MILLIPEDE: A GRAPHICAL TOOL FOR DEBUGGING DISTRIBUTED SYSTEMS WITH A MULTILEVEL APPROACH

by

Erik Helge Tribou

Bachelor of Science
University of Washington
2002

A thesis submitted in partial fulfillment of the requirements for the

Master of Science Degree in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
August 2005
Placeholder page for approval
ABSTRACT

Millipede: A Graphical Tool for Debugging Distributed Systems with a Multilevel Approach

by

Erik Helge Tribou

Dr. Jan B. Pedersen, Examination Committee Chair
Professor of Computer Science
University of Nevada, Las Vegas

Much research and development has been applied to the problem of debugging computer programs. Unfortunately, most of this effort has been applied to solving the problem for traditional sequential programs with little attention paid to the parallel and distributed domains. Tracking down and fixing bugs in a parallel or distributed environment presents unique challenges for which these traditional sequential tools are simply not adequate. This thesis describes the development and usage of the Millipede debugging system, a graphical tool that applies the novel technique of multilevel debugging to the distributed debugging problem. By providing a user interface that offers the abstractions, flexibility, and granularity to handle the unique challenges that arise in this field, Millipede presents the user with an effective and compelling environment for the debugging of parallel and distributed programs, while avoiding many of the pitfalls encountered by its predecessors.
TABLE OF CONTENTS

ABSTRACT..............................................................................................................................iii

LIST OF FIGURES..................................................................................................................vi

CHAPTER 1 INTRODUCTION....................................................................................................1
An Introduction to Parallel and Distributed Systems..................................................1
Why do we need Debuggers for Parallel and Distributed Systems?..............4
Objectives and Goals of this Thesis.................................................................6
Organization of this Thesis..............................................................................7

CHAPTER 2 PREVIOUS WORK...............................................................................................8
Extensions of Sequential Debugging Tools..................................................8
Visualization Tools.......................................................................................11
Logging and Replay Tools..........................................................................14
Summary and Conclusions Regarding Previous Work..........................15

CHAPTER 3 THE PVM DISTRIBUTED SYSTEM................................................................18
Sending and Receiving Messages..........................................................18
Packing and Unpacking Data.................................................................20

CHAPTER 4 MULTILEVEL DEBUGGING.............................................................................23
Sequential Level Errors.............................................................................24
Message Level Errors.............................................................................25
Protocol Level Errors..............................................................................27

CHAPTER 5 INTRODUCING MILLIPEDE...........................................................................29
Architecture and Overview........................................................................29
Sequential Level Debugging....................................................................32
Message Level Debugging.......................................................................40
Protocol Level Debugging........................................................................52
Other Tools...............................................................................................59
Summary of Tools.....................................................................................60

CHAPTER 6 THE IMPLEMENTATION OF MILLIPEDE..........................................................63
The Native Library.....................................................................................63
The Millipede Debugger............................................................................70

CHAPTER 7 CONCLUSION AND FUTURE WORK...............................................................81
Improvements upon Previous Tools.........................................................81
Future Work............................................................................................83
LIST OF FIGURES

Figure 1  Debugging with PVM’s built-in support ........................................ 10
Figure 2  The XPVM system ........................................................................ 11
Figure 3  Organization of the Millipede system .......................................... 31
Figure 4  Millipede’s main interface pane ..................................................... 33
Figure 5  The process log table ...................................................................... 34
Figure 6  Viewing the source code ................................................................. 37
Figure 7  Displaying the PVM man page ......................................................... 38
Figure 8  Displaying the error message for a failed call .................................. 38
Figure 9  Displaying the result of an unpack call .......................................... 39
Figure 10 Using Gdb within Millipede ............................................................ 39
Figure 11 Replaying \texttt{Wave\_master} with the log file ............................... 40
Figure 12 The summary table ....................................................................... 41
Figure 13 The SQL pane .............................................................................. 43
Figure 14 The \texttt{match} query ................................................................... 45
Figure 15 The \texttt{locate} query ................................................................... 45
Figure 16 The \texttt{status} query ................................................................... 47
Figure 17 The \texttt{dump} query ................................................................... 48
Figure 18 The \texttt{unmatched\_sends} query .................................................. 48
Figure 19 The \texttt{unmatched\_pk\_upks} query .............................................. 49
Figure 20 Editing unpacked values ................................................................. 51
Figure 21 Setting PVM breakpoints ............................................................... 51
Figure 22 The protocol pane ....................................................................... 52
Figure 23 Protocol rules for \texttt{Wave\_slave} and \texttt{Wave\_master} ............... 54
Figure 24 Matching a violating send and receive ......................................... 55
Figure 25 The process metadata for \texttt{Wave\_master} ................................... 61
Figure 26 The processes menu ..................................................................... 61
Figure 27 Using the PVM console ................................................................. 62
Figure 28 The Python script to run a process from a log file ....................... 70
Figure 29 The Millipede Debugger’s thread structure ..................................... 72
Figure 30 The \texttt{match} query implementation .......................................... 78
CHAPTER 1

INTRODUCTION

Over the past few decades, the demand for computational power in certain fields has increased faster than even what the prodigious rate of CPU improvement can provide. In the scientific field, many are finding that as the granularity and effectiveness of data gathering tools increases, they are now faced with the quandary that the data collected from an experiment or observation may now simply be too large or complex to effectively analyze by hand, or even a high-end computer. A famous example of this problem was experienced by SETI [SETI], a research effort to discover evidence of life outside Earth by examining radio (and other) signals from stars outside our solar system. Unfortunately, the considerable number of known stars and the signals coming from them make a thorough analysis impossible for a single person or computer. One popular solution to this computational problem (and the one that SETI implements on a massive scale) is to create multiple cooperating processes spread over multiple CPUs or computing systems. These processes work in parallel towards the computation of a larger goal (e.g., finding signs of intelligent life in a radio signal). With a larger pool of resources than a single process executing on a single CPU, such systems enable more encompassing, timely, and accurate analysis of available data.
An Introduction to Parallel and Distributed Systems

While the promise of the computing power that parallel systems can provide is great, the architecture of these systems is fundamentally different from the architecture found in traditional uniprocessor systems. These differences were detailed by Michael Flynn, who in 1972 coined a taxonomy for architectures that differ in the number of concurrent instruction and data streams they could execute [FLYN]. In Flynn’s taxonomy, a traditional uniprocessor system is classified as single instruction, single data (SISD), as they fail to exploit any parallelism in their instruction or data streams; that is, they work with a single instruction and piece of data at any time. Conversely, the parallel systems we are concerned with can be described as multiple instruction, multiple data (MIMD), as each processing unit may perform different operations (instructions) on differing data streams. The differences encountered in the MIMD architecture pose unique difficulties to the software development process due to the complexity of working with multiple data and instruction streams combined with the general unfamiliarity of the many users who have only been trained on traditional SISD systems. These difficulties are compounded by the dearth of mature tools to aid in the development process for parallel systems.

The processing units of parallel systems often need to share and communicate data with each other in order to coordinate their activities. Two popular programming paradigms have arisen to address this issue: message passing and shared memory. In a shared memory system, processes act as if they share a single memory space. Changing the state of a variable in the system is as simple as assigning to it. However, this simplicity is not without drawbacks; the
primary difficulty being that a process reading a variable or structure may see that variable or structure in an incomplete state if another process is in the process of writing to it. For example, if one process is inserting an element into the middle of a linked list, it will have to change pointers around, however, another process may try to read the linked list before all the pointers have been correctly altered, resulting in an incorrect view of the linked list for the reading process. This difficulty leads to additional complexity, as there is now a need for the locking or synchronizing of structures on top of the general difficulties of programming for a concurrent architecture. This locking may even severely hamper performance if several processes are trying to access the same resource, forcing most to block while one performs its work. Shared memory systems are restrictive in where they can be used, typically only found on multiple CPU computers where memory is in actuality physically shared.

Message passing, is the more popular of the two paradigms for large problem sets, with such systems dominating the list of the fastest 500 computer systems in the world [TOP5]. In a message passing system, processes have their own memory space and send messages to each other (which the other process receives and reads) in order to communicate. Message passing systems are often considered to be lockless, avoiding many of the synchronization issues of shared memory systems. While message passing has its own difficulties, it is popular because the paradigm is able to be used in a greater variety of parallel systems [CORN]. Message passing can be used with a cluster of computers on a network (that can send messages with network sockets), a multiple CPU computer (where messages can be copied from the address space of one process to another), and can even work between heterogeneous machines (e.g., machines where the endian
order of bytes differs) if a standard data format for the messages is agreed upon. Clusters of individual networked machines using message passing are particular popular because they can be built from cheap commodity components, and can be expanded by simply adding additional computers to the network. If the system easily supports heterogeneity the administrator has the freedom to add any computer he wants, enabling him to choose what is best for the task at hand while preventing the obsolescence of previous purchases (a practice commonly found in grid computing). Due to message passing’s popularity, this thesis focuses on such systems and any references made to parallel or distributed systems made in this thesis shall refer to message passing systems unless otherwise noted.

Why do we need Debuggers for Parallel and Distributed Systems?

Anyone who has written a computer program has doubtlessly experienced a bug. The program may crash, produce erroneous output, or perhaps innocuously seem to do nothing at all. Ensuring program quality, especially in mission critical applications is often a time consuming and expensive endeavor: for example, it is estimated that NASA spent about $1000 per line of code in its Space Shuttle Avionics software [EDDU]. Much of this cost was spent towards identifying and correcting defects in the code. Fortunately, many tools and techniques have been developed to assist in the process of identifying and eliminating bugs within a program; mitigating costs and allowing the programmer to return to productive development. Common debugging techniques include the usage of print statements to reveal program state, or the usage of powerful tools such as Gdb
[GDB] and Rational Purify [PUR] that can provide great detail, or are specialized at handling a specific class of bug (e.g., Purify is well suited for analyzing and reporting memory related errors). However, these tools are generally designed to be used to debug a single sequential program at a time. This is perfectly acceptable for debugging a typical application where all work is performed within a single thread of control; however, this specialization is a hindrance when one is working with a parallel or distributed system where work is not performed exclusively in a single process, but where data and functionality are divided among many.

In a parallel or distributed system, execution state is not defined by a single process, but by all of them. Such processes need to communicate and coordinate, often in an asynchronous manner, with each other in order to ensure correct execution of their task. This communication of messages must conform to a protocol (possibly informal) for correct execution. In addition to these qualities, distributed processes may be physically separated from each other as they reside on different machines. These machines may be heterogeneous, possessing differing endian properties, operating systems, execution speeds, instruction set architectures, or file systems. Taken together, all these properties combine to present challenges for which traditional sequential tools and techniques were not designed, and used alone, are not suitable for use.

Research suggests that one reason why bugs and errors are hard to find and correct, is because the cause and effect of an error are often separated by great distance in time as well as code [EISE]. This difficulty is exacerbated in a parallel system, because the cause and effect of an error may not even be visible within a single process. That is, an error in one process may only be visible in
another, and in yet another may actually lie the root cause. For example, erroneous output in one process may actually be the result of bad input values which in turn were passed to it from another process that performed an erroneous calculation.

The increased distance between cause and effect, combined with the previously identified unique issues presented by a distributed system, makes debugging such a system a much more difficult task than debugging an individual process. The state of the system is much larger and more complex than that of a single process, threatening to deluge the programmer with a plethora of data, thus resulting in information overload. Previous parallel debugging efforts are often criticized for not dealing with this issue adequately [PAN1]. Traditional sequential debugging tools are of limited use, as they do not coordinate well to present the entire system. These tools lack the ability to abstract and efficiently analyze key distinguishing characteristics of a parallel message passing systems, making it difficult to detect common problems not found in traditional sequential programs, such as deadlock or failure to adhere to a protocol.

Objectives and Goals of this Thesis

The goal of the Millipede project and the multilevel debugging technique it employs is to provide a debugging tool that can deal with the unique issues and characteristics of parallel and distributed systems while avoiding many of the flaws of its predecessors, such as information overload. Millipede aims to achieve this goal by providing appropriate abstractions for the analysis of high level information, automated detection of certain classes of error distinct to parallel and message passing systems (such as deadlock and protocol violations), while
still providing the tools to analyze low level problems and facilitate the mapping of the cause of an error to the actual source code. Our final goal being a tool that allows the programmer to access the information he needs at the appropriate level without being overwhelmed; thus allowing him to quickly complete the debugging task and continue with program development.

Organization of this Thesis

An overview of previously developed parallel debugging tools and the flaws that Millipede attempts to correct is provided in Chapter 2. Millipede operates with the PVM distributed system. An introduction to the system is introduced in Chapter 3. The principles and rationale behind the multilevel debugging technique that Millipede employs are described in Chapter 4. Chapter 5 presents Millipede, detailing its usage and features, while Chapter 6 describes the more interesting implementation issues and details. We finish with conclusions and recommendations for future work in Chapter 7.
CHAPTER 2

PREVIOUS WORK

Years of effort have been put into the development of 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.

Extensions of Sequential Debugging Tools

A number of tools already exist to support parallel debugging. The simplest class of these tools are extensions of sequential debuggers. One may have an integrated environment or the tool may simply involve the use of N copies of an already existing sequential debugger such as Gdb—one for each process. We refer to the latter type as N-version debuggers. The manual for the popular PVM distributed system advocates this straightforward approach, discussing how to start a sequential debugging tool for each process [GEIS, pp. 157-159], and noting that “Adding printf() calls to your code is still a state-of-the-art methodology” [GEIS, p. 157].
The primary disadvantage of using an extension of a sequential debugging tool is the overwhelming amount of information. These tools typically operate at the source and machine code levels, providing finely grained information in excruciating detail for the state of each process. This information is indeed useful, and may be what the programmer needs. Unfortunately, this information is also often scattered among N different windows, one for each process. An example of this type of debugging can be seen in Figure 1, featuring PVM’s built-in debugging support to launch eight copies of Gdb along with a console application for some control over machine state.

Information from these tools is not made available in a global context, focusing on the individual process, and thus, we also lack queries that could gather information on a global scale. A global perspective is necessary in order to solve global problems; a perspective these tools fail to provide.

The fine granularity of information these tools provides is also a liability. Knowing what is happening at the sequential level may be useful to the user, but he may also need to know higher level information regarding such constructs as message passing and whether these messages are being passed in accordance with the system’s protocol. With the low level view these tools provide, such information could be extracted, but the burden is placed on the programmer, who must manually collect and interpret the data for these higher level constructs from the information he is provided.

Finally, we note that managing the execution of multiple processes in a sequential manner is burdensome to the user. It is difficult to focus on a single task when needing to attend to others.
These flaws cripple the effectiveness of debugging with an extension to a sequential system, making them unsuitable for the debugging of large and nontrivial parallel applications. The information they provide is often overwhelming, they lack the ability to effectively assist with debugging at a higher level of abstraction when necessary, and they make process management burdensome and distracting. Examples of extensions to sequential tools include TotalView [TVEW], pdbx [PDBX], and p2d2 [P2D2], as well as PVM’s own built-in support for debugging.

Figure 1: Debugging eight processes with PVM’s built-in support.
Visualization Tools

A second class of tools for debugging parallel programs are those that visualize the system’s behavior. A typical tool provides the user with a fixed set of views displaying the status and behavior of the system in various ways, such as charts and graphs. This class of tools is popular for analyzing messages. They may display information regarding what messages are pending in queues, the connections between processes, and which processes are actively computing. Figure 2 shows an example of this type of tool, the XPVM system, working with a relatively modest 8 processes that have sent 1208 messages.

Figure 2: The XPVM system in use with eight processes.
This class of tool often suffers the opposite of a key problem problem with sequential debuggers, as the granularity of the information they provide is too large. While extensions of sequential debuggers provide low level information but lack the abstractions to effectively analyze the program at a higher level, visualization tools provide the abstractions necessary to analyze at a high level, but do not provide the means to map what is happening at the machine or source code level. For example, by examining the output of a visualization tool, the programmer may determine that there is a stray message in the system, but he still has to locate what line and file this message was sent from; making the mapping of the error to the code that caused it difficult. We note too, that the burden of interpreting the data in order to locate this stray message is still entirely on the programmer. Just as acquiring information regarding message passing is burdensome to do with a sequential tool, finding protocol and other higher level information must still be accomplished manually by the programmer.

Additional difficulties arise when we consider that visualization tools are often used to display global information. Unfortunately, global views are often too vast for the programmer to easily locate relevant information. For example, the red rectangle in the lower left of Figure 2 in the Space Time: Tasks vs. Time panel, is actually a series of lines that show the flow of messages between processes. Unfortunately, so many messages have been sent that they form a solid rectangle, making it impossible to derive much meaningful information from this view besides the fact that a lot of messages were sent. Furthermore, the red rectangle is obscuring information underneath it, detailing what the processes are doing with their time. Because of the enormity of the data, this aspect of the tool has become nearly worthless.
Another flaw of visualization tools that is demonstrated in Figure 2 is that they often provide limited flexibility in how the data can be viewed, only incorporating what the author of the tool thought was important. For example, in the Space-Time view, the user cannot limit his view to the communication between two processes, making it difficult to analyze communication between just those two processes. This lack of flexibility is largely caused by the absence of powerful user defined queries. Queries could help mitigate the deluge of information found in a global view, yet support for such queries in these tools is rudimentary at best. With a powerful query mechanism, the programmer would be able to locate and isolate precisely the information he needs to diagnose a bug from the immense collection of available data.

Like extensions of sequential tools, the flaws of visualization tools limit their effectiveness. While the information they provide is often useful, they often lack the ability to display low level information relating to the source and machine code levels, which makes mapping an error to the code that caused it difficult, and are not very adaptable to the needs of the programmer. The views of the available information are limited to a subset of what the author thought was important, and while this flaw is hard to avoid, its effects are aggravated by the lack of a powerful query mechanism to alleviate the difficulty of sorting through global information. Moreover, while these tools are often well suited for displaying data, the burden of interpreting it is still almost exclusively placed on the programmer. Examples of visualization tools include Paradyn [PARA], Vampir [VAMP], and XPVM [XPVM].
Logging and Replay Tools

The final class of tools we will consider are those that log program execution, allowing the program to be analyzed or replayed postmortem. Like the previous tools, logging tools suffer from lack of granularity, in that they are typically only (well) suited for analyzing at the level of the events they log. A system that exclusively logs message passing events for example, is probably unsuitable for analyzing low level program logic errors. Unless the tools allows other tools to be used alongside it during replay, the programmer is limited to the analysis features that the logging tool provides. These features are vulnerable to the same flaws of visualization tools or extensions of sequential tools.

Still, the ability to replay a program is of immense value. Parallel programs often run for large amounts of time. If we could eliminate the time spent blocking to receive messages by logging and reading their contents from a log file, a replay of an execution may proceed much faster than if the program was run live. This could be of enormous convenience to the programmer, who would not have to wait for messages to be passed around to duplicate a bug. A logging system could also help with bugs that appear intermittently if it was able to capture the state of the system when the bug actually occurred. A powerful synergy can exist between a properly flexible replay tool and other analysis applications, and it would be prudent to incorporate this functionality into any new large scale analysis tool (see chapter 6). Examples of logging and replay tools include BUSTER [BUST], PVaniM [PVAN], and PDT [PDT].
Summary and Conclusions Regarding Previous Work

To summarize, the previous available work is largely hampered by one or more of the following difficulties:

- The perspective they provide is too narrow, failing to see the forest for the trees or the trees for the forest:
  - Extensions of sequential tools focus on low level information and make it difficult to extract global or higher level information.
  - Visualization tools generally focus on high level and global information, but do not allow one to see what is going on at the machine or source code levels.

- They overwhelm the user with information or do not provide enough:
  - It is difficult to manage and keep track of the low level state for several processes with a sequential tool.
  - Visualization tools only display what the author of the tool thought to be useful, but the programmer may need a different abstraction, lower level information, or simply an alternate view of the data.
  - Logging tools only collect data about specific events.

- They lack the flexibility to allow the user to find the data he needs:
  - A powerful query mechanism could mitigate the effort needed to locate the information the programmer needs, especially when examining vast global states and history.
The user still needs to interpret the data in order to find the cause of an error:

- The cause and effect relationship is not well supported. Being able to detect errors at the protocol level would give the user a good idea of the cause of many message related errors.

It could be argued that one could avoid many of these drawbacks by using multiple tools. However, this approach is not without its own caveats. Learning and using these tools is hampered by the fact that different people using different strategies developed them. Each tool has its own unique user interface, representations, formats, and quirks. Trying to learn and use these systems simultaneously is not a pleasant task in the face of these inconsistencies. Combined with general user conservativeness and suspicion towards new tools, the efficacy of this approach should be questioned. Moreover, these tools may not be designed to work in conjunction with each other, with one tool inadvertently affecting or interfering with the other. For example, consider the case where process A was examining process B, and altered the execution of process B such that it would send messages to process A detailing process B’s state. If another tool was examining the message passing behavior of process B, it would note the messages that process B sent to A. This occurs because the second program does not know the purpose of the messages heading towards the first, even though these messages would not be sent during normal execution and thus the user is distracted with irrelevant and potentially confusing information. Ideally we would like a tool that can provide for most (if not all) of the user’s debugging needs in a unified package. This approach avoids the problems of
inconsistency and compatibility with the multiple tool approach, and has the additional advantage that a single tool should encounter less resistance to adoption by the user compared to a multiple tool approach. A tool that meets these goals is precisely what we have attempted to provide with Millipede.
CHAPTER 3

THE PVM DISTRIBUTED SYSTEM

PVM stands for Parallel Virtual Machine. Essentially, PVM is software that allows several, possibly heterogeneous, machines to be combined into one large virtual machine that can run multiple tasks in parallel in a message passing environment. The virtual machine abstraction allows the tasks to interact with each other almost as easily as if they were located on the same physical machine. Low level details such as sockets for communication, data format, and launching remote programs are abstracted behind the PVM library. Most PVM programs are developed in C [CKR], although many other languages have bindings to the library. Millipede is designed to be used with the PVM system and thus it would be prudent to devote some space to cover the basics of how it operates, particularly in relation to message passing between processes.

Sending and Receiving Messages

PVM provides three basic functions for sending messages and four for receiving. The most basic send and receive function are: \texttt{pvm\_send(tid, tag)} and its counterpart \texttt{pvm\_recv(tid, tag)}. Each process in the PVM virtual machine has a tid (task identifier) which serves the same function as a process id in a unix system, a unique identifier for each process executing on the machine; only in this case, the machine is virtual. If one process knows the tid of another,
it may send that process a message with the `pvm_send` function. The tag parameter allows each message to be tagged with an integer which is used as a basic form of metadata. Generally, the tag is used to denote messages of different types. For instance, if a computation had multiple steps or phases, a tag could be used to identify which phase a message corresponds to. All the send functions are asynchronous, returning as soon as the message is safely within the PVM system. A multicast send is possible with `pvm_mcast`, which has the same semantics as `pvm_send` but can send the same message to multiple processes at once.

`Pvm_recv` works similarly to `pvm_send`. The user is able to specify from which process he wishes to receive from and what the tag of that message should be. The value -1 can be used for either parameter in `pvm_recv` to denote a wildcard (receive a message from any sender, or with any tag, or both). `Pvm_recv` is blocking, meaning that it will wait indefinitely until it has received a message that matches its parameters’ criteria. The function `pvm_nrecv` is the nonblocking equivalent of `pvm_recv`. `Pvm_nrecv` will return immediately with an error code indicating failure if no messages matching its criteria are waiting. The function `pvm_trecv` bridges the gap between `pvm_recv` and `pvm_nrecv`, allowing the user to specify a timeout value that determines how long the function may wait to receive a message before returning with an error code. Finally, we have the pair of functions `pvm_psend` and `pvm_precv` which are intended as to be used as an efficient, low latency means of sending and receiving messages. These functions avoid the packing and unpacking of data, a topic we cover in the next section.
Packing and Unpacking Data

Since PVM is intended for use in heterogeneous environments, it provides the tools to transfer data in the platform agnostic XDR [XDR] format. Data may also be transferred in its raw, native format (generally for the efficiency of avoiding the need to encode and decode transferred data to and from XDR), but such transfers must be between homogeneous systems. Data is encoded into XDR by “packing” it into a send buffer. PVM provides numerous functions of the form `pvm_pk*`, such as `pvm_pkint` and `pvm_pkdouble` which allow both arrays and individual elements of their corresponding primitive types to be transferred between processes by packing the type into the send buffer. If the buffer is specified as using XDR encoding (the default), the packed data will be translated to that format as well. The programmer builds up the message he wants to send by calling the `pvm_pk*` functions to fill up the send buffer and then transfers this buffer with one of the send functions (except for `pvm_psend` which takes a pointer to the data, a length, and a data type, and then proceeds to send the data raw). The pack functions have counterpart unpack functions (of the form `pvm_upk*`) which extract the data from a receive buffer. A receive buffer is created upon a successful call to a receive function (except for `pvm_precv` which will simply deposit the raw data at a provided pointer). Besides providing a mechanism for the transfer of data between heterogeneous systems, buffers provide a convenient abstraction for the programmer as they remove the need for memory management (a send buffer grows as more data is packed into it and the programmer does not need to know how big a message is when he receives it), and allow messages to be read and written piecemeal instead of all at once.
However, it is important to note that the packing and unpacking operations on a buffer must have a one-to-one correspondence to each other or unexpected results may happen. For example, when packing bytes, PVM will pad the number of bytes to the nearest multiple of four (so if one packed fourteen bytes with one call to `pvm_pkbyte`, PVM would actually place sixteen bytes into the buffer), the unpack functions know this and skip extraneous bytes in the buffer. However, if one were to unpack these fourteen bytes individually through fourteen calls to `pvm_upkbyte`, one would be at position $14 \times 4 = 56$ in the receive buffer after this sequence of calls because the unpack calls assumed that the data was packed by fourteen `pvm_pkbyte` calls; meaning each byte would have been padded with three extraneous bytes. Unpacking data as a different type than it was packed is also highly dangerous. Unpacking something as an integer which was originally packed as a floating point value would assign the binary representation of that floating point value to the integer value, an action that is clearly undesirable.

In summary, PVM is a message passing system. All sends are asynchronous while receives may be blocking or nonblocking. Like any message passing system, messages can be sent to wrong, dead, or nonexistent receivers. We gain an additional problem by allowing blocking receives because these allow for the possibility of deadlock. Deadlock will occur if a cycle of receives is formed (process A is in a blocking receive waiting for a message from process B while process B is in a blocking receive waiting for a message from process A), if a process is receiving from a dead or nonexistent tid, or if all processes are receiving or dead (although one process must be alive to have deadlock, otherwise all the processes are simply dead). While packing data into buffers handles
memory management and data formatting for the user, we also have the possibility of unpacking the data in the wrong way, leading to unpredictable behavior. One of our goals when designing Millipede was to take these common forms of error into consideration and provide the tools necessary to detect and analyze them.
CHAPTER 4

MULTILEVEL DEBUGGING

The concept of multilevel debugging as embraced by Millipede was initially developed in [PEDE]. Multilevel debugging is a bottom-up approach to the debugging problem. Many debugging tools, particularly those associated with visualization, take a top-down approach to debugging. A top-down tool provides the programmer with a global view of the data and leaves it to him to narrow the search space. Unfortunately, these tools often do not support going down far enough, lacking the ability to localize the error at the source code level. Such tools allow one to identify a problem and speculate as to its cause, but to examine the actual source code is outside the provided functionality.

In contrast, the bottom-up approach assumes that the programmer already has an idea of the class of error he is examining (most likely through the error message that was generated as a result). Instead of providing a global view and letting the user form a hypothesis as to the cause of the error, multilevel debugging provides not only the tools for hypothesis creation, but specialized tools for each class of error that assist in the verification and refinement of the user’s error hypothesis, with a particular emphasis that these tools should be able to map the cause of the error back to the source code level.

More opportunities for error arise in parallel and distributed programs than in traditional sequential software. Many of these errors are unique to parallel and
message passing systems and thus a bottom-up approach needs to provide tools for these new classes. In Millipede, we classify errors as belonging to three different classes: sequential, message passing, and protocol. Each class has bugs that are unique to it, and we shall proceed by briefly examining each class so that we may understand what should be developed in order to be effective at and hopefully even automate parts of the bug analysis and correction process.

Sequential Level Errors

Sequential errors are familiar to anyone who has developed a traditional sequential program. The number of errors in this class are numerous. A subset of possibilities include logic errors in branching or looping constructs, memory related problems such as accessing an invalid pointer or an out of bounds array subscript, failure to properly initialize variables, incorrect use or misunderstanding of an API or the semantics of the language, or perhaps the algorithm the programmer is using is simply incorrect. The myriad of possible error types indicates that we need powerful tools in order to detect all of them.

The most commonly used sequential debugging technique is the placement of print statements that give an idea of the state of the program [PAN2]. Print statements are favored for their sheer simplicity, wide applicability (this technique can be used with any language with the capability to output text), and lack of a learning curve. Therefore, as a first step, support for communication through the standard IO mechanisms should be provided.

For more difficult and sophisticated problems, we should note that decades of research and effort have been spent on developing many fine tools, such as Gdb, to identify and correct sequential errors. It would be prudent to apply one of
these mature tools to debugging sequential level errors in order to avoid the redundancy of reinventing the wheel and the effort in refining it. We should take into consideration however, if we are to improve on previous efforts, our support for sequential debugging must avoid overwhelming the user with information or requiring too much user supervision.

Millipede provides several tools for working with sequential errors that meet these goals. It integrates Gdb in a nonintrusive fashion to provide a sophisticated tool for difficult sequential debugging tasks. The standard IO streams of each process are routed to Millipede, including stdin (this capability being significant because it is outside the provided functionality of the PVM system and is a sore point for many application developers), to allow the usage of print statements and interaction with the debugged program. Furthermore, PVM calls are logged to a file in order to provide a history of previous state, allowing the user to replay the execution of an individual PVM process offline under any sequential debugger with the addition of these log files.

Message Level Errors

Moving from a traditional sequential to a parallel or distributed program brings a new class of error due to the introduction of interprocess communication. While PVM takes care of many of the low-level details such as data representation and the channels of communication, there is still much that can go wrong while passing messages. Examples of errors at this level include improperly unpacking data in the ways that were mentioned in chapter 3, incorrect values being being sent and received, or despite proper packing and unpacking, misinterpretation of the data. Many of these errors are due to a
mismatch in the modules composing the sender and receiver processes. For example, the receiver may expect the sender to pack the data in a certain format, but the sender fails to do as expected. Errors at this level are typically more difficult to find and diagnose than sequential errors due to the increased distance between the cause and effect of a bug.

The distance between the cause and effect of a message level error increases spatially, temporally, and in code. Spatially because the cause and effect of a message level occur in separate processes. Temporally because it takes time to transfer a message, and due to the asynchronous nature of sending in PVM, the processing of the received message by the recipient may take place much later than when the send completed. Distance in code may be increased as the two processes may not have the same code base (i.e., the processes may be different modules), such as when a slave process communicates with a master process in the popular master/slave paradigm of parallel processing [PAPR]. Distance may be further extended if the effect of the error is not externally visible or noticed until it has propagated through several processes.

Since message passing occurs between multiple processes, it is often not sufficient to examine them in an individual context; instead, a global perspective is needed. However, a good debugging tool will avoid overwhelming the user with the enormity of this global information. We accomplish this by placing communication information in a SQL [SQL] database and make it available to the user along with several useful predefined queries that can detect common errors such as mismatches between packs and unpacks. This database allows the user to limit his view to only the data relevant to the task at hand, while giving him the flexibility to create his own views of the data with a powerful query mechanism.
For simpler tasks that can be managed without a global view of the data, a summary of communication behavior is provided for each individual process. Additionally, we allow the user to set breakpoints at PVM calls in order to control communication behavior as well as interactively view and edit the data a process has received.

Protocol Level Errors

The protocol level is our highest level of analysis. Millipede provides tools to detect deadlock and to verify that the message passing behavior of a parallel program conforms to a provided protocol specification. The tools provided at this level are, for the most part, automated. Deadlock is an undesirable state that can be detected without user intervention merely by observing which processes are alive and in a blocking receive call. Messages may also be sent to wrong receivers, with an incorrect tags, or be received by the wrong line of code. Millipede allows the user to specify a protocol detailing such issues and can automatically verify that the program has executed in accordance with this specification. The tools provided at this level allow for the easy and automatic detection of errors that would be difficult and time consuming to discover when working with data at a lower level.

By dividing up the error space into three components and providing tools that can analyze at each, we argue that the multilevel debugging support in Millipede makes it a more effective and compelling tool than its predecessors. The multilevel debugging approach taken by Millipede provides the necessary tools and abstractions to analyze the unique properties of parallel message passing
systems while maintaining flexibility, providing automated tools, and avoiding
information overload as we shall cover in detail in the next chapter.
CHAPTER 5

INTRODUCING MILLIPEDE

Millipede builds upon the work of a previous proof-of-concept multilevel debugger that was presented in [PEDE]. It expands upon this prototype with the inclusion of many new debugging features and ideas, improved usability and user efficiency through the addition of a GUI [GUI], and improved cross-platform compatibility through the use of Java [JAVA].

Architecture and Overview

Millipede currently operates with the PVM distributed system, and consists of two components: a native 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). These two aspects work together with the Millipede Debugger analyzing, logging, and displaying the debugging information sent to it as a result of calls made into the debugging library by participating distributed processes.

During a debugging session, the Millipede Debugger acts as a central manager for all spawned child processes. 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). Differing behavior as a result of a timing difference in a parallel or distributed program demonstrates nondeterminism, an undesirable trait in most programs. Although less severe, timing differences in execution also occur and need to be accounted for when debugging traditional sequential programs.

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 [PGRE]. 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 [PACD]) in a free, powerful, standards conforming package that many users already have the skills to exploit.

PVM programs 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 [JPVM]. 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 instead, 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 distributed systems by providing processor independence, a rich set of libraries along with a common runtime environment, as well as the ability to communicate complex objects between processes through the use of object serialization.

In true distributed fashion, the Millipede Debugger and the SQL database need not be run on the computers where the child processes are spawned as illustrated by Figure 3. The Millipede debugger need only be able to connect to the PVM virtual machine and simple TCP/IP network connectivity is required between the child processes and the SQL database. Figure 3 shows three
computers involved in the Millipede debugging session, one running the SQL database, one running a user process, and one running the Millipede Debugger and a user process. The two user processes communicate through standard PVM calls. The Millipede library is built on top of the PVM library and communicates with the Millipede debugger via PVM as well. The library also maintains a connection to the SQL database in order to insert message and protocol related information, which the Millipede Debugger can retrieve on demand at a later time.

By logging information with the Millipede Debugger and the SQL database, the user is able to analyze sessions both interactively and postmortem, even simultaneously; allowing a form of relative debugging where the state and results of one run of a program can be compared with another [RELD]. This could be used to discover differences between different versions of the program or differences in execution between heterogeneous architectures.

Sequential Level Debugging

Most of Millipede’s sequential level debugging support is provided by its main interface pane pictured in Figure 4. In this figure, we can tell from the tabs at the top that we are currently inspecting the status of a process with the TID 262238. Below the tabs, we can see that the process name is _Wave_master_ and that this process is currently alive, as denoted by the smiling icon found to the right. The user may switch his view to one of the four other available processes by clicking on the appropriate tab.

On the left side of Figure 4, we can see that a sequential log of activity is kept for each process within a colorful table. An enlarged view of this table is shown
in Figure 5. This table keeps track of all PVM API calls. For each call, the table displays its name, the source file and line where it was made, the call’s return code (generally denoting success or failure), the input parameters, and information specifically regarding communication (the message and item number columns, the details of which will be covered in the next section). Each call is colored according to its type and whether it was successful or not, allowing the user to easily spot errors or calls relating to a specific aspect of his program (errors are red, ineffectual calls, such as a \texttt{pvm_nrecv} made when no messages were pending, are gray, sends are dark blue, receives are brown, etc.). Through the use of a
contextual menu (also displayed in Figure 5), the user can delve deeper into the details of the call. The Jump to Code option allows the user to view the source where the call was made (Figure 6, the red text distinguishes the pertinent line). If the user does not recall the specifics of how a particular call works, the call’s manual page (Figure 7) is available from the Display Man Page Option. When an error does occur, it is useful to know the details of the returned error code for the affected call (Figure 8). Alternatively, if a call is successful, its results (such as the unpacked data as shown in Figure 9) are often relevant.

The log table is 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.” [GEIS, p. 239] This ensures that log messages from a child process will arrive at and be processed by the Millipede Debugger in the same order they were sent (i.e., in the same order the PVM API calls were made). Because of this property, no special timing issues such as Lamport clocks [LAMP] need to be addressed.

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 Figure 4. 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 his individual processes. Outside of a debugging context, this feature is useful for simply interacting with each process. In Figure 4, the user interactively specified to the Wave_master program that he wished for four slaves to be spawned. We can easily differentiate the user’s input from the program’s output by the fact that the user’s input is colored blue. This IO redirection is accomplished by replacing the standard IO file descriptors in each process with a TCP socket’s file descriptor that maintains a connection to the Millipede Debugger.

If the tools presented so far are not enough to handle the sequential debugging task at hand, the user may directly attach Gdb to the process (which can even be accomplished for a process residing on another physical machine). This attachment is accomplished by hitting the “Attach GDB” button visible in the lower right of Figure 4. The user is able to dynamically attach and detach to a running process during program execution as needed. A sample Gdb session
where the user steps through the code and inspects variables is shown in Figure 10. Detachment is as simple as closing the window. The addition of the Millipede library should not affect the debugging of a process in any way besides that as a consequence of the fact that the Millipede library is used in place of the standard PVM library, one may start or step into the insides of a Millipede library call where the user’s code indicated a PVM call. Aside from this, the process merely needs to be compiled with the standard debugging option (-g in gcc [GCC]) specified as normal in order to generate the necessary debugging info.

Finally, by logging the results of the PVM calls, the user can launch any one of the individual processes of the system and use the log file as a replacement to the PVM runtime. As long as the process makes the same PVM API calls, in the same order, with the same parameters (with the exception being pointers that are used as out parameters), the user can isolate and replay the execution of a process and optionally analyze it using any external sequential tool he may desire (a sample run is shown in Figure 11 where we replay a process from a log file under Gdb, note the use of a simple script at the top which performs setup).

With these tools and abstractions, the user is able to effectively debug his program at the sequential level. The log file provides a convenient summary of his program’s current and past activity and can be explored in depth, including mapping program events back to source code. The user may use the program’s standard IO streams to communicate in a familiar manner, and when faced with more difficult problems he may directly attach Gdb to a process in order to examine it in depth. Additional analysis can be performed postmortem, where the user can replay the execution of a process using the log file as a substitute for
the PVM runtime. This analysis can be performed using any sequential debugging tool of his choosing in an isolated environment.

![Figure 6: Viewing the source code of the highlighted call in Figure 4.](image)

```c
/* Wait for results from slaves */
msgtype = 5;
y = (double *) calloc(n, sizeof(double));
e = (n%proc); /* e = n mod nproc ; Antal slaver med ekstra punkt */
for(i=0 ; i<nproc ; i++) {
    printf("Wave_master.c: waiting for results......\n");
    fflush(stdout);
pvm_recv(-1, -1);/* PROTOCOL(receive_results) */
pvm_upkint(&who, 1, 1);
pvm_upkint(&result_length, 1, 1);
index = who*(n/nproc) + (who<e)?who : (who>=e)*e;
pvm_upkdouble(&y[index],result_length, 1);
char testString[500];
pvm_upkstr(testString);
}

//ERIK
if (y_file != stdout)
    fclose(y_file);

//ERIK
y_file = fopen("./Par_res.txt", "w"); //changed to local directory
```
Figure 7: Displaying the PVM man page for an unpack call.

Figure 8: Displaying the error message for a failed call where the user tried to add the same host twice to the PVM virtual machine.
Figure 9: Displaying the result (contents received) of a `pvm_upkdouble(&y[index], ...)` call.

Figure 10: Using Gdb from within Millipede to inspect variables.
Message Level Debugging

As noted previously, the process log table displays information concerning communication. Each sent message is given a unique message number to identify it. This number identifier is displayed in the ‘MsgNo’ column of the log table. The second through sixth lines of Figure 5 show that the inspected process packed a double value into a message and proceeded to send this message to process 26154. This message was numbered 165. The process later received a replying message identified as 167 from the same process it had just sent to. A double value was then unpacked from this received message.

The user can view the order in which 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 Figure 4, displays a summary of communication information. An enlarged view is available in Figure 12. With the summary table, the user may view the messages an individual process has sent, received, packed, or unpacked in the context of other calls of that type. From the process log table, one may jump directly to the matching call in the summary table and conversely, from the summary table, one may jump to the corresponding entry in the log table. The summary table is intended to allow the user to view how communication calls fit in with others of the same type in a simple and easy to understand manner while providing the ability to transfer and consider the call in a more sequential context.

![Figure 12: The summary table, showing what messages this process has received.](image)

If the simplicity of the summary and log tables are not sufficient to solve a 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 along with an example of a custom query is show in Figure 13. Entries are placed in the SQL database before the corresponding log entry is sent to the Millipede Debugger, ensuring that everything in the database is at least as up-to-date as the information displayed in the log table.

The left side of the SQL pane contains a listing of the available tables, so the user can see what is available, removing the need to remember the names of the columns and their types. Currently four tables are provided. The **Senders** and **Receivers** relations allow one to query which messages have been sent and received, while the **SentMessages** and **ReceivedMessages** relations allow one to query what data has been packed and unpacked. By performing a join between **Senders** and **SentMessages** the user can gain information on where messages were sent and what they contained, and by joining **Receivers** and **ReceivedMessages** the user can tell what messages were received and what was extracted from them.

We note that all tables provide a link to the the source code that produced them, providing the file and line of the call that created the entry. Successful queries into the database are saved and a number of predefined queries (to which the user may add his own) are provided for common and involved questions the user may wish to ask the database. These predefined queries provide a good demonstration of the power and flexibility made available through the SQL database.
Six predefined queries are provided by Millipede: `match`, `locate`, `status`, `dump`, `unmatched_sends`, and `unmatched_pk_upks`. `Match` and `locate` take parameters which they use to query data about specific events, `status` and `dump` provide summaries of the global information, while `unmatched_sends` and `unmatched_pk_upks` run queries that can automatically detect two common classes of error over the global space: sent messages that have not been received and unmatched or mismatched calls that pack and unpack data.
Given a specific message (i.e., message number), it is often useful to know what has been packed and unpacked to and from it (the user may wish to know this so he can spot erroneous and missing unpacks from that message, or just to see where those packs and unpacks occurred in the code). This is what the \texttt{match} query determines. Given a message number, the \texttt{match} query matches the packs and unpacks of a sent message and marks those that have their counterpart missing, were unpacked as a wrong type, or where an incorrect length was unpacked. Figure 14 shows a \texttt{match} query on a message that was multicast but where the data has not been fully unpacked by all the recipients (those denoted with unchecked boxes). We see that the message was sent by process 262248 to processes 262250, 262251, 1572865, and 1572866. Process 262250 has unpacked all of this message’s data. Processes 262251 and 1572866 have unpacked the first element of the message (an integer denoting the number of processes), but only 262251 has unpacked the second element (an array of integers containing a list of the tids for each process involved in the distributed program). The other two processes have not unpacked any data from the message yet (possibly because they have not received the message).

Often, it is useful to see what messages have been sent and received between two lines of code. The user may wish see to see how two modules are communicating with each other and verify that messages being sent at a certain line are being received at a certain line. The \texttt{locate} query provides this functionality. Given two line numbers, it will find all messages sent and received between them. An example is shown in Figure 15 where we try to find all messages that were sent and received by \texttt{Wave_slave.c} at lines 83 and 88 respectively.
Figure 14: A **match** query where a multicast message has not been fully unpacked by its recipients.

Figure 15: A **locate** query between lines 83 and 88 of Wave_slave.c
It may be useful to view the state and history of the entire communication system in an organized fashion. The *status* and *dump* queries provide this functionality. *Status* quite literally displays the status of the message passing system, showing all sent messages and indicating which messages have been received and which are still pending on a global scale; marking the pending messages as potentially problematic. *Dump* displays all messages that have been sent, but does so on a per process basis, allowing one to see which messages have been present in each process’ message queue. Figures 16 and 17 show the results of a *status* and *dump* query respectively.

The *status* and *match* queries have the ability to flag suspicious activity like unreceived messages and mismatched pack and unpack calls. The functions *unmatched_sends*, and *unmatched_pk_upks*, automate the identification of suspicious message passing behavior without clouding the results with successful calls. *Unmatched_sends* functions largely like *status* except that it only displays the pending calls and does not display any receiver information (besides who the message was sent to), because by definition the message has not been received yet. The *unmatched_pk_upks* query, works like *match*, except that it only displays mismatched or unmatched pack/unpack calls and is not restricted to a single message number. *Unmatched_pk_upks* displays both sender and receiver information because an unmatched call can occur on both sides. We can have an unmatched pack call if the receiver simply does not unpack everything in his message buffer. An unmatched unpack call is a little more complicated but can still happen. For example, the sender may pack 100 bytes in a single call, while the receiver makes multiple one byte unpack calls in an attempt to retrieve them, thus resulting in more unpack calls than pack. Once a problematic message has
been identified, the user can use the previous predefined queries or his own custom queries to extract more information. Results of an `unmatched_sends` and an `unmatched_pk_upks` call are displayed in Figures 18 and 19.

Figure 16: The `status` query displaying the current state of the message system.
Figure 17: The *dump* query displaying what has been sent, organized by message queue.

Figure 18: The *unmatched_sends* query in the same state as Figure 16.
Figure 19: The `unmatched_pk_upks` query showing the data from several messages that have not been unpacked or were unpacked incorrectly.

Besides the summary pane and SQL database, one last tool is provided for message level analysis, namely the ability to set breakpoints at PVM API calls and edit message data. The interface to accomplish this can be found in the lower right corner of Figure 4. The user 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 received (Figure 20). Editing of values could be used to correct faulty data without the need to restart the program or to simply observe the process’ behavior when passed different values.

This control is managed by sending messages through the PVM infrastructure. Every time a PVM API call is about to be made, a message describing the call is sent to the Millipede Debugger. Upon receipt of a response, execution continues, unless the current call unpacks data, which may result in the data being sent to the Millipede Debugger and waiting for another response containing any edited values. Although this system sends many messages, these
messages are very small, containing only one byte of data if the do not unpack anything, meaning that we essentially only pay the cost of latency. The control flow scheme was implemented in this way to ensure that all updates the user makes regarding breakpoints are reflected immediately. As soon as the debugger is instructed to have the process stop at a particular API call (the interface for this is shown in Figure 21), 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.

Through these tools, Millipede provides an effective environment for debugging programs at the message passing level. It is possible to quickly determine who is sending and receiving information and what that information is. Sent data can be edited and break points from PVM API calls can be set in order to affect control flow at a high level. Predefined queries are provided that can ease and automate the detection of message level errors, and if additional flexibility is required, it is possible to perform a custom query of the SQL database containing global communication information.
Figure 20: Editing unpacked values. Index 16 is being edited.

Figure 21: Setting PVM breakpoints
Protocol Level Debugging

The protocol conformance panel as pictured in Figure 22, handles the task of protocol level debugging. Automated checking of conformance to a user specified protocol and deadlock detection are provided by Millipede. Figure 22 shows the results of a problematic program that both violates the user’s protocol and is in a state of deadlock.

The user’s protocol specification is visible in the upper left text area of Figure 22. The examined program is setup in the common master-slave relationship. Execution starts with a master process (\textit{Wave\_master}) being spawned. This process reads its initial parameters and interacts with the user. Once setup has
been finished, the master spawns a number of slave processes (`Wave_slave`) which organize themselves in a line topology [PAPR]. At this point, communication between the processes begins to happen, resulting in three types of message passing events:

1. First, each slave is sent some initial configuration information by the master.

2. Each slave begins computations. At the end of its computation it shares its results with its left and right neighbors (assuming it has them). This behavior repeats for several cycles.

3. After these cycles complete, the slaves return their results to the master process which coordinates and outputs their answers.

Each of these three steps are reflected in the protocol rules displayed in the large text area visible in the upper left of Figure 22 and are shown with their lines numbers in Figure 23 below. This protocol specification language is a slightly modified version of the one found in [PEDE], where the semantics have been simplified to only consider a rule to be matched only if both the sender and receiver side of it are satisfied. Protocol is evaluated only when information about both the sending and receiving of a message is available. Messages that are still pending in the system and have not been received yet are displayed in the unreceived messages table found at the upper right of Figure 22.

A protocol specification file starts out with a declaration of the source code files that will be involved (i.e., the source code files that compose the program). These declarations are found in lines 1 and 2 of Figure 23, where `slave` has been
defined as the source file `Wave_slave_deadlock.c` and `master` has been defined as `Wave_master_deadlock.c`. The actual protocol rules are specified in lines 4 through 7. A rule takes the form: `file_name([group])([rank])([line])([tag]) -> file_name([group])([rank])([line])([tag]) ::quantifiers`, where items wrapped in brackets are optional. Note that each rule is divided into two parts by the `->` symbol. The left hand part specifies the conditions a message must meet on the sender’s side in order to satisfy the rule. Conversely, the right hand part specifies the conditions a message must meet on the receiver’s side in order to satisfy the rule.

A sender or receiver rule consists of five components: `file_name`, `group`, `rank`, `line`, and `tag`. All components are optional except for `file_name`. If a component is omitted, it is considered to be satisfied (i.e., evaluates to true). `File_name` specifies which files the rule refers to. `Line` and `tag` are simple properties that specify what line and what tag a message should be sent or received with in order to satisfy the rule. In order to make specifying a line easier for an evolving program, one may mark lines in their source code with a comment of the form `/* PROTOCOL(Line_Identifier) */`, where `Line_Identifier` is an identifier with which that line may be referred to. An
example of this markup can be seen in Figure 24. In Figure 23, we note that all lines are referred to by their identifiers (send_left, send_right, receive_left, receive_right, etc.) instead of with raw integers, so that as the program evolves, and lines are added or removed, no change to the line numbers need to be made, as the Millipede Debugger will parse the source files and substitute in the correct line number as necessary. Group and rank are concepts not directly supported by PVM, but supported by Millipede. When a process is spawned, it is assigned a group and a rank. If this process has never been spawned before, its group is 0, otherwise its group is determined by how many previous calls to spawn for the same process have been made. Multiple processes may be spawned with a single call, and thus many processes may belong to the same group. A process’ rank is determined in ascending order according to its tid in a spawned group with the lowest tid member having a rank of 0. Group and rank allow
structure to be placed upon spawned processes. For example, the slave processes that form into a line topology all belong to the same group, and are laid out from left to right according to their rank. With these preliminaries covered, we can now examine the actual rules.

The first rule at line 4, $\text{master}(0)(0)(\text{send\_parameters})() \rightarrow \text{slave}(0)() (\text{receive\_parameters})()$, corresponds to the first message event where the master sends configuration information (i.e., parameters) to each slave process. This rule specifies that the master process with rank and group of 0 (i.e., the first and only spawned master process), may send messages from a line marked as \textit{send\_parameters}. If this is true, then the receiver portion of the message can be evaluated. In the receiver portion of this message, we see that this message is supposed to be received by slave processes that belong in group 0 (the first group of spawned slave processes) of any rank at a line marked \textit{receive\_parameters}. If both the sender and receiver conditions are met, the rule is considered to be satisfied. The last rule, $\text{slave}(0)(i)(\text{send\_results})() \rightarrow \text{master}(0)(0)(\text{receive\_results})() : \forall i : i < s\text{GroupSize}$, corresponds to the final message events where the slaves return their results to the master, and is similar to the first. This rule indicates that the slaves may send from a line marked \textit{send\_results} and this message should be received by the master at the line marked \textit{receive\_results}. We note that this last rule however binds the variable $i$ to the sender’s rank, and although unnecessary, this rule has a quantifier, which checks that $i$ is less than the sender’s group size (sender group size and receiver group size can be referred to by the constants $s\text{GroupSize}$ and $r\text{GroupSize}$ respectively). These constants can have different
meaning depending on topology. For example, if the receiver’s group is in a line
topology, then \( r\text{GroupSize} - 1 \) would refer to the rightmost process.

The second and third rules specified on lines 6 and 7, refer to the second part
of the message sending events, where the slaves communicate with their left and
right neighbors. They are similar, only differing in the fact that messages sent at
send_left should be received at receive_right (if a message is sent to a slave’s
left neighbor, that neighbor is receiving the message from his right) and messages
sent at send_right should be received at receive_left. Again, \( i \) is bound to
the rank of the sending process, but we can enforce that messages actually be
sent to a left or right neighbor with the expressions \( i-1 \) or \( i+1 \) used as the rank
on the receiver side. These expressions can be composed of the standard
arithmetic operators as well as involve modulo (useful for processes organized in
grid topologies) and exponents (useful for tree structures).

Messages that violate the protocol are displayed in the violating receives table
as shown in the lower right section of Figure 22. A violation indicates that either
the system is flawed, or the user’s understanding of how it should operate is,
either of which should be corrected. Multiple levels of sensitivity can be used for
the detection of violating messages. At its strictest sensitivity, any message that
satisfies zero or more than one rule is marked as violating (that is, every message
should satisfy one and only one rule). If the protocol is fully specified, but not
uniquely, the user may lessen the sensitivity so that only messages that satisfy
zero rules are considered to be violating. At its lowest sensitivity, messages that
satisfy zero rules are not marked as violating. This may be used during program
development when a full specification has not been developed yet and the user
does not want to be constantly alerted to protocol violations. At all levels,
messages whose evaluation results in an error during the interpretation of a protocol rule are flagged.

Once a violating message has been detected, further information as to the cause can be determined through the lower level message and sequential tools. To aid this process, Millipede can directly show the code where the violating send and receive were executed. This functionality is shown in Figure 24. In Figure 24, we see that a message that was sent at the line marked send_left, was not received by the line marked receive_right, but instead an anonymous unmarked line below it. This feature is another example of the ability of Millipede to map a high level error to the actual source code the caused it.

Millipede also supports the automatic detection of deadlock. As mentioned in chapter 3, deadlock can occur as a result of three events involving blocking receives: if a cycle of receives is formed, a process is receiving from a dead or nonexistent tid, or if all processes are receiving or dead (with at least one non-dead process). All three of these events can be detected through analysis of the state of the message passing system and knowledge of the liveness of processes. The text area at the lower left of Figure 22 shows the results of this analysis, indicating that all processes are stuck in receives and a cycle has been detected. If we examine the receiver code in Figure 24 again, we note that the line marked receive_right is actually exactly the same as the line marked receive_left, a common copy and paste error that has resulted in all slave processes except the leftmost to attempt to receive from the left when they should have been attempting to receive from the right. Since the leftmost slave is receiving from the right, and its right neighbor is receiving from the left, a cycle is formed. The next slave process is trying to receive from the second deadlocked process, with
its right neighbor trying to receive from it, and so on; resulting in all processes waiting in blocking receives. The automated system has detected and informed the user of both these critical errors.

Other Tools

A number of other tools that do not fit precisely into the multilevel debugging paradigm are available to ease the user’s debugging experience. For instance, a distributed debugger should consider the fact that there is a considerable amount of metadata associated with each process. In both sequential and parallel programs it is common for the programmer to wish to know what state a process is in (whether it is dead, alive, or blocking) or what arguments a program was launched with. The additional complexity of a distributed system introduces additional opportunities for metadata, such as what machine a process is running on (since it may not be the same machine the user is debugging on), and what its process id (PID) is. As a result of processes running on different machines, the location of the executable in the file system may also be different and should be recorded. We also note that Millipede associates its own metadata to processes, such as the group and rank properties assigned by the protocol level tools. This metadata is accessible through the “Additional Process Info” button shown in Figure 4. Figure 25 shows the results of displaying this data. Additionally, we note that in the upper right of Figure 4, there is a face icon that indicates the state of each process (alive, dead, or blocking) at a glance.

Another common difficulty arises from the fact that it is often confusing to navigate between processes using only their numeric tids. Millipede alleviates this problem by providing the “Processes” menu as shown in Figure 26. This
menu enables the user to jump between processes in a structured fashion, allowing the user to locate a process by its name, group, and rank. This feature also allows for viewing the state of multiple processes (with the face icons) in an organized manner.

Finally, we consider the fact that in addition to retrieving metadata related to individual processes, a user may wish to retrieve metadata relating to the entirety of the PVM virtual machine itself (such as what physical machines the virtual machine is constructed from). This functionality is provided by Millipede’s integration of the PVM console application as show in Figure 27. The console allows the user to view and affect virtual machine state in a simple fashion that is familiar to most users of PVM.

Summary of Tools

Millipede provides a number of tools to aid in the debugging process. In order to avoid the twin failings of information overload and not providing the necessary information to debug a particular problem, Millipede provides tools at three primary levels of analysis: sequential, message passing, and protocol. The tools at each level range from the simple (access to the standard IO streams of a process) to the sophisticated and flexible (detection of protocol violations) allowing the user to choose the tool that is most appropriate for his needs [PEDE]. In order to ease the tracking of cause to effect, tools at all levels support the vital task of mapping their information back to the original source code. Some of the provided tools are even automated, facilitating the detection of certain common classes of error at the message passing and protocol levels (unreceived messages, mismatches between packs and unpacks, deadlock, etc.),
without user intervention, further lessening the burden on the user to sift through the available information and allowing him to return to productive program development.

Figure 25: The process metadata for Wave_master.

Figure 26: The processes menu. The first slave is dead, the second is blocking, and the rest are alive and executing normally.
Figure 27: Using the PVM console to inspect virtual machine state.
CHAPTER 6

THE IMPLEMENTATION OF MILLIPEDE

As mentioned in the previous chapter, Millipede exists as two components: a native library (written in C) used by the examined processes to communicate with the second component, a Java application (the Millipede Debugger) that along with performing most of the required analysis, provides a graphical user interface to display the results of these analyses, much of the raw data, and which facilitates interaction with the user. This chapter focuses on highlighting some of the more interesting implementation details and difficulties found in these two aspects of Millipede.

The Native Library

The purpose of the native library is to act as an intermediary between an examined process’ source code and the PVM library. When calls are made from the process’ code into PVM, these calls are instead routed to the Millipede library which can perform such tasks as sending information back to the Millipede Debugger, waiting for permission to continue execution (i.e., respond to breakpoints), and inserting entries into the SQL database. These actions can be performed both before and after the actual execution of a requested PVM call (e.g., a request for breakpoint information is made before a call is made into the PVM library, while an entry into the SQL database is made only after it has been
determined whether the call succeeded). This intercession performed by the Millipede library call is accomplished through the use of the C preprocessor to \#define macros that redirect the call to a PVM function to the Millipede library.

To see how this is implemented, consider the following:

(1) The prototype for the standard \texttt{pvm_upkdouble} function is:
\begin{verbatim}
int pvm_upkdouble( double *dp, int nitem, int stride ).
\end{verbatim}

(2) There is a corresponding function in the Millipede library of the form:
\begin{verbatim}
int _PVM_upkdouble(char* variable, double *dp, int nitem, int stride, char* pname, int line).
\end{verbatim}
The signature is similar to the corresponding PVM call, but has three extra parameters: \texttt{variable} which represents the name of the variable an unpack call is writing to, \texttt{pname} which represents the name of the file where that was made, and \texttt{line} which indicates the corresponding line of that file.

(3) We can change a call to \texttt{pvm_upkdouble} to \texttt{_PVM_upkdouble} with the following macro: \begin{verbatim}
#define pvm_upkdouble(X,Y,Z) _PVM_upkdouble(#X,X,Y,Z,__FILE__,__LINE__).\end{verbatim} We note that the preprocessor is able to provide us with necessary variable, file, and line information.

This technique is repeated for all PVM functions that Millipede supports, essentially allowing us to override calls made in the user’s code. These macros are placed into the user’s code by replacing the standard PVM header file with our own. At compile time, a flag is checked in order to determine if the program should include Millipede (the user indicates this by including \texttt{-DMILLIPEDE} or \texttt{-DDEBUG} as arguments to gcc). If this flag has not been defined, the standard
PVM header file is included as normal; if the value is defined, then the macros and a Millipede header file are used in place of the PVM header file.

A particularly involved issue that arose during development was the fact that when a new process starts up, it needs to connect with the Millipede Debugger. The connection needs to be established before control is given to user code (i.e., before \texttt{main} is executed), primarily because we do not know what this user code will do (e.g., if it crashes on the first line, when or if it will make any PVM calls, etc.); therefore, the mechanisms for interacting with the Millipede Debugger should be setup beforehand so they can be exploited immediately. Execution before \texttt{main} can be achieved by using gcc, which allows functions to be marked with various attributes. The attribute of particular interest to us is \texttt{constructor}, indicating that the marked function will be executed before \texttt{main}, at the time of global variable initialization (we briefly note that if the function is in a dynamic library, it will execute when that library is loaded; in order to avoid this, the Millipede library is a static library). We exploit this functionality to initialize and perform setup before \texttt{main} by declaring and implementing the function:

\begin{verbatim}
void initializeGlobals(void) __attribute__((constructor)).
\end{verbatim}

This \texttt{initializeGlobals} function acquires the necessary setup information in a message from the parent that spawned the process (which may be the Millipede Debugger), via the PVM infrastructure. The setup information includes such data as the tid of the Millipede Debugger (i.e., where all the Millipede related messages should go), what features are being used (the SQL database, breakpoints, whether events are being logged, etc.), as well as an IP address and port for connecting standard IO to.
We can easily replace the standard IO file descriptors with a TCP socket because TCP provides a reliable stream abstraction to network communications. The Millipede library will form a TCP connection with the Millipede Debugger (the necessary information already being known as it was received by the initial setup message), transfer information that will allow the debugger to identify it (i.e., the process’ tid), and from then on, this socket will be used for standard IO. The original standard IO file descriptors can be closed and replaced with the `dup2` system call, after which, higher level functions such as `printf` and `scanf` can be used as normal for communication with the debugger.

Once standard IO is connected with the Millipede Debugger, any errors in the Millipede library (generally dealing with memory allocation) can be reported by sending messages through it. However, there is a period of time before standard IO is connected, where errors can still occur (e.g., failure to join the PVM system, in creating the TCP socket between the process and the debugger, or in the allocation of memory). These errors need to be handled and reported as they are generally due to improper system configuration. In order to handle this rare case, before a connection is established with the debugger, output is routed to the file ‘/tmp/MillipedeErrorLog-pid’, where `pid` is the process’ id. The file is deleted once communication with the Millipede Debugger is established (if an error occurs, the process will terminate before the file is deleted). Thus, between standard IO and the error file, we have produced an unobtrusive error reporting mechanism.

Once everything is setup, standard IO flows through the TCP socket while other communication with the Millipede Debugger is accomplished with PVM. PVM is well suited for the communication of events due to its packet oriented
interface. Each event we wish to communicate can simply be placed in one message. We must be slightly cautious however. Since we are using PVM to communicate, we must avoid interfering with the message buffer state of a process. This is accomplished by performing all communication exclusively with \texttt{pvm_precv} and \texttt{pvm_psend}, thus avoiding the usage of message buffers entirely.

Messages are divided into several different categories (requests for a new message number, log information for a successful call, requesting whether a breakpoint has been set for a call, etc.). The category to which a message belongs is distinguished by the message’s tag. In order to easily differentiate messages from different sessions, the processes in each session are given a range of tags they can use (currently Millipede uses five different tags, but the range each process is assigned is twenty-five wide, allowing for future expansion). For example, processes belonging to the first debugging session are allowed to use tags 1 - 25, those in the second session use 26 - 50, and so on. As a result of this convention, the debugger can use the range each message’s tag falls into to determine which session a message belongs to, and the tag’s actual value (modulo 25) to determine the message type.

Since many messages are exchanged between the spawned processes and the Millipede Debugger (if all options are enabled, between three and five messages are exchanged per PVM call), it would be prudent to minimize the amount of data transferred. We accomplish this by using a binary (as opposed to textual) format. We account for differing process architecture by carefully ensuring that all data is sent in big endian order and reverted to the receiving machine’s native order upon receipt. This scheme is particularly simple with Java, because it
reads and writes all data in big endian order by default, regardless of processor type.

To further reduce the overhead of running Millipede, all insertions to the SQL tables are performed by the processes themselves (using the libpq [LBPQ] library to interface with PostgreSQL). This distributes the burden of potentially costly IO operations with the SQL database over the individual processes rather than delaying the Millipede Debugger (with which the user is most likely heavily interacting) with unnecessary work.

Finally, we will consider a few aspects of replaying a process from a previously generated log file. The log file consists purely of the raw data transferred in messages marked with the log tag; making the process of reading the results of a PVM call the opposite of writing it. When running from a log file, the program will proceed as normal, but whenever a PVM call is made, the call is not actually executed, but instead, the results are read from the log file. In order for this to proceed successfully, the program must execute without nondeterminism in communication. Call types are compared (i.e., if the third call a process makes is to `pvm_send`, then the third call in the log file should be a call to `pvm_send`), as well as their parameters (except for pointers used as out parameters). Failure to match results in execution halting, otherwise the same result is returned and any changes visible to user code (e.g., the unpacking of variables into an array) are performed.

The actual process of informing the process that it should read from a log file is a little involved however. Since initialization of Millipede occurs before `main`, the obvious route of reading parameters from `argc` and `argv` is not possible. Instead we rely on environment variables to communicate this information, as
they can be read before main. If the user defines the variable ‘MILLIPEDE_LOG_MODE_FILE’ at startup, then the process will skip connecting to PVM and instead proceed by attempting to read the file specified by the variable. Other environment variables allow the process to be given a log file via a TCP socket. These variables aren’t intended to be exploited by the user, but are instead used by the Millipede Debugger to replay execution of a process on the machine an examined process was originally spawned on. This is done in order to provide an automated way to replay the execution of a process in an environment that is as close as possible to the original. We note that before control is returned to user code, any Millipede related environment variables are unset, so that any processes launched with fork and exec, do not have their environment polluted by them.

In order to facilitate the launching of a process in log mode without setting the environment in the user’s shell (and thus inviting the error of inadvertently launching subsequent processes in log reading mode if the user forgets to unset the variable), a seven line Python [PYTH] script (shown in Figure 28) is provided by the Millipede distribution. Python is used instead of a shell script because it provides easier and safer mechanisms for reading and passing parameters that contain whitespace. This script takes the log file as its second parameter (the first being the location of the script), the process to launch as its third, and all arguments to that process as the following parameters. The first few lines of Figure 11 demonstrate how this script can be used to launch a process under gdb.
#!/usr/bin/python

import os
import sys

if len(sys.argv) < 3:
    sys.exit('Not enough arguments')

os.environ['MILLIPEDE_LOG_MODE_FILE'] = sys.argv[1]
os.execlp(sys.argv[2], *sys.argv[2:])

Figure 28: The Python script to run a process from a log file.

The Millipede Debugger

The Millipede Debugger is written in Java, and is much more complex in implementation than the native library. The application is multithreaded, using the following four primary threads when fully setup and configured:

1. The standard IO thread. This thread handles communication via standard IO with all processes.

2. The PVM thread. PVM is not thread safe. Therefore, we execute all PVM calls on a single thread.

3. The connection listening thread. This thread listens for incoming TCP connections from newly spawned processes.

4. The Swing [SWNG] event thread. The Millipede Debugger uses the Swing libraries to provide a GUI. Swing is not thread safe, therefore all events effecting the display must occur on the event thread. Moreover, almost all computations and message analysis is performed on this thread.
These threads essentially work together through a combination of the popular observer and asynchronous processing concurrency patterns [PINJ]. The event thread is registered as an observer to the other three threads. These threads are only concerned with handling IO and interact directly with the participating processes. When events on these IO threads happen (e.g., a PVM message is received, a new TCP connection is formed, a process has written something to its standard IO streams, etc.), a notification of the event is posted to the event thread and placed in a queue. The event thread will asynchronously retrieve and process the event at a later time. Conversely, if the event thread needs to interact with an aspect managed by one of the three others (e.g., sending a message via PVM), the IO threads have a similar mechanism where they can be given jobs to run.

Figure 29 illustrates this multithreaded structure and how it interacts with the other processes. The user’s processes interact only with the three IO threads. Processes one and two have already established a connection and interact through PVM and standard IO. Process three is in the process of establishing a connection with the Millipede Debugger, and is thus only interacting with the connection thread. The IO threads are oblivious to each other and only interact with the event thread, which both acts as an observer to incoming events and posts tasks to the other threads as necessary.

We note, the Millipede Debugger is designed so that no mutable structures are shared between threads. This enables us to eliminate almost all locking and synchronization beyond the actual posting and retrieval of events and tasks from a thread’s queue. Application of this strategy allowed us to avoid many of the complications encountered in the creation of thread safe structures.
We have to be careful when working with the event thread however. Any long computations made on it can cause the program to appear to freeze. This happens because user interface events cannot be processed while other computations are being performed in the event thread. Fortunately, the computations required for processing an individual event posted from another thread (such as a successful execution of a PVM call by an examined process) is quite miniscule, generally requiring more computational effort to place its results on the screen (i.e., drawing the necessary text or pictures that represent the
result of an event) than actually analyzing it. We are able amortize the cost of the more complex calculations, such as deadlock detection, over multiple events.

By performing most of the processing in the event thread, we also have an easy solution to the often immense difficulty of safely shutting down a multithreaded application. When we receive a signal (such as SIGTERM) or the user issues the command to quit, we simply insert our cleanup code into the event thread’s queue. After the event thread has run this code, the thread will shutdown (cease processing events) and the rest of the process will halt. Our only requirements for this strategy to succeed are that the other threads cannot set any state that will be saved, and that at the end of processing any event in the event thread’s queue, the process will be in a state where it is safe to quit and save; that is, after processing an entry in the event queue, nothing can be set to a partial state that will be fixed up by a later event.

A particularly complicated issue arises when working with the standard IO of an unknown (and potentially buggy) user process. When writing to a process over a TCP socket (or even through standard pipes), buffers may fill up, causing a write to block. Since the details of the receiving process are unknown, we do not know whether it will perform a read in the future and thus we may potentially block forever. For this reason, we cannot perform writes to standard IO on the event thread as it could lockup the program. Instead, writes are posted to the standard IO thread.

The standard IO thread exploits the nonblocking reads and writes along with the channel selector functionality introduced with the NIO package of Java 1.4 [JV14]. The selector functionality combined with nonblocking reads allows this single thread to monitor for and respond to any incoming messages from multiple
socket connections simultaneously. Upon notification that it can read from a channel without blocking, the standard IO thread will read as much as it can and post the results to the event thread. Nonblocking writes combined with selector functionality allow us to queue up messages and only write them (or as much of them) when possible to do so without blocking. The selector supports this capability by indicating when it is safe to write again. Previous versions of Java only supported blocking reads and writes, thus requiring a separate thread to read from each individual socket, and an additional thread when asked to write if we wish to deal with the fact that a write may block indefinitely. The solution provided by Java 1.4, allows us to both monitor multiple channels simultaneously and safely write messages without blocking from a single thread, a much cleaner and efficient solution.

The PVM thread functions similarly to the standard IO thread. It essentially spins in a loop, receiving PVM messages. However, since PVM is not thread safe, we cannot send a message while we are trying to receive a message. We ensure that any other PVM actions are performed safely by placing them in a queue which is cleared out after every receive call. Unfortunately, this scheme introduces a dilemma. We cannot call pvm_recv, because we do not know if we will even receive a message in the future, or how long it will take; thus, we cannot guarantee that writes will ever be performed or that they will be performed in a timely manner. We could use pvm_nrecv, but then our loop will essentially become a busy wait loop when no messages are pending. Pvm_nrecv will never block, resulting in a rapid sequence of ineffectual calls being made until a new message arrives. Thus, our solution is to compromise with the timeout mechanism provided by pvm_trecv. In this fashion, we can ensure that calls will
block for some time if no messages are available (thus we avoid wasting processor
time), while still being able to guarantee that messages pending to be sent will be
processed. Moreover, by adjusting the timeout value, we can ensure that
messages will be processed and sent in a timely manner.

Besides interacting with a user’s processes, Millipede sports the ability to
launch and interact with processes (such as Gdb) on other computers. PVM
requires that all involved computers be able connect to each other via rsh or ssh
without the need for the user to be involved with the login process (this is
generally accomplished through the use of a trusted hosts file or through the
usage of RSA keys to avoid passwords). Ssh and rsh can be used not only to
login and start a new shell, but also to launch remote processes. For example,
the command: ‘ssh donald-duck.cs.unlv.edu gdb -pid 512’, would run gdb
on the computer donald-duck.cs.unlv.edu and have it attach to the process (on
donald-duck) with a pid of 512. We note that these parameters and more could
be retrieved from the process metadata as shown in Figure 25. Standard IO from
the gdb process is routed through ssh to the local computer, where it can be read
and written through standard pipes. It is reasonable to assume that this
functionality already exists (otherwise PVM would not work), so the Millipede
Debugger merely needs to execute a command like the one shown above in order
to achieve this functionality.

Another interesting aspect arises when we consider that the Millipede
Debugger manages its own data files. This is largely done in order to abstract
the need to directly manage them from the user. These files are kept in the
invisible directory ~/.Millipede (with ~ being the user’s home directory). This
folder holds information on previous runs as well as setup information such as the
list of predefined SQL queries. One file stored here is of particular interest
however, namely the lock file. If multiple instances of the Millipede Debugger
were run at once, we run the risk of both trying to use or modify files in this
directory at the same time. For instance, it would be very dangerous if one
Millipede Debugger process was writing the data files for a run, while another
was trying to delete them. Additional problems arise not only in the file system,
but also with the sharing of information that should be global if multiple
debugger processes were running. An example would be if one process adds or
removes a predefined query. If this situation happened, we would be faced with
the challenge of how to get this change reflected in the others.

We avoid these problems by using a lock file that prevents multiple instances
of the program from executing under the same user. Traditionally, programs
prevent multiple instances of themselves from being executed by atomically
creating a file (perhaps with the system call \texttt{mktemp} or its derivatives). If the file
fails to be created (generally because it already exists), then the process assumes
that there is a duplicate process running and quits. This technique has the
severe flaw that if a process is terminated abnormally (such as by receiving a
SIGKILL signal or by the computer losing power), it may not delete the log file;
thus preventing processes in the future from executing until the user removes it
himself (PVM and PostgreSQL both suffer from this deficiency). We avoid this
by not simply trying to atomically create a lock file (although the file will be
created if it does not exist), but by using Java’s file locking mechanism (which is
again provided by the NIO additions of Java 1.4). With this mechanism, files
can be marked as being in use by other processes (i.e., locked); thus if a Millipede
Debugger process finds the file locked, it knows that it cannot safely proceed and
quits. The primary advantage of using the file locking mechanism is that the file is always unlocked at process termination, even if the owning process was terminated abnormally. We note that there should in fact, be no need for a single user to run concurrent debugger processes. Millipede can handle multiple differently configured online and offline runs simultaneously with a single debugger process.

Next, we will briefly consider the implementation of one of the more interesting predefined SQL queries. Figure 30 shows the SQL implementation of the `match` query, which attempts to match packs and unpacks for a specified message number.

We first note that in order to allow our predefined queries to be called with parameters, they are created as prepared statements. The single parameter of the `match` query is referred to by the identifier ‘$1’ inside the query.

The `select` portion of the query extracts the data created by the `from` portion of the query. We note that even though the column names in the provided tables are short for ease of typing, since we expect `match` will be used frequently, we take the time to rename the columns in order to give more descriptive results. Generation of the `Match` column (containing boolean values indicating whether a pack and unpack match) is fairly involved. We consider a pack and unpack to be matched if they have the same type, length, and occurred in the same message positions (e.g., if three integers were packed as the fourth pack call of a message, then we have a match if three integers are unpacked by the fourth unpack call of a message). Since `match` finds both missing packs and unpacks, we expect that some of the columns we examine in order to determine this may be null. This can interfere with the calculation of the `Match` column. If
PREPARE match (int) as

SELECT
    coalesce((temp.no = rm.no AND temp.type = rm.type AND temp.scount = rm.rcount), false) AS "Match",
    $1 as "Message ",
    s.stid as "Sender Tid",
    rtid as "Receiver Tid",
    temp.no as "Pack ",
    rm.no as "Unpack ",
    temp.type as "Pack Type",
    rm.type as "Unpack Type",
    temp.varname as "Pack Var",
    rm.varname as "Unpack Var",
    temp.scount as "Items Packed",
    rm.rcount as "Items Unpacked"
FROM
    (select * from receivedmessages where msgno = $1) as rm
FULL JOIN
    (select distinct
        senders.rtid,
        no,
        type,
        varname,
        scount
    from senders, sentmessages
    where senders.msgno = sentmessages.msgno AND
    senders.msgno = $1)
    AS temp using (rtid, no),
    (select distinct
        stid
    from senders
    where msgno = $1) as s
ORDER BY rtid, temp.no, rm.no

Figure 30: The match query implementation.
one of the values it examines is null, the answer will not be false, but null instead. We use the `coalesce` function that PostgreSQL provides to avoid this problem. This function returns the first non-null value of its arguments, allowing us to substitute false if the match evaluation returns null.

The `FROM` portion of the query is much more complex. The basic idea is that we wish to perform a full outer join between the information concerning the pack and unpack calls of a single message (i.e., between the `sentmessages` and `receivedmessages` tables). By using a full outer join, we can find any missing packs or unpacks (these are the rows that contain null values), as well as compare the pack and unpack portions of calls that do have counterparts. Unfortunately, the simplicity of this plan is shattered by the fact that multicast sends are allowed. To solve this problem we create a table (called `temp`) that essentially performs a cross join between `senders` and `sentmessages`, resulting in a table that joins and repeats the pack information for each receiver of a message. The full outer join is performed between `temp` and a table derived from `receivedmessages` (called `rm`) that only contains the unpack information regarding the message number we are examining. These tables are joined by their receiver tids, and pack/unpack number; basically matching the pack information for each receiver in `temp` with the corresponding unpack in `rm`. Since we may have an unpack that doesn’t have a corresponding pack (i.e., we may have rows where we have an unpack indicated by `rm`, but no data is available in `temp`), we cannot rely on `temp` to provide us with the sender tid. Thus, we cross join the resulting table with the single sender tid value we can retrieve from the `senders` table (this works because every message has one and only one sender).

`Unmatched_pk_upks` operates with a very similar query, differing only in that it
performs the analysis over all message numbers and it only shows rows where \textbf{Match} would evaluate to false.

Finally, we will consider the rationale behind the primary method of navigating the Millipede Debugger, tabbed panes. As stated, one our primary goals was to avoid overwhelming the user with too many windows. This is a typical problem experienced by users of graphical tools, who often complain of “popping up windows I don’t need in places I don’t like” [PAN3, p. 48]. Tabbed based navigation of a single window is the paradigm we decided to provide in order to solve this problem. Tabbed based navigation also serves another purpose by providing a hierarchical view of the data in order to avoid distracting the user with extraneous information.

To understand what the implications of this hierarchical view, we must consider how the user typically works with a debugging application. Research suggests [PAN3] that a typical user workflow starts by examining the overall state of the program for anomalies. Once an anomaly is detected, the user will then wish to zoom in and examine the problem at a lower level. The Millipede debugger has been designed around this workflow. It is intended for the user to view and retrieve global information with the tools provided at the protocol and message passing levels (which will often automatically flag anomalous behavior for the user), and once an anomaly has been detected, to examine the location in detail with the sequential and source code level tools. By separating disparate domains of analysis with the tabs, the Millipede Debugger allows the user to view the state of program with the perspective he needs, without confusing the user with extraneous information.
CHAPTER 7

CONCLUSION AND FUTURE WORK

In this thesis, we discussed Millipede, a tool for debugging distributed programs, using a bottom-up approach known as multilevel debugging. By analyzing the flaws of previous debugging efforts, and using the multilevel debugging paradigm as our design philosophy, we believe that Millipede does an effective job of filling the distributed debugging niche. Millipede provides the necessary abstractions and granularity to handle the complexity inherently found in these systems, while the analysis of its predecessors allows us to avoid many of their flaws.

Improvements upon Previous Tools

One of the primary goals of the Millipede project was to avoid or correct several of the flaws of previously developed parallel debugging tools. We shall proceed by reexamining the issues with previous tools that were identified in Chapter 2 and examine how we have applied Millipede and the multilevel debugging approach to alleviate them.
• The perspective they provide is too narrow:
  ○ Previous tools focused on either individual processes or the overall system at the expense of its counterpart. Millipede provides abundant tools to analyze both aspects of a distributed system, allowing the user to debug at the correct level.

• They overwhelm the user with information or do not provide enough:
  ○ By dividing the error domain into three separate parts we prevent extraneous information from clouding the analysis of a bug.
  ○ Simplified and advanced views of the data extracted at each level are provided so that the user can deal with simple problems quickly, but has access to more complex information should the need arise.
  ○ Tools at all levels of analysis provide the ability to map the data back to the source code that produced it.

• They lack the lack flexibility to allow the user to find the data he needs:
  ○ A powerful query mechanism is provided by the SQL database of communication information. This allows the user to create and customize his own views of complex global information if the provided tools are not sufficient for the task at hand.
• The user still needs to interpret the data in order to find the cause of an error:

  ☐ Automated tools are provided at the protocol level that can detect protocol violations and deadlock.

  ☐ The predefined queries into the SQL database include two that can detect and isolate messages that have not been received and data the has not been unpacked or unpacked incorrectly; two common classes of error in PVM and parallel message passing systems in general.

Millipede takes into account the distinctive properties of message passing systems, allowing the user to debug at the sequential, message passing, and protocol levels. It is powerful enough to provide fine-grained and low level information, but does so with appropriate abstractions that avoid overwhelming the user with irrelevant information. The user is given the flexibility to create his own queries of extensive global information, a necessity when working with a distributed system; allowing him to create his own views so that he may quickly discern the information necessary to complete the parallel debugging task.

Future Work

I would like to expand the capabilities of Millipede and employ the multilevel debugging technique to other distributed systems beyond PVM, most notably the popular MPI message passing system [MPI]. Currently, there is a project in progress to implement Millipede for LAM/MPI.
There is an algorithm that can suggest corrections for a deadlock induced state given a protocol specification which can be found in [CORE]. Inclusion of this algorithm could begin to extend the automated detection of errors to the automated correction of errors.

Finally, it would interesting to strength our protocol analysis with the incorporation of temporal and pattern information as well as enabling the checking of message format. The current implementation only checks to see if a rule is satisfied, but doesn’t allow the user to specify in what order the rules should occur. Given protocol rules A, B, C, and D, we could specify a temporal constraint in a simple regular expression like language. The constraint A (B|C)* D would mean that a process’ first send should satisfy rule A, followed by sending zero or more messages satisfying B or C, after which the process must end with a message satisfying rule D. We could also use a similar regular expression scheme to define message format. In this manner, we can verify that not only the sends and receives of a message passing system are correct, but also take a step towards automatically verifying that the contents of these messages are correct as well.
REFERENCES


http://gcc.gnu.org/


Tutorial for Networked Parallel Computing*, Prentice Hall International,
1994.

[GUI] M. Rayterberg. “An empirical comparison of menu-selection (CUI) and
desktop (GUI) computer programs carried out by beginners and


[JPVM] jPVM: A native methods interface to PVM for the Java Platform.
http://www.chmsr.gatech.edu/jPVM/

java.sun.com/j2se/1.4/index.jsp


[LBPQ] PostgreSQL: Documentation: Manuals: PostgreSQL 8: libpq - C


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


         “VAMPIR: Visualization and Analysis of MPI Resources”,

[XDR]   RFC 1014 (rfc1014) - XDR: External Data Representation standard.

         1.1”, Proceedings of the 29th Annual Hawaii International Conference
         on System Sciences, 1996, pp. 290-299.
VITA

Graduate College
University of Nevada, Las Vegas

Erik Helge Tribou

Local Address:
  4801 Spencer St. Apt. 190
  Las Vegas, Nevada 89119

Home Address:
  10641 NE 68th St
  Kirkland, WA 98033

Degrees:
  Bachelor of Science, Computer Science, 2002
  University of Washington, Seattle, Washington

Thesis Title: Multilevel Debugging and Millipede: A Novel Approach to Debugging Distributed Systems

Thesis Examination Committee:
  Chairperson, Dr. Jan Pedersen, Ph. D.
  Committee Member, Dr. Thomas Nartker, Ph. D.
  Committee Member, Dr. Kazem Taghva, Ph. D.
  Graduate Faculty Representative, Dr. Rama Venkat, Ph. D.