulation to Make Parallel Tools More Communicative*," in Applied Parallel Computing, ed. B. Kagstrom et al., Springer Verlag, Berlin, , pp. 400-417, 1998.