Sequential debugging of parallel message passing programs

Jan Bækgaard Pedersen  
Department of Computer Science  
University of British Columbia  
Vancouver, BC, Canada, V6T 1Z4

Alan Wagner  
Department of Computer Science  
University of British Columbia  
Vancouver, BC, Canada, V6T 1Z4

Abstract

Millipede (Multi Level Interactive Parallel Debugger) is a support environment developed to assist programmers to debug message passing programs. Millipede is a collection of tools designed to help the user with specific debugging problems. In this paper we focus on a tool to support the use of a sequential debugging environment on a parallel program. This tool extracts the code and messages necessary for a process to execute sequentially and allows existing sequential debugging tools to assist in debugging parallel code.

Keywords: Parallel debugging, Programming tools, Message passing debugging

1 Introduction

Debugging is a challenging problem, especially in a parallel environment where the user must reason about the correctness of an asynchronously executing set of processes loosely coordinated by the passing of messages. Studies have show that users typically spend the same amount of time debugging their parallel code as they do in initially writing it [8]. Debugging is an important part of the code development process and it is important to provide tools to reduce this time.

In this paper we describe Millipede a collection of related debugging tools for PVM and MPI currently being developed at the University of British Columbia. We focus on the Sequential Debugging Module of Millipede which allows the programmer to extract a sequential process from a parallel program and debug it using any sequential debugging tool.

In Section 2 we discuss the motivation and design goals of Millipede. In Section 3 we describe Millipede and how it is used. Section 4 illustrates the use of Millipede to debug some common errors that can occur. Section 5 discusses the implementation of Millipede and Section 6 contains the conclusions.

2 Motivation and Related Work

In general tools have not been widely adopted [2, 6, 7]. It has been estimated that between 35% to 90% of parallel programmers rely on print statements as their main debugging and performance tool [8, 2]. There are several reasons why tools have not been adopted, these include pragmatic concerns, the nature of the debugging process, lack of focus and information overload.

There are variety of tools such as DIWIDE [5] and TotalView [3] that extend the traditional sequential debuggers to the parallel environment. These tools however still require users to learn a new GUI and do not scale well since users have to individually control the execution of every process and thread in the system. DIWIDE does provide some higher level controls to allow for collective breakpointing and macro-stepping.

There are tools available (e.g., pdbx [9] and p2d2 [4]) with simple GUI interfaces that essentially spawn a sequential tool for each process in the system. However, N-version debuggers do not scale well and suffer from both information overload and lack of focus. It is difficult in these systems for the user to focus on the one process that may be causing the problem.

Finally, there are a variety of replay or animation tools to support both correctness and performance debugging of parallel programs. Two examples of these tools are BUSTER [11], a post-mortem replay system, and PVaniM [10] a post-mortem replay and visualization tool for PVM. These tools suffer from a lack of focus due to the difficulty of mapping the incorrect behaviour back to the error in the code.
causing the error.

In addition to these problems, there are pragmatic concerns with respect to learning new GUIs, the use of unfamiliar tools, the relative instability of current parallel runtime environments, and the difficulty of working with tools which are either too restrictive or too general.

The design of Millipede follows a divide and conquer approach to parallel debugging. We have implemented a suite of tools each tailored to a different stage in the parallel code development process. We assume a development process that consists first of the design of the sequential code, the addition of message-passing and the decomposition of the data, the testing of the program on smaller configurations, and finally the scaling up of the application to execute on the target configuration. This paper discusses the first stage, supporting the sequential debugging process.

It has been estimated that 31% of the time users develop the parallel program from scratch [8]. Given the amount of code that is sequential it is reasonable to try to debug it as much as possible in a familiar sequential environment. Millipede allows the user to extract the source code and message history of a single process to allow it to be debugged using familiar sequential debugging tools. In the 49% of the time when the program is transformed from correct sequential code it is necessary to decompose data structures and add the message-passing. The transformation of the code to parallel code can introduce errors in the sequential parts of the program. For example, decomposing an array across a collection of machines may cause illegal loop accesses. Again, in this situation, these errors can be found by sequential tools designed to find memory leaks, illegal pointer references, and access to uninitialized data. In the remaining cases the parallel program is a modification of an existing program or built from libraries. A sequential debugger may be of use in these situations as well.

Supporting sequential debugging is a pragmatic approach to allow users to leverage their existing expertise and should be supported and exploited as much as possible. Existing tools do not make it easy for the user to extract a process, and even replay tools do not extract an executable process which can run as a stand-alone program. We describe the Sequential Debugging Module of Millipede along with a description of the information that needs to be extracted to execute a process as a stand-alone program. We illustrate the use of the tool to demonstrate that even when the parallel program terminates abnormally the tool captures sufficient information to find the errors using a sequential debugger. The Sequential Debugging Module is just one of the tools in Millipede. There are other tools to be used later in the process for debugging communication mismatches and protocols.

3 Millipede Sequential Debugging

Millipede consists of a number of modules, each capable of performing debugging at a different point in the development process. (see Fig. 1) At the lowest level in Millipede is the Sequential Debugging Module which allows the programmer to individually debug each of the processes in the system in a sequential debugging environment. Once the errors in the sequential parts of the parallel code have been found there remains the possibility of communication related errors. Identifying and correcting communication errors is the main objective of the other modules in Millipede. The next level, a Message Debugging Module, allows the user to control the contents of messages sent and received in one or more processes while the entire program is executing. The higher levels are tailored towards deadlock detection and correction as well as communication protocol verification.

The Sequential Debugging Module works as follows. It first requires the user to compile and execute the program in parallel. During execution Millipede intercepts messages sent to all of the processes in the system and writes the contents of these messages to a file, one file for each process. In addition, the return values of all message passing function calls are also written to the file. The log files collected for each process stores the messages in the order they were released to the process. It is not necessary to time-stamp the messages in the system since we are concerned only with the behavior of a single process. It is however necessary to capture the return values of all communication calls as program behavior may depend on these values. For example, non-blocking receive calls can return without receiving any data, and it

<table>
<thead>
<tr>
<th>Communication Visualization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadlock Detection &amp; Correction</td>
</tr>
<tr>
<td>Comm. Protocol Verification</td>
</tr>
<tr>
<td>Message Debugging</td>
</tr>
<tr>
<td>Sequential Debugging</td>
</tr>
</tbody>
</table>

Figure 1: The Millipede levels.
is important to capture this behavior to mimic the behavior of the process when it was run in parallel.

In order to use the Millipede debugging system the run-time system must to be linked to the programs. This is done by adding a flag to the compiler call and recompiling the entire program with the `-DMILLIPEDE` option set:

```
 gcc -g -DMILLIPEDE -o Pgm Pgm.c -lpvm3
```

In order to instruct Millipede to create log files the Replay Collection Mode (REM) must be chosen. This is done by setting an environment variable called MILLIPEDE_RCM:

```
 setenv MILLIPEDE_RCM
```

This instructs Millipede to create log files labeled Millipede_RPF-xxx-yyyy where xxx is the name of the program and yyy is the process ID. Note that the message passing system must be started prior to running the program. Once the parallel execution has terminated Millipede can be switched to Replay Execution Mode (REM) and the message passing system can be closed down:

```
 unsetenv MILLIPEDE_RCM; setenv MILLIPEDE_RCM
```

Now in Replay Execution Mode the program can be run sequentially, without the message passing system and the other processes running, by simply typing in the name of the executable, or through a debugger, for example:

```
 gdb Pgm
```

The Millipede runtime library asks the user which replay file should be used to execute the program:

```
 Replay filename: MILLIPEDE_RPF-Pgm-262152
```

Sequential debugging now commences as if the program was in sequential program. The Millipede run-time library reads the log file each time a message passing call is made in the code, and thus supplies the program with values for the variables received through messages. The programmer can debug, recompile and re-execute the process with the message log until the errors have been corrected. If the programmer wishes to debug the same process with another set of messages he simply restarts the program and specifies the name of a different log file.

In order to simplify the setting and unsetting of environment variables, there is a simple GUI called the Millipede Control Panel Program (MCP1P) that can be used to accomplish the same tasks.

### 4 Examples

The examples shown in this section are used to illustrate the types of errors that can easily be found and corrected by using Millipede. It shows that the errors occurred in the parallel execution were faithfully reproduced in the sequential execution of the process containing the error. As a basis of these tests we used a master/slave implementation of an iterative hyperbolic differential equation solver which we seeded with errors. Figure 2 and 3 show the relevant parts of the slave program containing the errors.

#### Parallel code:

```c
 pvm_upkint(&nproc, 1, 1);
 pvm_upkint(tids, nproc, 1);
 pvm_upkint(&n, 1, 1);
 e = n%nproc;
```

#### Tool: GDB

```
 (gdb) step
 45 e = n % nproc;
```

Program received signal,
SIGPFE Arithmetic exception 0xe4a86e8
in ()

![Figure 2: Division by zero.](image)

Consider the code shown in the left part of Fig. 2. The assignment causes a division by zero if `nproc` equals zero. When the program is executed in parallel the slave process executing the illegal division encounters an arithmetic exception and terminates. By stepping through the sequential program extracted my Millipede using `gdb` (the right part of Fig. 2) the error is easily located. In contrast, to find this error using `N` versions of `gdb` online the programmer would have to single step each process long enough to make sure the communication has taken place, i.e. that the process in the focus of attention can execute the receive call that leads to receive the value of the faulty `nproc` variable.

Tools that allow macro stepping (DIWIDE) or tools that allow control of all debuggers at the same time (p2d2 or TotalView) could have handled this problem as elegantly because they have im-
plemented stepping (which was the feature used in gdb). However, if the programmer had tried to use another sequential tool that had not been ported to a specialized parallel debugging tool, one would have had to use the N-version approach, if applicable at all.

The left part of Fig. 3 shows an example of a code fragment that indexes an array out of bounds. The `z` array is not large enough and is indexed from 0 to `nodes + 1`, but has only been allocated to hold `nodes` number of elements. This error can result in two different program behaviours, (a) an incorrect result, (b) a segmentation fault where the process terminates abnormally. This error is easily detected by using a tool like purify, which is tailored to examine sequential programs for memory leaks. Note that even if this program had initially been a sequential program, the error would not have been present since it was introduced when the data needed to be distributed across a number of processes, which appears only in the parallel version of the program. It is possible to attempt to use the N-version, however it is not easy to apply a tool like purify to a parallel running program, without having to display too much information. Another disadvantage is that the user must concentrate on N versions of a tool to locate and correct an error that can be found with one instance of the same tool. Finding other well known errors, such as bad pointer references and other types of memory leaks, can easily be detected in a similar manner.

In parallel programs the cause of an error and the effect of the same error can be quite far from each other in terms of both program lines and execution time. In [1] it is estimated that 15% of errors in sequential programs are caused by this cause/effect chasm. Introducing message passing, which allows erroneous data to propagate through messages from one process to another, further exacerbates this problem. The division by zero example shows how this is possible: the value of the `nproc` variable originated from a communication call, where the error was propagated from the sender of that message. The sequential debugging module of Millipede helps the localization of the cause and effect of bugs due to erroneous values being received from other processes. Once it has been determined that the bug is located in a variable received from another process, this process can be sequentially debugged in the same fashion to locate the origin of the error.

4.1 Limitations

Millipede is tailored to allow running a single sequential tool. There are classes of errors that are difficult to find using this approach.

If a parallel program contains an irreproducible error (i.e. an error that does not occur every time the program is run, and if the Millipede log files were not generated during the execution that encountered the error) the sequential debugging module of Millipede will be of no use. Millipede is also not thread safe, and cannot be used to debug multi-threaded programs with 100% confidence. In general debugging multi-threaded parallel programs introduces yet another type of concurrency that further complicates the debugging process. However, as tools for debugging threads become available, Millipede will allow us to use them in the parallel domain.

5 Implementation

In this section we focus on the existing PVM implementation of Millipede. The MPI version (which is only partly implemented) is implemented exactly the same way.

The core of the Millipede system is the redefined message passing calls. In Fig. 4 a few of these are shown. When a program is compiled with the `-DMILLIPED` option all PVM message passing calls are substituted with the equivalent Millipede message passing calls, e.g. any call to `pvm_recv` is substituted with a call to `pvm_recv`. These are the functions executing the Millipede debugging code, and their implementation is linked into the original PVM library (libpvm3.a). When a PVM program is compiled the redefinition is only included when the `MILLIPED` flag is set. When the flag is not set the program executes like a normal PVM program. This makes it easy for the user to switch between normal and debugger execution, and does not require any rewriting of the program, just recompilation.

Millipede consists of several modules each used at different levels of the development process. These modules are controlled through the setting and un-setting of environment variables. The environment variables determine which action is to be taken.

The Sequential Debugging Module consists of 2 parts:

1. The collection part that intercepts the values and names of variables being packed (calls to the `pvm_pack` functions) and return values of other message passing functions.

2. The replay part that reads the log files written
ABW: Array bounds write
This is occurring while in:

```
main [Wave_slave.c:57]
for (i=1; i<=nodes; i++)
  x[i] = (is(start+i-1))/(n-1);
```

Writing 8 bytes to 0xdc630 in the heap. Address 0xdc630 is 1 byte past end of a malloc’d block at 0xdc5a8 of 136 bytes. This block was allocated from:

```
malloc [rtlib.c]
calloc [rtlib.c]
main [Wave_slave.c:50]
```

Thus

```
⇒ x = calloc(nodes,sizeof(double));
⇒ y = calloc(nodes,sizeof(double));
```

Figure 3: Memory leak that leads to a segmentation fault.

```c
#define pvm_initsend(X) _PVM_initsend(X,__FILE__,__LINE__)
#define pvm_recv(X,Y) _PVM_recv(X,Y,__FILE__,__LINE__)
#define pvm_upkint(X,Y,Z) _PVM_upkint(#X,Y,Z,__FILE__,__LINE__)
#define pvm_pkint(X,Y,Z) _PVM_pkint(#X,Y,Z,__FILE__,__LINE__)
```

Figure 4: Redefined PVM functions

in the collection part, when pvm_upk* functions are called.

The collection part is straightforward. When packing functions are called the names and the values of the variables are written to the corresponding log file for the process. All other message passing functions also write their return values to the log file in order to assure during a replay that an exact copy of the original program is executed.

During replay, the log files are read, i.e. instead of performing a call to the network for a packet of data the log files are read, and the values are returned to the caller as if a message had arrived from the network. The majority of the code for this part of the tool is for checking that values are placed into the right variables. This is done by comparing the names of the variables in the pvm_upk* call with the names of the variables in the log file. If a mismatch between variable names is found, Millipede automatically prompts the user for a value for the variable that was unsuccessfully unpacked.

At present, the sequential debugging module consists of approximately 5-20 lines of code for each PVM function.

6 Conclusion

We have presented a module of Millipede to allow a programmer of a parallel message passing program to debug a process sequentially without having to worry about message passing (The message passing system does not need to be running, and nor do the rest of the processes of the parallel program). Any sequential debugging or analyzing tool can be applied to a parallel process, thus making debugging parallel programs as easy as debugging ordinary sequential programs. The advantage of this approach is that it allows users to use familiar sequential debugging tools to debug their parallel programs.

As stated earlier the Sequential Debugging Module is used to detect errors in the sequential code of a parallel message passing program. However this is not the only place that an error can occur in a parallel program. Corrupt message data, deadlocks,
even protocol errors are examples of other errors that can be found in parallel programs. Millipede is an ongoing project and we are currently working on the other modules shown in Fig. 1. These modules will attempt to address errors in the communication between processes.

We are currently working on an Interactive Message Debugging module which allows the user to change the contents of messages at run-time, without having to use a debugger like gdb. Deadlock correction is another possible module of the Millipede system, which is currently being investigated. In the case of a deadlock arises the Millipede deadlock module will analyze the remaining messages in the system, including posted receive requests, and attempt to suggest a possible cause for the deadlock and a way to fix communication mismatches, errors in the actual parameters in send and receive calls.

Finally, a module for automatic communication protocol verification is being developed. This allows the programmer to use Millipede to check that messages flow in the system according to a protocol specification. This specification can be very general in the beginning, and once it has been verified, it can be refined to specify further details of the protocol. In this case, the checking takes place at run time so that Millipede can inform the programmer which of the messages (content, sender program name/line, and receiver program name/line) violated the protocol.

References


