Millipede: A Multilevel Debugging Environment for Distributed Systems

Erik Helge Tribou  
School of Computer Science  
University of Nevada, Las Vegas  
Las Vegas, NV 89154  
triboue@unlv.nevada.edu

Jan Bækgaard Pedersen  
School of Computer Science  
University of Nevada, Las Vegas  
Las Vegas, NV 89154  
matt@cs.unlv.edu

Abstract

The Millipede debugging environment is a realization of a multilevel debugging tool to quickly and easily correct the unique class of bugs which crop up in parallel and distributed systems. Through the use of the multilevel debugging approach, Millipede provides the abstractions, flexibility, and granularity to handle the unique and difficult challenges that arise in this field while avoiding several of the shortcomings of previous efforts.

Keywords: multilevel debugging, PVM, graphical user interface, distributed computing

1. Introduction

The need for debugging is present in any software development project. Programs have errors and bugs, and these need to be corrected. The approaches most used for debugging traditional sequential programs typically include the insertion of print statements, logging, and using debugging tools such as Gdb [1] and Purify [2] to view and monitor program state [3]. The already difficult task of debugging a program is exacerbated when that program exists within a distributed system where it must function and cooperate (often in an asynchronous manner) with other processes. Unlike a traditional sequential process, data and functionality must be distributed to the various processes in a system. This is accomplished by the passing of messages in adherence to an established protocol.

Research has shown that one important reason errors are hard to find is because the cause and effect are often separated by great distance in time as well as in code [4]. In a distributed system, the cause and effect of an error may not even be visible within a single process; that is, the effect of an error may only be visible in a process different than where the error originated (for example, a process may behave incorrectly due to erroneously computed values that were passed to it in a message). In addition, to view and debug the state of a distributed system, it is typically not sufficient to merely view the state of a single process; one must take into account the state of all participating processes. This often leads to a plethora of data which is difficult to manage and is often overwhelming to the programmer. Current tools in parallel debugging are often criticized for precisely this reason [5]. Millipede attempts to mitigate this information overload by the use of convenient abstractions and views while still allowing considerable flexibility through the use of a technique know as multilevel debugging. The goal being to allow the user to easily find the information necessary to complete the debugging task in a distributed system and continue with program development.

2. Previous Work

Years of effort have been put into developing parallel debugging tools, resulting in many excellent applications. Typically however, these tools are well suited for certain tasks or debugging a certain class of error, but poorly suited at others. They may overwhelm the user with irrelevant information, require excessive user intervention, or simply be unable to provide the information or functionality that is actually needed. We shall proceed by providing a brief overview of the available tools.

2.1. Extensions of Sequential Debuggers

A number of tools already exist to support parallel debugging. The simplest type are extensions of sequential debuggers. One may implement this as an integrated environment or may simply involve the use of N copies of a sequential tool such as Gdb—one for each process. The manual for the popular PVM parallel computing system itself discusses how to implement the N-copy approach to debugging [6, pp. 157-159]. The primary disadvantages of this approach include the overwhelming amount of information, the level of granularity cannot be varied (as it is often too fine), and the focus...
is always on the source code, leaving the user with the functionality of a sequential tool which may not be suited for the parallel debugging task. An N-copy system will spread the information over N windows and the information will not be available for queries. It takes an overwhelming amount of time for the programmer to extract, collect, and interpret the information available in a nontrivial system using this approach. Additionally, managing all the processes in a sequential manner is burdensome to the user, making it difficult to focus on a single task when needing to attend to the others. Examples of these types of debuggers include TotalView [7], pdbx [8], and p2d2 [9].

2.2. Visualization Tools

Another common class of tools are those that provide visualization of the program’s behavior. A typical tool offers a fixed set of views displaying information about the system in various ways, such as with graphs and charts. For example, a tool may provide a view detailing message passing to aid the programmer in locating stray messages or erroneous protocol violations. These tools generally suffer from the opposite problem of sequential debuggers, in that the granularity they provide for analysis is often too large; as they typically lack the ability to zoom in and provide low level information and map their views back to the actual source code. Such tools are well suited if the programmer is trying to obtain a global view of the entire system. However, global views are often too vast for a programmer to easily locate errors, this deficiency is typically exacerbated by the lack of capability to provide user defined views, providing only a limited subset of those the authors felt were most useful. Examples of visualization tools include Paradyn [10], Vampir [11], and XPVM [12].

2.3. Replay Tools

Finally, we will consider the class of tools that allow the user to replay the execution of program. Such tools collect and log information about the system as the program executes. Under typical usage, one will use such a tool offline in order to view the message content and program state of a process postmortem. Many of these systems have a set level of granularity, focusing on, for example, the source code level, leaving the user helpless if debugging at a higher level of abstraction is needed. Conversely, a system that logs higher level events would be unsuitable to solving a lower level problem. Other disadvantages of these tools include the massive level of information involved and the need to learn a new environment. While the replay mechanism is extremely useful, it is virtually impossible to perform a specific debugging task unless the tool specifically supports it. Examples of such systems include BUSTER [13], PVaniM [14], and PDT [15].

2.4. Summary of Related Work

We believe that many of the existing tools are hindered by the following major difficulties:

- Restrictive interfaces that support only a number of predefined tasks.
- The data needs to be interpreted by the user to map the error back to the cause. That is, the cause/ effect relationship is not well supported.
- Fixed or limited granularity, that combined with restrictive interfaces often makes debugging at an appropriate level of abstraction impossible or at best complicated.
- Information overload: the amount of information presented can be so large that the time needed to find the information needed becomes unmanageable.

Each of these systems have their strong points and are well suited for certain tasks. However, applying different tools for different tasks is difficult. Different user interfaces, representations, and formats makes changing between tools difficult, usually resulting in the user only choosing one or at best a small subset of the available tools that may not be optimal for the debugging task. We believe that Millipede provides a compelling alternative with its fresh approach to the problem through its usage of multilevel debugging.

3. Multilevel Debugging

Multilevel debugging is a novel bottom-up approach to debugging parallel message passing programs. The concept was developed in [16], and has been refined along with the development of the Millipede Graphical User Interface described in this paper. Multilevel debugging’s focus is closely tied to the major points of multidimensional analysis as described in [4]. The primary goal is to develop tools and techniques that make use of the extra information that can be extracted from a parallel message passing program as well as address some of the common shortcomings of existing tools as described in the previous section. This extra information includes the messages and protocol information.

3.1. The Three Levels

The multilevel debugging hierarchy consists of three levels: at the bottom, the sequential level, which is concerned with sequential errors as known from sequential programs. The second level is the message level. This level is concerned with messages, their content, and relations between sender and receiver. At this level we desire to provide message debugging (the ability to inspect and change the content of messages) and message queries (a database system that contains information about
the messages). Queries allow the user to determine relations between sender and receiver with any view or level of granularity they desire. Finally, at the third level, the protocol level, we are concerned with the overall communication protocol of the system. At this level we include tools like deadlock detection and correction as well as a protocol conformance checker, which checks the actual messages against a protocol specification.

4. Introducing Millipede

A proof-of-concept multilevel debugger was presented in [16]. We have expanded on this prototype by adding several features, improving usability and user efficiency through the addition of a GUI [17], and improved cross-platform compatibility through the use of Java.

Before proceeding, it is important to note that in pursuing the Millipede project we had the following three goals regarding understanding and implementation and wished to show they were manageable tasks:

1. Error classification. We wish to determine the various types of errors involved in parallel message passing programming and develop a methodology for efficient debugging of such programs.

2. Tool Development/Use. A number of errors not present in sequential program will arise. The multilevel debugging strategy suggests that specialized tools be developed towards the type of error they are expected to find. For example, use sequential debuggers to debug the sequential code and specialized algorithms to detect and correct deadlock or find protocol violations.

3. Automation. It is possible that some of the tools can be semi-automated to remove part of the burden of debugging from the programmer. By focusing on different error types in an isolated way, tasks that might have been intractable become tractable, and in some cases, it is possible to automate the debugging or correction process.

4.1. Architecture and Overview

Millipede currently operates with the PVM distributed system, and exists as two components: A debugging library that is linked to during compilation in place of the standard PVM libraries and a Java based GUI (henceforth referred to as the Millipede Debugger). The Millipede Debugger analyzes, logs, and displays the debugging information sent to it as a result of calls made into the debugging library by participating distributed processes.

The Millipede Debugger acts as a central manager for all spawned child processes in a debugging session. Information is transferred between the debugger and child processes via the already established communication infrastructure provided by the PVM runtime system. Communication is bidirectional, as processes not only have to inform the Millipede Debugger of their current state, but they must be controlled and informed appropriately in response to the user’s actions within the debugging environment (e.g. pause at breakpoints, perform an edit of received data, etc.). Millipede logs this information in order to provide a persistent record for later examination or playback. These observations do not affect the execution of a process beyond the alteration of timing (due to additional overhead and as a consequence of pausing at breakpoints). Timing differences should already be accounted for in any proper distributed system.

In addition to the logging performed by the Millipede Debugger, protocol and message information may optionally be persistently stored and managed within a SQL database. Our implementation uses the open source SQL database system PostgreSQL [18]. The insertion of data is done in a distributed manner where each PVM process inserts its own relevant data. PostgreSQL provides the efficiency, safety, and sanity to deal with these concurrent transactions (i.e., the ACID properties [19]) in a free, powerful, standards conforming package that many users already have the skills to exploit.

Programs that are written for use with PVM are typically written in C for which native libraries are provided with the PVM distribution. The Millipede Debugger (or any Java program) is able to interface with the PVM runtime system through the use of a library known as jPVM [20]. By linking the jPVM library with the standard PVM libraries, Java applications can access nearly all the features of the PVM runtime environment, including the ability to communicate with other PVM processes. Since the Millipede debugging libraries act in the place of the standard PVM libraries, when building jPVM, the user can link against the Millipede debugging libraries and gain the ability to debug a spawned Java process with Millipede. Java offers much potential towards greatly easing development and particularly deployment of applications in heterogeneous systems.

In true distributed fashion, the Millipede Debugger and the SQL database need not be run on the computers where the the child processes are spawned. The Millipede debugger need only connect to the PVM virtual machine and simple TCP/IP network connectivity is required between the child processes and the SQL database. Through the information logged by the Millipede Debugger and the SQL database, the user is able to analyze sessions both interactively and postmortem. He may even do so simultaneously; comparing the results of an interactive session to one that previously ended.
4.2. Multilevel Debugging Support Within Millipede

As discussed previously, what distinguishes a parallel process from a traditional sequential program is that it additionally sends messages to and receives messages from other processes while adhering to a communication protocol. In addition to debugging a process in a sequential manner, Millipede provides tools to debug at the convenient abstractions of the protocol and message levels, allowing the user to proceed at the appropriate level of abstraction for the debugging task at hand.

4.2.1. Sequential Level

Millipede’s main interface pane demonstrates most of the provided support for sequential level debugging. The pane is shown in Fig. 1. We note that in the figure, we are currently examining a process with the ID 262161 which is named Wave_master. The user may switch the view to one of the four other available processes by changing to the appropriate tab.

On the left side, we can see that a sequential log of activity is kept for each process. An enlarged view of this is shown in Fig. 2. This table keeps track of all PVM API calls. The table displays the name of the API call that was made, the source file and line where that call was made, the return code of that call (generally denoting success or failure), the input parameters, and information specifically regarding communication. The user may view the actual source code where the call was made, the man page for the call, an expanded view of the result of a call (e.g. data that was unpacked), and other information through the use of a contextual menu (also visible in Fig. 2). The calls are maintained in chronological order. Despite communication being performed over a network, log information being transferred via the PVM infrastructure maintains chronological order because the PVM runtime guarantees: “If task 1 sends message A to task 2, then task 1 sends message B to task 2, message A will arrive at task 2 before message B. Moreover, if both messages arrive before task 2 does a receive, then a wildcard receive will always return message A.” [6, p. 239] This ensures that log messages from a child process will arrive at, be read, and then processed by the Millipede Debugger in the same order they were sent (which will be the same order the PVM API calls were made).

In addition to the logged information, we maintain a connection to the standard IO streams of each child process. This functionality is visible on the bottom-left side of Fig. 1. This allows the user to easily supplement the debugging of a process with the familiar technique of using print statements. Each process’s output is maintained in its own pane, enabling the user to easily differentiate output from individual processes. Outside of a debugging context, this feature is useful for simply interacting with each process. In Fig. 1, the user interactively specified that he wished for four slaves to be spawned. This IO redirection is accomplished by replacing the standard IO file descriptors in a child with a TCP socket’s file descriptor that maintains a connection to the Millipede Debugger.

Additionally, if these tools are not sufficiently powerful to meet the user’s sequential debugging needs, Millipede allows the user to directly attach Gdb to an already running process. An interactive session of Gdb can be seen in Fig. 3. The addition of the Millipede library should not affect the debugging of a process with Gdb in any significant way beyond the need to advance a small number of times to get out of a Millipede library call if the user happens to attach to a process that it is currently inside one. Aside from this, the process merely needs to be compiled with the standard debugging option (-g in gcc) specified as normal.

Finally, by logging this information, the user can launch an individual process and use the log file as
a replacement to the PVM runtime. This allows the user to replay the execution of the process while using any sequential debugging tool he may desire.

We believe these tools and abstractions allow the user to effectively debug his program at the sequential level. The log file provides a convenient summary of his program’s current and past activity, the user may use the program’s standard IO streams to communicate in a familiar manner, and with more difficult problems he may attach Gdb to a process. By maintaining a log file, any process may be replayed and analyzed with any sequential tool.

<table>
<thead>
<tr>
<th>Type</th>
<th>File</th>
<th>Line</th>
<th>MsgNo</th>
<th>Item</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>pvm_upkdouble()</td>
<td>Wave_slave.c</td>
<td>93</td>
<td>163</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>pvm_inSend(PvmDataDefault)</td>
<td>Wave_slave.c</td>
<td>81</td>
<td>165</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>pvm_pkddouble()</td>
<td>Wave_slave.c</td>
<td>82</td>
<td>165</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>pvm_send(26154,22)</td>
<td>Wave_slave.c</td>
<td>83</td>
<td>165</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>pvm_rec(26154,11)</td>
<td>Wave_slave.c</td>
<td>92</td>
<td>167</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>pvm_upkdouble()</td>
<td>Wave_slave.c</td>
<td>93</td>
<td>167</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>pvm_inSendPvmDataDefault</td>
<td>Wave_slave.c</td>
<td>81</td>
<td>170</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>pvm_send(26154,22)</td>
<td>Wi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pvm_rec(26154,11)</td>
<td>Wi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pvm_pkddouble()</td>
<td>Wi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pvm_send(26154,22)</td>
<td>Wi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pvm_pick(26154,11)</td>
<td>Wi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pvm_upkdouble()</td>
<td>Wave_slave.c</td>
<td>93</td>
<td>170</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 2: The process log table

<table>
<thead>
<tr>
<th>Type</th>
<th>File</th>
<th>Line</th>
<th>MsgNo</th>
<th>Item</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>pvm_upkdouble()</td>
<td>Wave_slave.c</td>
<td>93</td>
<td>163</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>pvm_inSend(PvmDataDefault)</td>
<td>Wave_slave.c</td>
<td>81</td>
<td>165</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>pvm_pkddouble()</td>
<td>Wave_slave.c</td>
<td>82</td>
<td>165</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>pvm_send(26154,22)</td>
<td>Wave_slave.c</td>
<td>83</td>
<td>165</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>pvm_rec(26154,11)</td>
<td>Wave_slave.c</td>
<td>92</td>
<td>167</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>pvm_upkdouble()</td>
<td>Wave_slave.c</td>
<td>93</td>
<td>167</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>pvm_inSendPvmDataDefault</td>
<td>Wave_slave.c</td>
<td>81</td>
<td>170</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>pvm_send(26154,22)</td>
<td>Wi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pvm_rec(26154,11)</td>
<td>Wi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pvm_pkddouble()</td>
<td>Wi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pvm_send(26154,22)</td>
<td>Wi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: Using Gdb within Millipede

4.2.2. Message Level

As noted previously, the process log table displays information regarding communication. Each sent message is given a unique message number to identify it. The second through sixth lines of Fig. 2 show that this process packed a double value into a message and sent this message to process 26154 with message number 165. It later received a message identified as 167 from the same process, from which it unpacked a double value. One can view the order data was packed and unpacked in a message via the item column, which presents this sequence in successive order. If the user desired to see the corresponding receive for a send or unpack for a pack (or vice versa), he may use the Jump to Counterpart option in the contextual menu.

The table on the right hand side of Fig. 1, is a summary of communication information. An enlarged view is available in Fig. 4. The summary table allows the user to view the messages a process has sent and received along with what was packed or unpacked from those messages. From the process log table, one can jump to the corresponding summary entry to see how that communication operation fits in with others of the same type.

If the simplicity of the summary table is not sufficient to provide the information necessary to solve the debugging problem at the message level, then a query of the global and local state of the communication system may be performed through the SQL database. The interface to the SQL database is shown in Fig. 5. Entries are placed into the SQL database before the corresponding log entry is sent to the Millipede Debugger to ensure that everything displayed in the log is available through the SQL database.

Figure 4: Summary of logged information

The left side of the SQL pane contains a listing of available tables. The user can see every available field and its type. Queries can be entered in the upper right pane. Successful queries are stored in the history pane below it and a number of useful predefined queries are available at the bottom. Results of a query are displayed as a standard table as shown in the right hand floating window of Fig. 5. By providing access to the SQL database, we enable the user to create his own views of message level data at any level of granularity he may desire.

Another feature that aids in message level debugging is the ability to set breakpoints at PVM API calls. The interface to accomplish this is visible in the lower right corner of Fig. 1. One may instruct a process to stop at the next PVM API call (stop), to continue along until specific API calls are made (play), or to step over a currently waiting API call and continue (step). If a process has stopped at a call that unpacks data, one may view and edit the data the process has received. Viewing
the unpacked data is also supported in the log table for calls that have completed previously. This control is managed by sending messages through the PVM infrastructure. Every time a PVM API call is made, a message describing the call type is passed to the Millipede Debugger. Upon receipt of a response, execution continues, unless the current call unpacks data, which may result in the data being transferred to the Millipede Debugger and a reply message containing any edited data. Assuming the call did not unpack data, this exchange will only consume one byte each way, meaning that for this communication we essentially only pay the cost of latency. We implemented this control flow scheme in the way described in order to ensure that all updates are visible immediately. As soon the debugger is instructed to have the process stop at a particular API call, the response will be instantaneous. We do not have to perform a potentially costly, delayed, or otherwise unwieldy distribution of updated debugging information to the affected processes.

4.2.3. Protocol Level

Analysis at the protocol level is done primarily within the Protocol Conformance pane. This pane is visible in Fig. 6. One specifies a protocol using the language developed in [16]. A glimpse of this language can be viewed in the left half of Fig. 6. This language allows one to specify rules concerning how sends and their corresponding receives should match. The protocol processes information at both the communication and source code levels. A source file may have its communication calls marked with comments such as /*PROTOCOL (send_left)*/ or /*PROTOCOL (receive_right)*/, allowing the line numbers of those calls to be referred to by name. Protocol rules specify which processes may send to which (for example, a rule may specify that a slave process may only communicate with its immediate left and right neighbors), where a message sent in one proc-
ess should be received in another (for example, if a slave sends a message to its left neighbor, its left neighbor should receive it at a line marked ‘receive_right’), as well as what PVM message tags a message should be sent and received as.

When the Millipede Debugger logs a send and a corresponding receive (we are able to determine a match through the use of message numbers), it compares them against the specified protocol. At its strictest sensitivity, if no rules, or more than one rule of the protocol is matched, the pair is flagged as violating the protocol and their information is put in the table at the lower right of Fig. 6. A protocol violation signals that either the user’s understanding of how the system should work is flawed or the system itself is flawed, either of which should be corrected. Messages that have not been received are queued in the table at the upper right of Fig. 6. This allows the user to easily see potentially stray messages, or realize that something is wrong with a process (such as being involved in a deadlock), such that it cannot receive its pending messages. Further information about the individual messages composing a protocol violation can be examined at a lower level from the SQL database and log tables.

The protocol pane allows the user to analyze the behavior of their distributed program at a very high level. The user specifies how the individual processes may communicate with each other and Millipede provides the tools to ensure that his processes are behaving as they should. As information about the system is gained, the protocol may be refined to a complete specification of process communication.

5. Conclusion

Through the use of the multilevel debugging technique, we believe that Millipede is a unique and novel tool in the field of debugging parallel distributed programs. Millipede takes into account the distinctive properties of distributed programming by providing the tools to analyze a distributed system at the sequential, message, and protocol levels in a unified package. The Millipede Debugger presents the user with convenient abstractions that avoid overwhelming him with information, yet is flexible enough to provide a fine level of granularity and creation of user defined views, allowing him to quickly discern the information necessary to complete the debugging task.

6. Future Work

We would like to expand the capabilities of Millipede and the multilevel debugging technique to other distributed systems beyond PVM, most notably to the popular MPI system. Currently, we are working on implementing Millipede for LAM/MPI [21].

We desire to provide additional tools at the protocol level, specifically support for detecting and correcting deadlock. We have developed an algorithm described in [22] that will detect when deadlock has occurred and will suggest a possible correction. This algorithm merely needs to be incorporated into the Millipede system.

Additional work may also be applied to linking the Millipede libraries with jPVM. Millipede’s libraries function with jPVM, but line and file numbers of PVM API calls are produced by the calls made within the jPVM native library, instead of within the actual Java source. This hampers analysis, especially at the protocol level. We have a technique for retrieving the correct information that has been successfully applied to a few critical functions. This technique merely needs to be applied to the multitude of remaining PVM functions. With this accomplished, one can fully exploit Millipede in the debugging of distributed Java applications.

7. References