Multi Level Debugging of Parallel Message Passing Programs

A different approach to debugging parallel programs

Jan Bækgaard Pedersen

University of British Columbia

matt@cs.ubc.ca

February 13, 2001
Contents

1 The Problem 5
   1.1 Rationale, Significance, and Need for the Study 5
      1.1.1 The Parallel Programming Domain 5
   1.2 Statement of the Problem 8
      1.2.1 Top-down versus bottom-up debugging 13
      1.2.2 The Purpose of This Research 14
      1.2.3 Error Classification 15
      1.2.4 Tool development 16
      1.2.5 Automation 18
      1.2.6 Existing Approaches 19
      1.2.7 Thesis Statement 20
   1.3 Hypotheses, Theories, and Research Questions 20

2 Millipede 22
   2.1 Design 23
   2.2 Implementation 24
      2.2.1 Wrapper functions 24
      2.2.2 The runtime system 25
      2.2.3 Current Status of Millipede 25
   2.3 Millipede Sequential Debugging Module 26
      2.3.1 Examples of Uses of the Sequential Debugging Module 28
      2.3.2 Limitations 30
2.4 Millipede Message Debugging Module

2.5 Millipede Protocol Debugging Modules
   2.5.1 Millipede Deadlock Correction Module
   2.5.2 Millipede Online Protocol Error Detection Module

2.6 Expected Future Publications

3 Review of the Literature
   3.1 Program Development Environments
   3.2 Visualisation Tools
   3.3 Extension of Sequential Debuggers
      3.3.1 Debugging Environments
      3.3.2 N-version Debuggers
   3.4 Replay Tools/Debuggers
   3.5 Relative Debuggers
   3.6 Language Support for Communication
   3.7 Summary of Related Work

4 Plan for completing this work
Introduction

Within the last, decade distributed computing has become readily available through clusters of high performance commodity processors and high speed networks, such as gigabit Ethernets and Myrinets. Although the hardware is now accessible the difficult problem remains of designing and implementing programs to run in parallel.

To exploit parallelism on such systems programmers often use message passing to allow communication between processes executing on different physical processors. Two of the best known message passing systems are PVM (Parallel Virtual Machine) [ea94] and MPI (Message Passing Interface) [Don94].

When developing, updating or maintaining software, the need for debugging faulty programs arises. The debugging task, well known from the sequential world, is significant and unavoidable. It becomes even more more complicated 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. The complexity of the debugging task increases, as errors can occur in not only the sequential parts of the program, but also in the message passing calls or in the overall communication protocol. In addition, errors propagate easily from one process to another through message passing calls, thus making locating the source of the error extremely complicated [Eis97].

Studies have shown that users typically spend the same amount of time debugging their parallel code as they do initially writing it [Pan94]. This suggests that debugging is not only an important task but also a very time consuming one. It is also estimated that between 35% and 90% of parallel programmers rely on print statements as their main debugging and performance tool [Pan94, ea93, Eis97].

Although there has been a considerable amount of work done in the area of parallel debugging, the reluctance of users to adopt special environments to construct and debug parallel code remains a persistent problem. According to a number of studies done by Cheri Pancake, many of the existing debugging tools and programming environments often suffer from the following [ea93, Pan93a, Pan93b]:

- Both tools and environments are too restrictive and they give the user a feeling of working against the environment.
They are too general and too difficult to learn and adapt to the particular situation at hand.

These factors complicate the debugging task as the programmer easily gets confused and loses track of information, especially when using advanced systems. This confusion can be caused by the overwhelming amount of information presented by a tool. Often a tool offers multiple windows and views all presenting information in different ways. Unfortunately, most do not allow the user to customize views to suit his needs, thus making the extraction of information cumbersome and time consuming [Pan99]. Since it can be hard to extract the exact information needed to complete the debugging task at hand the user is often left with too much, potentially useless, information. I will refer to this concept as “information overloading” in a tool. This information overload often result in the programmer reverting to the tried and true debugging-by-print-statements method.

I suggest a different approach to parallel message-passing debugging: multi level debugging. Multi level debugging applies different techniques to different parts of the parallel system, depending on the hypothesis that the user is trying to verify, and the type and location of the error. For example, when looking for sequential errors, the programmer should not be concerned with the message passing part of the system when the error is purely sequential. This allows the programmer to concentrate on the specific type of error/bug she is looking for, and thus avoids the information overload often found when using other tools. Within such a system the debugging task should be easier, more accessible and faster.

Millipede is a prototype of a multi level debugging environment. It consists of a number of different modules, each tailored to assist the programmer to debug his message passing program at various levels.
Chapter 1

The Problem

1.1 Rationale, Significance, and Need for the Study

Debugging sequential programs can be a tedious and time consuming task. Time spent can be greatly reduced by using some of the many different tools developed for this task. Some of the more well known debugging tools include gdb, dbx, purify and various integrated development environments accompanying programming languages. Unfortunately, these tools are not as readily available in the parallel programming domain.

To better understand the lack of tools and the limited use of existing tools, the next section briefly introduces some of the problems encountered when working in the parallel programming domain.

1.1.1 The Parallel Programming Domain

Parallel programming involves a set of components that each must be considered when developing a parallel system. This set, which I regard as the parallel programming domain, includes program source code, interprocess communication, synchronisation, processor utilisation among others. Understanding the issues involved with the components of this domain makes understanding the source and manifestation of errors easier. This understanding is useful for determining the approach needed to efficiently debug parallel programs and for determining where to focus effort, depending in which component of the domain the programmer looks for errors.
In [Fos95] PCAM, a 4 stage model representing the parallel programming domain is suggested. The 4 components are these:

1. **Partitioning.** The computation to be performed and the data which it will operated on are decomposed into small tasks.

2. **Communication.** The communication required to coordinate task execution is determined, and appropriate communication structures and algorithms are defined.

3. **Agglomeration.** The task and communication structures defined in the first two stages of a design are evaluated with respect to performance requirements and implementation costs.

4. **Mapping.** Each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs.

The two last components, agglomeration and mapping, are mostly concerned with performance issues which, while important, are beyond the scope of this thesis.

For the first two components, partitioning and communication, I propose the following breakdown:

1. **Algorithmic changes.** David Gelernter once noted, “Most parallel programs begin life as a sequential program.” This means that parallel algorithms are often based on and derived from existing algorithms and/or programs. A transformation from the sequential to the parallel domain therefore must occur. The transformation of a sequential program into a parallel program typically consists of inserting message passing calls into the code and changing the existing data layout; for example shrinking the size of arrays as data is distributed over a number of processes.

   However, if the sequential algorithm is not suitable for parallel implementation, a new algorithm must be developed. A well known example of this problem is matrix multiplication: the pipe-and-roll version \([FJL^+88]\) does not have a sequential counterpart.

2. **Data decomposition.** When a program is re-implemented with new code it is often due to the fact that data has been distributed in a certain way among the processes involved in
the parallel program. Data decomposition is a non-trivial task that cannot be ignored when writing parallel programs, as not only correctness but also efficiency greatly depends on it.

3. **Data exchange.** As parallel programs consist of a number of concurrently executing processes, the need to exchange data inevitably arises. This problem does not exist in the sequential world of programming where all the data is available in the process running the sequential program. However, in parallel programs, the need for data exchange is present. On a shared memory machine, the data can be read directly from memory by any process. There is still the problem of synchronized access to shared data to consider, but no sending and receiving of data is needed. When working with a cluster of processors, each having separate memory, message passing becomes necessary.

When message passing systems like PVM and MPI are used, the programmer is responsible for a number of different tasks: specifying the correct IDs of the involved processes, packing message buffers, using the correct functions to pack the data depending on the type, assigning tags to the message and more.

Data exchange is concerned with the point to point communication of data between two processes and not the overall communication structure of the entire program. Thus for every data exchange there is one send operation and at least one receive (if broadcast or multicast is used there can be multiple receivers).

4. **Protocol specification.** The protocol for a parallel system is defined as the contents, order and overall structure of the message passing between communicating processes. Along with data exchange, the communication protocol of the program is a new concept that has been introduced by parallelising the algorithm.

Figure 1.1 shows a stylized representation of a sequential and a parallel program. As shown, a sequential program is depicted as a single box, representing the sequential code of the program. The parallel program is represented as a number of boxes each consisting of three nested boxes. The innermost of these boxes represents the sequential program that each process in the parallel program executes. The sequential code of the parallel program can be either an adaption of the existing sequential program, or a completely rewritten piece of code. The middle box represents the
messages being sent and received in the system (the data exchange), and the outer box represents the protocol that the communicating processes must adhere to.

### 1.2 Statement of the Problem

In this section I will introduce the debugging problem, briefly present ideas about how to debug in general, describe the problems with current approaches, look at the purpose of my research and explain how it differs from existing systems.

A well known approach to debugging was proposed by Araki, Furukawa and Cheng [AFC91]. They describe debugging as an iterated process of developing hypotheses and verifying or refuting them.

They propose a 4 step iterative process as follows:

1. **Initial hypothesis set.** The programmer hypothesises about the errors in the program, including the places in the program where errors are supposed to occur, their causes, behaviour, and modifications needed to correct them.

2. **Hypothesis set modification.** As the debugging task progresses the hypothesis changes through the generation of new hypotheses, refinement and the authentication of existing ones.
3. **Hypothesis selection.** Hypotheses are selected according to certain strategies, such as narrowing the search space, significance of error and more.

4. **Hypothesis verification.** The hypothesis is verified or discarded using one or more of 4 different techniques: static analysis; dynamic analysis (executing the program); semi-dynamic analysis (hand simulation and symbolic execution) and program modification.

If the errors have not been fixed after step 4, the process is repeated from step 2. In the above model, step 4, hypothesis verification, is the focus of my research.

In [Eis97] M. Eisenstadt describes a 3-dimensional space in which sequential errors are placed according to certain criteria. This classification shows some interesting results, which I will briefly summarise here:

51 programmers were asked to participate in a study in which programming errors were placed in a 3 dimensional space. The 3 dimensions are these:

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Why was the error difficult to find?</td>
</tr>
<tr>
<td>2</td>
<td>How was the error found?</td>
</tr>
<tr>
<td>3</td>
<td>What was the root cause of the error?</td>
</tr>
</tbody>
</table>

Figure 1.2: The 3 dimensions proposed for categorising an error.

I will briefly describe the results of the survey with respect to each of the dimensions.

**Dimension 1: Why is an error hard to find?**

This first dimension is concerned with the difficulty of locating the problem, and is further divided into 5 sub-categories:

1. **Cause/effect chasm.** This is often the symptom of the error if far removed in space/time from the root cause.
2. *Tools inapplicable or hampered.* This covers bugs that go away when switching on the debugging tool or tools being useless for the specific task because the granularity of the tool is not tuned to locating the type of errors the user is looking for.

3. *WYSIPIG (What You See Is Probably Illusory, Guv’nor).* A piece of code is mis-conceived; it does not give the result that is looks like it will produce.

4. *Faulty assumption/model.* The programmer does not understand the underlying system, model or the environments she is using.

5. *Spaghetti code.* The code is simply unreadable.

The 51 answers were placed in the following ways in this dimension:

<table>
<thead>
<tr>
<th>Category</th>
<th>No. of answers</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cause/effect chasm</td>
<td>15</td>
<td>29.4%</td>
</tr>
<tr>
<td>Tools inapplicable or hampered</td>
<td>13</td>
<td>23.5%</td>
</tr>
<tr>
<td>WYSIPIG</td>
<td>7</td>
<td>13.7%</td>
</tr>
<tr>
<td>Faulty assumption/model</td>
<td>6</td>
<td>11.8%</td>
</tr>
<tr>
<td>Spaghetti code</td>
<td>3</td>
<td>5.9%</td>
</tr>
<tr>
<td>No answer</td>
<td>8</td>
<td>15.7%</td>
</tr>
</tbody>
</table>

Figure 1.3: Dimension 1. Why is an error hard to find?

It is notable that over 50% of the cases are caused by the 2 first categories.

**Dimension 2: How is an error found.**

This dimension is concerned with how an error can be found, and it is divided into 4 categories:

1. *Gathering data.* The programmer finds out more using methods such as print-statements and breakpoints.

2. “*Inspection*”. This term covers inspection of the code and hand simulation.

3. *Expert recognised cliché.* The programmer received assistance from other people.
4. **Controlled experiments.** Once the cause of the error is better understood specialized tools or approaches can be applied.

The placement of the answers in this dimension can be found in Figure 1.4.

<table>
<thead>
<tr>
<th>Category</th>
<th>No. of answers</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gather data</td>
<td>27</td>
<td>53%</td>
</tr>
<tr>
<td>“Inspection”</td>
<td>13</td>
<td>25.5%</td>
</tr>
<tr>
<td>Expert recognised cliché</td>
<td>5</td>
<td>9.8%</td>
</tr>
<tr>
<td>Controlled experiments</td>
<td>4</td>
<td>7.8%</td>
</tr>
<tr>
<td>No answer</td>
<td>2</td>
<td>3.9%</td>
</tr>
</tbody>
</table>

Figure 1.4: Dimension 2. How is an error found.

An interesting but not surprising result is that data gathering (e.g. print statements) and hand simulation account for almost 78% of the techniques reported in locating errors. This result corroborates the result of Pancake [Pan94]: up to 90% of all debugging is done using print statements.

While the use of print statements is straightforward when working with sequential programs its parallel counterpart often becomes a complicated task. Often processes run on remote processors, and that makes redirecting output to the console a difficult task. Even when output can be redirected to the console, all processes will be writing to the same window, thus making the interpretation of the output a challenging task. Furthermore, the order of the output will not be the same for every run as the processes execute asynchronously and only synchronise through message passing. A possible solution is to have each process write its output to a disk file. However, this introduces the problem of non flushed file buffers; if the process crashes the buffer might not be flushed, thus missing output that was written by the program. In the worst case this can lead the programmer to believe that the process crashed somewhere between the last print statement that appears in the file and the first one that does not. A lot of time could then be wasted looking for an error in a place where no error can be found.

**Dimension 3: The root cause of the error.**

This last dimension contains nine categories:
1. **Memory**: Memory was clobbered or used up, managed poorly and so on.

2. **Vendor**: Hardware faults and buggy compilers, development tool or runtime systems.

3. **Design logic**: The algorithm is incorrect.

4. **Initialization**: Covers initialization errors, wrong types and so on.

5. **Variables**: Wrong use of operators or variables.

6. **Lexical**: Lexical problem, bad parse or ambiguous syntax.

7. **Unsolved**: Still a mystery.

8. **Language**: Language ambiguities, erroneous semantics and so on.

9. **Behaviour**: Unanticipated behaviour by user.

<table>
<thead>
<tr>
<th>Category</th>
<th>No. of answers</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>13</td>
<td>25.5%</td>
</tr>
<tr>
<td>Vendor</td>
<td>9</td>
<td>17.7%</td>
</tr>
<tr>
<td>Design logic</td>
<td>7</td>
<td>13.7%</td>
</tr>
<tr>
<td>Initialization</td>
<td>6</td>
<td>11.8%</td>
</tr>
<tr>
<td>Variables</td>
<td>4</td>
<td>7.8%</td>
</tr>
<tr>
<td>Lexical</td>
<td>3</td>
<td>5.9%</td>
</tr>
<tr>
<td>Unsolved</td>
<td>3</td>
<td>5.9%</td>
</tr>
<tr>
<td>Language</td>
<td>2</td>
<td>3.9%</td>
</tr>
<tr>
<td>Behaviour</td>
<td>2</td>
<td>3.9%</td>
</tr>
<tr>
<td>No answer</td>
<td>2</td>
<td>3.9%</td>
</tr>
</tbody>
</table>

Figure 1.5: Dimension 3. What was the root cause.

Figure 1.5 shows that nearly 50% of the errors are caused by the first two categories. This also perfectly agrees with previous studies where tools and runtime systems have been described as a source of errors [Pan94].
The classification used in dimension 3 is a mixture of deep plan analysis [Joh83, SSP85] and phenomenological analysis [Knu89]. Deep plan analysis states that many bugs can be accounted for by analysing the high level abstract plans underlying specific programs, and by specifying both the possible fates that a plan component may undergo (i.e. missing or misplaced). An alternative phenomenological taxonomy can be found in [Knu89] where the root causes are divided into 9 categories, all very similar to the ones relayed here.

1.2.1 Top-down versus bottom-up debugging

![Diagram of top-down vs. bottom-up debugging](image)

Figure 1.6: Top-down vs. bottom-up debugging.

Many debugging tools and environments offer a global view of the entire program and leave it to the programmer to narrow the search space, including specializing the formation and testing of hypotheses and searching for errors. This approach poses one of the greatest problems with existing tools and environments. The set of visualisation tools and environments do support a global problem identification and hypothesis making process, however, they do not readily support the process of localising the error and mapping it back to the source code. I believe that this is due to the information overload theory presented earlier. I refer to this method as the top-down approach to debugging.

In Figure 1.6 the inner shape, containing various error types, represents the potential errors in a program. The figure has been divided into a number of parts, one for each type of error that can occur. The outer bold shape represents the typical way of debugging when using debugging tools: the top-down approach.
A different approach is obtained by turning this well known method upside down. Instead of providing a global view of a program and allowing the user to look for any kind of error using just one tool, I propose that a bottom up approach be adopted.

Assuming the user has made some hypothesis about the type of error, typically based on the report obtained when the error manifested. I propose the application of a tool specifically tailored to supporting hypothesis creation, verification and error search of the specific type of error.

1.2.2 The Purpose of This Research

The purpose of this research is to examine the idea of a bottom up approach, which I refer to as multi-level debugging, over the more conventional top-down approach described earlier. The focus will be closely tied to the major points of the multi dimensional analysis described earlier and the description illustrated in Figure 1.1.

To succeed in this task I believe that the following three points must be understood and shown to be manageable tasks:

1. **Error classification.** I wish to determine the various types of errors involved in parallel message passing programming and develop a methodology for efficient debugging of parallel message passing programs. A number of new types of errors will arise when dealing with parallel message passing systems. I still believe that dimensions 1 and 2 can be applied as is, whereas dimension 3 must be extended to contain error types caused by message passing.

2. **Tool development.** I believe that understanding these error types makes it possible to write specific tools that can greatly assist the programmer to more easily debug parallel message passing programs.

3. **Automation.** I believe that some of these tools can be semi-automated to remove part of the burden of debugging from the programmer.

If these three tasks can be accomplished, they will promote the writing of parallel message passing programs by allowing easy-to-use debugging tools that users will find useful.

The subjectiveness of easy-to-use should optimally be verified through user studies, but this lies beyond the scope of this thesis. I will rely on the acceptance of the technique and the tools by
the parallel programming community (through papers and conferences) as a measure of success. In the following sections I will elaborate on the three points mentioned earlier.

### 1.2.3 Error Classification

When studying the task of debugging parallel programs Figure 1.1 illustrates a good starting point. An error in a parallel program can occur at any of the three different levels shown in Figure 1.1. The data decomposition can contain errors as well, but this thesis is not concerned with these types of errors. However, this is a large separate subject that has been described in detail in books such as [Fos95, FJL+ 88].

I will briefly discuss some of the obvious types of errors that can be encountered at the three different levels.

The errors at the sequential level have already been analysed in great detail and described in the previous sections, but as mentioned, many of these errors occur in the parallel domain as well. In particular it is worth noting that the cause/effect chasm mentioned in [Eis97] is further widened, as the possibility for even greater distance between cause and effect arises when message passing is involved. When messages are propagated from one piece of code to another through message passing, an incorrect value can occur and be used in a piece of otherwise correct sequential code.

![Diagram of error propagation through message passing.](image)

**Figure 1.7: Error propagation through message passing.**

Figure 1.7 illustrates this situation. Process A computes a bad value for variable a (assuming a=0) and sends it to process B. Process B uses the value of a as a divisor and hence crashes. It immediately looks as if the error is caused by faulty division, but indeed it is caused by a wrong computation by function f in process A. This error then propagates to process B through message passing. This example is typical of how the cause/effect chasm becomes even more important when message passing is involved, because now errors can be propagated from one process to
another through the network. Furthermore, this issue becomes even more severe when an error propagates through a number of processes before it is detected.

This point carries directly into debugging at the message level. Here are some of the errors that can occur:

- **Wrong values (variables) sent/received.** This could be an example of one of the types of errors Knuth [Knu89] classifies as a type T (Trivial typo) error.

- **Too little/much data sent/received.** This fits nicely into the M category (mismatch between modules); the programmer is unaware of the mismatch between packs and unpacks.

- **Variables packed/unpacked in the wrong order.**

The highest level of debugging involves debugging the communication protocol, that is the structure according to which the processes of the system communicate. Some of the potential errors here are these:

- **Deadlocks.**

- **Messages delivered to the wrong receiver.**

- **Messages attempted received from the wrong sender.**

All these points are potential pitfalls in the parallel programming domain. Debugging for all these errors is necessary. Some of them are fairly easy to deal with, while others are more problematic.

### 1.2.4 Tool development

In the previous section I gave examples of the types of errors that can occur, and divided them into the three categories associated with the breakdown in Section 1.1.1.

Since these errors are conceptually different, that is sequential errors are associated directly with the sequential code, message errors can be manifested by otherwise correct sequential code, though caused by a different process than the one the errors occurred in. Given the difficulty in
having users adopt tools, I also believe that in order to increase the usage of the tool, it must be
designed with the following goals in mind:

- It is vital that the tool can be used directly on the source code without having to convert it to
  a special environment. If the user can simply recompile or relink with a debugging library
  instead of having to translate or transcribe the code to an environment that does not fully
  support the programming task, the likelihood that the user will adopt such a tool is higher. I
  believe that this is partly caused by not only the time it takes to transform the code but also
  by the inherited conservatism of users using tools as described in [Pan93b]. A major goal is
  to make the tools easy to use.

- To promote the appeal of the tool it must be easily executable, either from the command line
  or within a simple interface that does not require the user to learn a new environment.

- Finally, the tool should have the ability to find and correct a specific type of error depending
  on its manifestation.

These points are supported by the design goals proposed by Eisenstadt in [Eis97].
The most important ones are these:

- Computable relations should be computed on request by the tool, not left to the user to
deduce on his own.

- Displayable states should be displayed on request, not left to the user to draw or visualise
  herself.

- Views for “key players” (important pieces of information) other than variables should be
  provided.

- A variety of navigation tools should be provided at different levels of granularity.

Thus, I propose a multi level tool whose modularity (levels) closely follows that of the error
classification mentioned in section 1.1.1 and the above design goals. (See Figure 1.1)
The last of the above points can be expanded into the two following design goals for such a
tool:
1. *Conceptual modularisation.* Depending on which type of error the user is trying to correct, an appropriate tool should be applied. Not only does this mean that a certain part of the tool is tailored specifically to finding and correcting errors of a specific type, it also means a reduction in the amount of useless information reported to the specific debugging task.

2. *Extensibility.* The overall debugging tool should allow for easy extension if new tools are implemented and need to be added.

### 1.2.5 Automation

If a certain task in a debugging session can be automated the tool should do so. For example, when trying to resolve deadlocks it is possible to automate the search for a way to change the program to avoid a deadlock; at a higher level it is possible to automate the verification of the protocol of the system at runtime.

Protocols can be specified and verified using tools like CSP [Ros93, Ros94]. However, in order to use such tool the user must have a strong background in theory, as the specification of a protocol is a complicated task. For CSP models it is the protocol that is checked for deadlocks, livelocks and so on, I propose a verification module where the specification is much easier to write, and where the system simply checks all messages against the protocol, that is, the protocol is not verified but the system is checked against some specification called a protocol.

I believe that automation is a strong concept and that a lot of potential lies in this area. However, many of these problems are NP-complete or undecidable, so there is a need to develop heuristics and other methods to suggest solutions or at least hints to help the programmer solve the problem at hand.

I believe that by focusing on a particular type of error and developing heuristics geared towards this error type, it might be possible to raise the limit of what can be automated. This means that by narrowing the search space, as shown in Figure 1.6, we can increase the size of the set of problems that can be solved (or semi-solved) by automation. This is illustrated in Figure 1.8.
1.2.6 Existing Approaches

In the previous sections I described the parallel programming domain, showed some of the numerous places where errors can occur, and described some of the errors. This discussion shows the importance of good tools for programmers working within this domain.

In this section I will briefly describe some of the approaches that exist, not so much for debugging, but more so for initial program development. For a more detailed explanation on existing debugging methods I refer to the related work section in Chapter 3.

The obvious group of tools to be deployed when developing parallel systems is the family of parallel programming environments. These environments can greatly reduce the number of errors programmers make.

There has been a number of these tools developed over the years, and in Chapter 3 some of them are presented. Though, many of these tools restrict the user to avoid certain types of errors, their main purpose is not to generate error free programs, but to assist the user in more easily writing parallel applications. Even when using tools or programming languages with built-in support for parallel programming, the problem of locating and correcting errors persists. Despite the obvious advantages found in many of these systems, in [pancake] it is argued that not many are widely adopted. In fact, it is claimed that “often only the developers of the tools end up using them in the end.” A number of reasons for this paradox is given:

- *Steep learning curve.* Many of the tools are advanced and offer a wide variety of functionality. They can be quite difficult and time consuming to learn.
• **Difficult abstraction.** The abstraction adopted by a tool, for example the way a program is represented, the way communication is specified, and its limitations, can be difficult to understand and familiarise oneself with.

• **Restrictiveness.** Many tools are so restrictive they often end up working against the programmer. One example is a tool that assures that any code created is deadlock free. However, this apparent advantage has a drawback: programs with dynamic communication can not be expressed using this tool.

• **Conservatism.** There tends to be a general skepticism towards new tools or languages, especially if they require the user to learn a new language, integrated tool or other time consuming task

   Given the difficulty with tool adaptation and the inherent conservatism that perpetuates the use of well known methods and tools, debugging is still unavoidable.

### 1.2.7 Thesis Statement

The aim of this thesis is to show that debugging can be decomposed into the use of several tools, each tailored to a specific error type, thus working on different levels of the program structure (sequential code, message passing, communication protocol and more). I propose a bottom-up technique called “multi level debugging” and develop tools to support debugging according to this methodology. These techniques will focus on the different types of errors described in section 1.1.1, and I will present an implementation (Millipede), a prototype that can be used to evaluate the quality of the techniques.

### 1.3 Hypotheses, Theories, and Research Questions

The main hypothesis of this research is therefore,

> It is possible to decompose the debugging task of a parallel message passing program into the use of a number of tools operating on different levels of the program, each specifically tailored to locating and assisting in correcting a specific type of error.
Another well known analogy:

“it is possible to screw in Phillips screws with a regular screwdriver, but it takes a lot longer and the result might not be as good as if you did it with the right tool.”

I believe that my hypothesis is supported by the extensive work of Cheri M. Pancake [Pan94, Pan93a, ea93, Pan93b, Pan99], Araki et. all [AFC91] and others [MB93, Eis97], especially by the following points:

- The apparent lack of use of debugging tools and other graphical user interfaces for programming.
- The information overloading current tools suffer from.

In [AFC91] the reason given for the lack of use of debugging tools is because they do not provide enough abstraction to represent and retrieve information at the specification and computation-model level. This strongly supports my theory about multi level debugging.

By implementing a prototype of a tool that follows the multi level debugging methodology I will hopefully be able to show that the hypothesis holds true. I also believe that publishing papers on the subject and thereby promoting dialogue among others in the field, will prove useful in putting my hypothesis to the test.
Chapter 2

Millipede

In this chapter I will introduce a prototype of a multi level debugger, Millipede, and explain the proposed debugging techniques for different levels, plus introduce modules in the Millipede tool to deal with these levels.

As mentioned earlier, I propose a three level model to represent a parallel program:

- The first level, a sequential (core) layer, that consists of the straight line code of the parallel program. The vast majority of code lines are found here as this deals with the functionality of the program. This layer can either be adapted from a sequential version of the original program (if the algorithm parallelizes in a manner similar to the sequential code) or it can be an implementation of a (different) algorithm solving the original problem in parallel.

- The second level, is where message passing occurs. This level is concerned with sending and receiving messages. The communication calls found in the parallel code are associated with this level. Here the first problem with parallel debugging arises: inspecting and changing messages and their content is virtually impossible using state of the art sequential debuggers. Nevertheless, it is an important part of debugging parallel message passing programs, because often an error is caused by data (or lack there of) passed in message passing calls. One of the challenges of debugging messages using a sequential debugger is having to handle more than one process (in most cases two processes).

- The third level, the protocol level, is no longer concerned with only one or two processes,
but with the entire program and all its processes. Since messages are sent between processes according to a specific scheme, it is important and useful to be able to control and debug at this high level of abstraction. This is not directly possible with a sequential debugger because it does not have all the information needed to support such a debugging task.

This briefly summarizes the three major levels. As will become apparent later, each level can further be divided into smaller parts depending on the specific debugging task needed. For example, the third level (the protocol level) has two major components: the protocol checking module (MOPED) and a deadlock correcting module (DCM).

### 2.1 Design

Some of the more crucial design goals taken into account include the following:

1. Access to source code debugging is still vital as most errors are located in the straight line sequential code. Thus, it should be possible to apply whichever favourite sequential debugging tool the programmer might have to the sequential part of the parallel system.

2. Access to messages sent, as well as messages still in the message queues is needed. Not only do many of the modules make extended use of this information, but it might also be useful for new modules that perform other forms of analyses on the messages sent and still left in the queues.

3. Extensibility of the message passing calls should be developed, such as allowing the programmers to add new functionality to existing message passing calls. This could become necessary when other modules are developed by other users of Millipede.

**Millipede** consists of the following main parts:

- **Wrapper functions for all the communication calls.** These new functions are added to the original communication library, overwriting the standard calls. They execute the Millipede debugging code and then call the original functions. Figure 2.1 shows a few examples of these new functions.
A runtime system, consisting of several separately executing processes, which allows the programmer to interact with the debugging system.

2.2 Implementation

As mentioned earlier, Millipede consists of two main components: the wrapper functions and runtime system. In the following subsections I will briefly describe the implementation of these two parts.

2.2.1 Wrapper functions

By redefining the PVM functions, as seen in Figure 2.1, the C compiler will substitute all PVM calls in the user code with calls to the equivalent underscore functions (e.g. _PVM_pkint instead of pvm_pkint). These functions then perform the Millipede debugging code (informing the runtime system about changes, prompting for input, printing output and writing log files, etc.) The implementation of these functions is linked into the original PVM library (libpvm3.a). When a PVM program is compiled, the redefinition is only included if the MILLIPEDe flag is set during compiling. This means that if this flag is not set the program will execute like a normal PVM program, but if the flag is set, Millipede debugging functions will be executed when the program runs. This way of switching between normal and debugger execution is extremely easy to manage, and does not require any rewriting of the program, just a recompilation or relinking.

Figure 2.1: Redefined PVM functions.
2.2.2 The runtime system

The core of the runtime system consists of following three processes:

1. A centralized package number administrator process takes care of assigning unique package numbers to all messages sent.

2. The system reports the status of the processes, information about events and any information requested by the programmer in a status window.

3. A user interface process that queries from the programmer.

A driver process is used to start the program being debugged. Furthermore, it is the main interaction window with Millipede. The driver program starts the above subprocesses and maintains information about the number of running processes. It also provides a front end to the run time system; that is it lets the programmer perform queries about the system, such as message queues and errors. All queries are forwarded to the status window, which is the core component in Millipede.

Figure 2.2: Implementation of Millipede.

2.2.3 Current Status of Millipede

The backbone of the Millipede system has been successfully implemented, i.e. package numbering, message history and the interactive front end (a crude text based version so far) A number of queries can be performed:
- **status** shows the status of all processes in the system.

- **match** given a package number reports the sender and receiver.

- **dump** dumps the entire message history.

- **deadlock** determines if a deadlock has occurred and tries to fix it.

- **locate** given a sender and receiver line number determines if any packages have been sent between the send and the receive.

- **show** shows the contents of the message queues of the entire system.

The sequential debugging module is partially implemented with log-files read/write facilities. It has successfully been used to debug a single process of a parallel system using the **gdb** sequential debugger and the sequential tool **purify**, a tool for locating and correcting memory leaks. The sequential debugging module of **Millipede** can be further explored in [PW00], which is the first paper written about **Millipede**. This paper was presented at The International Conference on Communications in Computing (CIC'2000) in Las Vegas, Nevada, June 26-29, 2000, and published in the proceedings from this conference. The following section goes into more detail about this module.

### 2.3 **Millipede Sequential Debugging Module**

At the lowest level in **Millipede** lies 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. The Sequential Debugging Module works as follows. It first requires that the user compiles and executes 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 in which 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
necessary, however, 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 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 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:

\[
\text{gcc -g -DMILLIPEDE -o Pgm Pgm.c -lpvm3}
\]

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:

\[
\text{setenv MILLIPEDE_RCM}
\]

This instructs Millipede to create log files labeled Millipede_RPF-xxx-yyy 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 parallel execution has terminated Millipede can be switched to Replay Execution Mode (REM) and the message passing system can be closed down:

\[
\text{unsetenv MILLIPEDE_RCM; setenv MILLIPEDE_RCM}
\]

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:

\[
\text{gdb Pgm}
\]

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

\[
\text{Replay filename: MILLIPEDE_RPF-Pgm-262152}
\]

Sequential debugging now commences as if the program were a 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.

2.3.1 Examples of Uses of the Sequential Debugging Module

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 that 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. Figures 2.3 and 2.4 show the relevant parts of the slave program containing the errors.

<table>
<thead>
<tr>
<th>Parallel code:</th>
<th>Tool: GDB</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pvm_upkint(&amp;nproc, 1, 1);</code></td>
<td>(gdb) step</td>
</tr>
<tr>
<td><code>pvm_upkint(tids, nproc, 1);</code></td>
<td><code>45 e = n % nproc;</code></td>
</tr>
<tr>
<td><code>pvm_upkint(&amp;n, 1, 1);</code></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>Program received signal,</td>
</tr>
<tr>
<td><code>e = n%nproc;</code></td>
<td>SPGPFE Arithmetic</td>
</tr>
<tr>
<td>.</td>
<td>exception 0xef4a86a8 in ()</td>
</tr>
</tbody>
</table>

Figure 2.3: Division by zero.

Consider the code shown on the left side of Figure 2.3. 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 by Millipede using gdb (the right side of Figure 2.3), the error is easily located. In contrast, to find this error using $N$ versions of gdb online the programmer would

28
have to single step each process long enough to make sure the communication had taken place
that is, that the process in the focus of attention could execute the receive call that would lead to
receiving the value of the faulty nproc variable.

| Tool: Purify |
| ABW: Array bounds write |
| This is occurring while in: |
| main [Wave_slave.c:57] |
| for (i=1;i<=nodes;i++) |
| ⇒ x[i] = (l*(start+i-1))/(n-1); |

Parallel code:

| 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 [rtlbin.o] |
| alloc [rtlbin.o] |
| main [Wave_slave.c:50] |
| ⇒ x = calloc(nodes,sizeof(double)); |
| y = calloc(nodes,sizeof(double)); |

Figure 2.4: Memory leak that leads to a segmentation fault.

The left part of Figure 2.4 shows an example of a code fragment that indexes an array out of
bounds. The x array is not large enough and is indexed from 0 to nodes + 1, but has only been allo-
cated to hold nodes number of elements. This error can result in two different program behaviours:
one, an incorrect result; two, 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 pro-
grams 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 $N$ versions of a sequential debugger; one for each process. 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. Other well known errors, such as bad pointer references and other types of memory leaks, can easily be detected in a similar manner.

As seen earlier the cause and effect of the same error can be quite far off from each other, in terms of both program lines and execution time. In [Eis97] 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. To correct such errors the first step is to locate the error locally, that is, figure out which variable holds the faulty value. Once the variable has been determined, the second step is to determine whether the variable got its faulty value with in the local process, or if it received the value through a message call. If it is determined that the faulty value arrived from another process through message passing, the programmer can sequentially debug this process to see where the value was computed. 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.

## 2.3.2 Limitations

*Millipede* is tailored to allow running of 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
Receiving New Package:

```
pvm_recv(-1,0) < ok>
pvm_upkint(&nproc,1,1) = [2] < ok>
Do you want to change this [y/n] ? y
int: nproc = 2. New value = 3
pvm_upkint(tids,3,1) = [262151,262152,35] < ok>
Do you want to change this [y/n] ? n
```

Figure 2.5: Inspecting and changing message content.

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.

### 2.4 Millipede Message Debugging Module

The message module is still under development. However, it is possible to choose specific processes for debugging, for example, inspecting and changing message content. Figure 2.5 illustrates this; the boldface numbers and letters are user input.

As briefly mentioned earlier a number of queries can be performed.

Figure 2.6 shows an example of executing the `status` command. Millipede will match each send to a receive, show the file names and the line numbers of the message passing calls. For a `pvm_send` the first argument is the ID of the receiver, and the second argument is the message tag. For a `pvm_recv` the first argument is the Id of the sender, and the second argument is the message tag. Both sender and message tag in a receive call can be specified as -1, a wild card value, which matches any sender or message tag.

By executing the `match` command and typing in a package number, Millipede will query the
<table>
<thead>
<tr>
<th>#</th>
<th>Command</th>
<th>Line</th>
<th>File</th>
<th>Command</th>
<th>Line</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td><code>pvm_send(262150,5)</code></td>
<td>118</td>
<td><code>slave.c</code> ↔ <code>pvm_recv(262152,5)</code></td>
<td>86</td>
<td><code>master.c</code></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td><code>pvm_send(262150,5)</code></td>
<td>118</td>
<td><code>slave.c</code> ↔ <code>pvm_recv(262151,5)</code></td>
<td>86</td>
<td><code>master.c</code></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td><code>pvm_send(262152,0)</code></td>
<td>78</td>
<td><code>master.c</code> ↔ <code>pvm_recv(262150,0)</code></td>
<td>22</td>
<td><code>slave.c</code></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td><code>pvm_send(262151,0)</code></td>
<td>78</td>
<td><code>master.c</code> ↔ <code>pvm_recv(262150,0)</code></td>
<td>22</td>
<td><code>slave.c</code></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><code>pvm_send(262152,11)</code></td>
<td>75</td>
<td><code>slave.c</code> ↔ <code>pvm_recv(262151,11)</code></td>
<td>89</td>
<td><code>slave.c</code></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td><code>pvm_send(262151,22)</code></td>
<td>80</td>
<td><code>slave.c</code> ↔ <code>pvm_recv(262152,22)</code></td>
<td>85</td>
<td><code>slave.c</code></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><code>pvm_send(262152,11)</code></td>
<td>75</td>
<td><code>slave.c</code> ↔ <code>pvm_recv(262151,11)</code></td>
<td>89</td>
<td><code>slave.c</code></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><code>pvm_send(262151,22)</code></td>
<td>80</td>
<td><code>slave.c</code> ↔ <code>pvm_recv(262152,22)</code></td>
<td>85</td>
<td><code>slave.c</code></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.6: Executing the `status` command.

message queues for all packing and unpacking commands that involves this package number. Figure 2.7 shows an example of using the `match` command to look for package number 6. The upper half of the figure shows the packing routines and their line numbers, and the lower half shows the unpacking routines and their line numbers.

If the user wishes to see which packages were sent to which processes, the `dump` command can be applied. This command will show all the packages sent by each process in reverse order.

The last of the queries currently available is the command `locate`. This command and query the system to locate packages sent between a set of line numbers. Figure 2.9 shows the result of executing the `locate` query with line numbers 75 and 89.

Two of the design goals are that displayable states should be displayed on request and computable relations should be computed on request. Instead of providing a number of fixed views like the ones described, I believe that providing an SQL like interface to message queues, sent messages, and data would greatly increase the usability of this module.
Figure 2.7: Executing the match command.

<table>
<thead>
<tr>
<th>Queue: 0</th>
<th>Filename: master.c</th>
<th>Tid: 26228</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package no.</td>
<td>Sender tid</td>
<td>Receiver tag</td>
</tr>
<tr>
<td>6</td>
<td>262229</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>262230</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Queue: 1</th>
<th>Filename: slave.c</th>
<th>Tid: 26230</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package no.</td>
<td>Sender tid</td>
<td>Receiver tag</td>
</tr>
<tr>
<td>4</td>
<td>262229</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>262229</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>262228</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Queue: 2</th>
<th>Filename: slave.c</th>
<th>Tid: 262229</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package no.</td>
<td>Sender tid</td>
<td>Receiver tag</td>
</tr>
<tr>
<td>5</td>
<td>262230</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>262230</td>
<td>22</td>
</tr>
<tr>
<td>1</td>
<td>262228</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 2.8: Executing the dump command.
Looking for packages sent from line 118 to line 86.

<table>
<thead>
<tr>
<th>Package number</th>
<th>Sender Tid</th>
<th>Sender File</th>
<th>Sender Line</th>
<th>Receiver Tid</th>
<th>Receiver File</th>
<th>Receiver Line</th>
<th>Size</th>
<th>Sender tag</th>
<th>Receiver tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>262229</td>
<td>slave.c</td>
<td>86</td>
<td>262228</td>
<td>master.c</td>
<td>118</td>
<td>152</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>262230</td>
<td>slave.c</td>
<td>86</td>
<td>262228</td>
<td>master.c</td>
<td>118</td>
<td>144</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 2.9: Executing the locate command.

2.5 **Millipede Protocol Debugging Modules**

The third level, the protocol level, currently consists of two tools:

1. *The Deadlock Correction Module (DCM)*, which can be used to correct deadlocks in a halted system.

2. *The Protocol Verification Module (MOPED)*, which can be used for verifying a communication protocol at runtime.

The Deadlock Correction Module is a tool that extracts sent and left over messages from a deadlocked system and suggests corrective measures to change the program in order remove the deadlocks. The theory is further described in [PW01], which has been accepted at the HIPS workshop at The International Parallel and Distributed Processing Symposium (IPDPS) to be held in San Francisco, April 23-27, 2001.

This module is operational. In [PW01] we have given a polynomial time algorithm, but this algorithm only works for systems without message tags, that is only a receiver/sender is specified in a send/receive call. The algorithm currently implemented in Millipede uses a brute force search through all possible combinations of sender/receiver pairs and chooses the permutation that requires the least number of program changes to remove the deadlock from the program. The major disadvantage of this approach is the time complexity of this algorithm which is \(O(n!)\) However, we do believe that this algorithm, based on flow networks, can be extended to also cover systems that make use of message tags. I believe that further research into this area will show, that with tags a similar algorithm can be applied. The following section goes into more detail about this module.
2.5.1 Millipede Deadlock Correction Module

The basic send and receive calls in PVM and MPI are as follows:

\[ \text{send}(\text{buffer, receiver, nodeID, tag}) \]

\[ \text{recv}(\text{buffer, sender, nodeID, tag}) \]

Mistyping the nodeID or tag value results in a message that is either undelivered or received by the wrong process.

For example, consider the simple case of a single error as shown in Figure 2.10.

\[
\begin{array}{c|c}
\text{Process A} & \text{Process B} \\
\hline
\text{Send(buf, B, t1)} & \text{Recv(buf, A, t1)} \\
\text{Recv(buf, A, t2)} & \text{Send(buf, C, t2)} \\
\end{array}
\]

Figure 2.10: Simple error.

There is an error in the send call of process B in Figure 2.10; B attempts to send a message to A but incorrectly sends it to C. Depending on whether the communication is synchronous or asynchronous, process B either blocks, eventually hanging the system, or terminates. In either case it results in an undelivered message in the system. Millipede records a message history and makes it possible to extract both undelivered messages and recently delivered messages from the system. The same kind of error can occur if a tag value is incorrect.

The Algorithm

**Definition 1:** Let \( S = (s_0, s_1, \ldots, s_{n-1}) \) be an ordered list of senders where each \( s_i = (a, b) \) and \( a, b \) integer process identifiers (ranks in MPI). Let \( R = (r_0, r_1, \ldots, r_{n-1}) \) be an ordered list of receivers where each \( r_i = (a, b) \) and again \( a, b \) are process identifiers. For \( s_i = (a, b) \in S, a \) is fixed by the ID of the sending process, and for \( r_i = (a, b) \in R, b \) is fixed by the receiver.
Definition 2: A match between a sender $s_i = (a_i, b_i)$ and a receiver $r_j = (a_j, b_j)$ occurs when $(a_i = a_j) \land (b_i = b_j)$. The opposite is called a mismatch.

For the sake of simplicity we do not consider message tags or wild cards in this first analysis. We will, however, return to these cases later. The intuition behind the algorithm is as follows:

Find a permutation of $S$ denoted $\pi_s$ and a permutation of $R$, denoted $\pi_r$, where a minimal number of fields need to be changed in order to obtain a system without any unmatched sends/receives. If $\pi_s$ and $\pi_r$ are such permutations it follows that after changing the required fields that for all $s_i \in \pi_s$ and $r_j \in \pi_r$, where $(i = j)$ that $a_i = a_j$ and $b_i = b_j$. This means that if the user changes her program accordingly the deadlock will disappear. The desired permutations can be obtained by computing a hamming distance between all possible combinations of permutations of senders and receivers, and choosing the one or ones that give rise to the smallest hamming distance.

Unfortunately this algorithm has time complexity $O(n!)$. However, it is possible to reduce the problem to a bipartite matching problem [Pre92]. The approach is as follows:

Let $\tilde{G} = (\tilde{V}, \tilde{E})$ be a graph where

- $\tilde{V} = \tilde{V}_s \cup \tilde{V}_r$ where $\tilde{V}_s$ and $\tilde{V}_r$ both are sets representing all the process ids in the system. ($\tilde{V}_s$ represents sending processes and $\tilde{V}_r$ represents receiving processes).

- $\tilde{E}$ is constructed in the following way.
  - For all messages $m$ not delivered (left in message queues) do the following:
    * If $m = (s, r)$ is an outstanding send add edge $(s, r)$ where $s \in \tilde{V}_s$ and $r \in \tilde{V}_r$ to $\tilde{E}$ with capacity 2.
    * If $m = (r, s)$ is an outstanding receive add edge $(r, s)$ where $s \in \tilde{V}_s$ and $r \in \tilde{V}_r$ to $\tilde{E}$ with capacity 2.
  - Iterate backwards through all successfully delivered messages $(u, v)$ and add edges $(u, v)$ and $(v, u)$ with capacity 2 to $\tilde{E}$ if no other edge exist in $\tilde{E}$ that has $u$ or $v$ as either
source or destination.

- Add edges with capacity 1 to $\tilde{E}$ to make $\tilde{G}$ a complete graph bipartite graph.

Now run the maximum bipartite graph matching algorithm, which uses flow-graphs to obtain a matching in $\tilde{G}$ [CLR90]. This matching results in a system without deadlocks. Furthermore this matching can be obtained by changing a minimum number of fields in the senders and receivers. The time complexity is $O(|\tilde{E}| \cdot |f^*|)$ where $|f^*|$ is the size of the matching. Since $\tilde{G}$ is a complete graph $|\tilde{E}| = n^2$ and $|f^*| = n$. Therefore the time complexity is $O(n^3)$. We do not yet have a polynomial time algorithm for the case where tags are considered, but as previously stated, we believe that the bipartite graph matching problem can be adapted to cover this case as well.

**Algorithm accuracy**

In this section we will evaluate the quality of the algorithm, that is we will verify that the algorithm does not frequently return an incorrect answer or more than one answer (there could be more than one way to fix a deadlock with a minimum number of field changes). To do this we need to introduce a model that precisely describes a system of senders and receivers equivalent to the one used in the previous section. In the following, $n$ denotes the number of senders and receivers, and $k$ the number of errors in the system. We start out by defining a few concepts.

A **communication configuration** $C$ is a pair $(S, R)$ ($S$ and $R$ defined in definition 1). $SR(n)$ is a set of all communication configurations with $n$ senders and $n$ receivers.

A send $s_i = (a_i, b_i)$ is called **unmatched** if for $r_{b_i} = (a_j, b_j), a_j \neq a_i$. Equivalently a receive $r_j = (a_j, b_j)$ is called **unmatched** if for $s_{a_j} = (a_i, b_i), b_i \neq b_j$. We call a communication configuration **valid** if it has no unmatched sends or receives.

Given a configuration $(S, R) = \{\{s_0, \ldots, s_{n-1}\}, \{r_0, \ldots, r_{n-1}\}\}$ in $SR(n)$, $s_i = (a_i, b_i)$ and $r_j = (a_j, b_j)$. The associated **directed bipartite graph** $G = (V, E)$ is defined by

$$V = \left( \bigcup_{s_i \in S} a_i \cup \bigcup_{r_j \in R} a_j \right)$$

$$E = \left( \bigcup_{s_i \in S} (a_i, b_i) \cup \bigcup_{r_j \in R} (b_j, a_j) \right)$$
Figure 2.11: All valid communication configurations in SR(2).

Figure 2.12: $\bar{B}(v_1, 1) \subset SR(2)$.

Figure 2.11 shows the only two valid configurations in a system with two senders and two receivers (SR(2)).

A valid communication configuration $v \in SR(n)$ where $s_i = r_i \ \forall \ i, \ 0 \leq i < n$ is called the correct configuration (there is only one correct configuration in $SR(n)$). In Figure 2.11 the communication configuration $v_1$ is the correct communication configuration of SR(2).

From now on the correct configuration will be denoted $v_c$.

**Definition 3:** $B(v, i)$ is the set of communication configurations that can be created by moving $i$ or less arcs from $v$, $B(v, i) = B(v, i) \setminus B(v, i - 1)$ and $B(v, 0) = B(v, 0) = \{v\}$
Figure 2.13 shows $\mathcal{B}(v_1, 1)$ for $v_1$ from Figure 2.11. Let any invalid communication configuration $v$ be given. In Figure 2.13 the boldface $\times$ marks $v$. The boxes mark valid configurations and the rest, marked by $\times$, are other invalid configurations. The solid line marks $\mathcal{B}(v, 0)$, the dashed line $\mathcal{B}(v, 1)$ and the dotted line $\mathcal{B}(v, 2)$. We wish to correct the invalid communication configuration to be a valid communication configuration by moving as few arrows as possible. This is equivalent to choosing the correct configuration(s) with the smallest hamming distance to $v$ and correcting the fields that do not match. This is done by choosing the first (one or more) valid communication configuration(s) included in the series $\mathcal{B}(v, 1), \mathcal{B}(v, 2), \ldots$. In the example in Figure 2.13 a valid configuration is found in $\mathcal{B}(v_1, 1)$.

We will show that for any invalid communication configuration in $\text{SR}(n)$ the probability that the first encountered valid communication configuration is the correct communication configuration is high. In other words if we introduce $k$ errors into a valid communication configuration $C$, then the algorithm will in most cases end up proposing $C$ as the correct way to fix the erroneous system. Only in rare cases will it suggest any of the other valid communication configurations in the corresponding system.

Given $\text{SR}(n)$ and a list of the valid configurations $\mathcal{V} = \{v_0, \ldots, v_{n!-1}\}$. Let $k \leq \frac{n}{2}$ be the number of errors in the system. We need to consider the configurations obtainable by introducing one error to all $v_i$. This is a set of sets:

$$\mathcal{B}_1 = \{\mathcal{B}(v_0, 1), \mathcal{B}(v_1, 1), \ldots, \mathcal{B}(v_{n!-1}, 1)\}$$
Figure 2.14: Example of overlapping \( B \) sets: \( \bigcap_i B(v_i, 1) = \emptyset \) but \( \bigcap_i B(v_i, 2) \neq \emptyset \)

If we know for every system with one error that
\[
\bigcap_{i=0}^{n!-1} B(v_i, 1) = \bigcap_{b \in B_1} b = \emptyset
\]
then the algorithm will always suggest a correct solution. Figure 2.14 illustrates this example.

The goal is therefore to show the following:
\[
\frac{|B_i \cap B_j|}{|B_i|} \quad \text{and} \quad \frac{|B_i \cap B_j|}{|B_j|} \quad \text{is small} \quad \forall \; e \leq k, \forall \; B_i, B_j \in \mathcal{B}_e, \; (i \neq j).
\]

In other words small means an acceptably low fraction of wrongly proposed fixes. We will see that this is equivalent to showing that
\[
\frac{|B_0 \cap B_i|}{|B_0|} \quad \text{is small} \quad \forall \; e \leq k, \forall \; B_i \in \{ B(v_1, e), \ldots, B(v_{n!-1}, e) \}
\]
where \( B_0 = B(v_c, e) \) where \( v_0 = v_c \) is the correct communication configuration of \( \text{SR}(n) \).

To more easily handle these communication configurations, we are going to introduce a short hand notation. For each communication configuration in \( \text{SR}(n) \) (the size of \( \text{SR}(n) \) is \( n^{2n} \)), we assign a \( 2n \) digit number \( s_1r_1 \ldots s_nr_n \) \( (s_i, r_i \in \{0, \ldots, n-1\}) \) in the following way:
$s_i$ equals the number of the receiver that sender number $i$ is sending to, and $r_i$ equals the number of the sender that receiver number $i$ is trying to receive from.

For example, using the 2 configurations in Figure 2.11 we get the following short hand representation: $v_1 = 0011$ and $v_2 = 1100$. Figure 2.15 shows which configurations can be reached in $k$ steps from the correct configuration.

![Figure 2.15: Configurations that can be reached in $k$ steps from the configuration 0011.](image)

We now proceed to proving a number of lemmas that will help prove (1).
**Lemma 1:** The number of valid configurations in SR(n) is n!.

**Proof:** For a configuration to be valid each sender must send to a distinct receiver, and this receiver must receive from this sender. If $s_i = j$ then $r_j = i$. It is therefore sufficient to determine the number of different ways to order n senders. There are n! such ways. \qed

**Lemma 2:** The size of $B(v, i)$ denoted $|\mathcal{B}(v, i)|$ is

$$\sum_{j=0}^{i} \binom{2n}{j} (n-1)^j$$

**Proof:**

$$|\mathcal{B}(v, i)| = |\bigcup_{j=0}^{i} \mathcal{B}(v, j)| = \sum_{j=0}^{i} |\mathcal{B}(v, j)| = \sum_{j=0}^{i} \binom{2n}{j} (n-1)^j$$ \qed

**Definition 4:** The distance between two valid configurations in SR(n), denoted $d(v_i, v_j)$, is defined as:

$$d(v_i, v_j) = \sum_{i=0}^{2n} [v_i \neq v_j]$$

where $v_i = v_{i_1}v_{i_2}...v_{i_{2n}}$, $v_j = v_{j_1}v_{j_2}...v_{j_{2n}}$ and

$$[v_i \neq v_j] = \begin{cases} 
0 & : \quad v_i = v_j \\
1 & : \quad v_i \neq v_j 
\end{cases}$$ \qed

The valid configurations in SR(3) are 001122, 110022, 002211, 220011, 221100 and 210210. Figure 2.16 shows the distances between the different valid configurations.

**Lemma 3:** For any SR(n), a distance $k$, and a valid configuration $v$ the number of valid configurations in SR(n) with distance $k$ does not depend on the choice of $v$.

**Proof:** Permutations are automorphisms. \qed
**Lemma 4:** The possible distances between valid configurations in $\text{SR}(n)$ are $4, 6, \ldots, 2n - 2, 2n$.

**Proof:** A necessary condition for a configuration to be valid is that $\{s_1, \ldots, s_n\} = \{r_1, \ldots, r_n\} = \{0, \ldots, n - 1\}$. Since all valid configurations are equivalent, consider $v_c = s_1 r_2 s_2 r_2, \ldots, s_n r_n$. A minimum of two send/receive pairs must be switched to obtain a difference valid configuration. This gives a minimum distance of 4. Now choose two send/receive pairs $a, b$ to switch. There are three cases to consider:

1. Both pairs are of the form $s_i r_i = ii$, which means that either they have not been switched before or that they have been switched back to their original state. When these pairs are switched the distance increases by 4.

2. One of the pairs, say $a$, is of the form $s_i r_i = ii$, and the other one, $b$, is not. When $a$ and $b$ are switched, $a$ will contribute with distance 2 to the total distance and $b$ already contributed with distance 2, so the total distance increases by 2.

3. Neither $a$ nor $b$ are of the form $s_i r_i = ii$. Neither will contribute further to the total distance by being switched.

Let $\mathcal{D}(v, k)$ be the set of valid configurations with exactly distance $k$ from the valid configuration $v$. 

---

**Figure 2.16:** Distances between valid configurations in $\text{SR}(n)$. 

<table>
<thead>
<tr>
<th></th>
<th>001122</th>
<th>110022</th>
<th>002211</th>
<th>122001</th>
<th>221100</th>
<th>210210</th>
</tr>
</thead>
<tbody>
<tr>
<td>001122</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>110022</td>
<td>4</td>
<td>0</td>
<td>6</td>
<td>4</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>002211</td>
<td>4</td>
<td>6</td>
<td>0</td>
<td>4</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>122001</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>221100</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>210210</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>
Lemma 5: The size of $D(v, m)$ for $m = 2k$ is

$$|D(v, 2k)| = \binom{n}{k} c_k$$

where

$$c_0 = 1 \quad , \quad c_1 = 0 \quad , \quad c_k = k! - \sum_{i=1}^{k} \binom{k}{i} c_{k-i}$$

Proof: Let $v_c$ be given. The number of valid configurations at distance 0 from $v_c$ is 1 (only $v_c$ itself is distance 0 away!); therefore, $c_0 = 1$. According to Lemma 3 no valid configuration is distance 2 away, so $c_1 = 0$. For distance 4 choose two of the $n$ send/receives to move; this can be done in $\binom{n}{2}$ ways. The number of ways these two send/receives can be permuted is $2!$, but we must subtract the permutation that does not change anything (and results in $v_c$), which is 1, so we get $\binom{n}{2}(2! - 1) = \binom{n}{2}(2! - \binom{2}{2}c_1 - \binom{2}{0}c_0)$. Set $c_2$ equal to this expression, which is the number of configurations with exactly two permuted send/receives giving distance 4. In general we must subtract the number of configurations that are at the wrong distance. For each distance $m = 2k$ there are $\binom{n}{k}k!$ permutations, and again we must subtract all the permutations that do not exactly permute $k$ send/receives, that is do not give the correct distance of $2k$, so we get this:

$$|D(v, m)| = \binom{n}{k} (k! - \binom{k}{1} c_{k-1} - \binom{k}{2} c_{k-2} \cdots - \binom{k}{k-1} c_1 - \binom{k}{0} c_0) = \binom{n}{k} c_k$$

where

$$c_k = k! - \sum_{i=1}^{k} \binom{k}{i} c_{k-i}$$

Consider the following two configurations: $v_1 = 001122$ and $v_2 = 002211$. These two configurations differ in the last four positions, thus have a distance of 4. To compute the intersection $B(v_1, 2) \cap B(v_2, 2)$ we must find the configurations reachable from both $v_1$ and $v_2$ by changing at most two positions in each. Since the distance between the two configurations is four and we may
change at most two positions in each configuration, it follows that we must change exactly two in each. Choose two fields in $v_1$, say $v_{1_i}$ and $v_{1_j}$. Change these two positions to have the values of $v_{2_i}$ and $v_{2_j}$ and obtain $v'_1$. We know that $d(v'_1, v_2) = 2$. Now change the two positions in $v_2$ that differ from $v'_1$, say $v_{2_i}$ and $v_{2_m}$ to have the values of $v_{1_j} = v'_{1_i}$ and $v_{1_m} = v'_{1_m}$ and obtain $v'_2$. We now know that $d(v'_1, v'_2) = 0$. The original distance was four and we must change two fields in each configuration. The number of different ways this can be done is $\binom{4}{2} = 6$.

These are the six configurations that can be obtained:

$$
\begin{align*}
113333, & \quad 112222, & \quad 112323, & \quad 113232, & \quad 113322, & \quad 112332.
\end{align*}
$$

The underlined positions are the fields changed in $v_1$ and the overlined fields are the ones changed in $v_2$.

According to Lemma 5 all valid configurations are equivalent. Therefore we can simply study the properties of the correct valid configuration $v_c$ of SR$(n)$. We will now determine the number of elements in the intersections of the B sets.

**Theorem 1:** Let $e$ be the number of errors in a communication system. The number of configurations with $e$ errors for which the algorithm will either suggest a wrong valid configuration or a set of valid configurations where the correct one is included is

$$
\left| \bigcup_{v \in \mathcal{V}} \mathcal{B}(v, e) \cap \mathcal{B}(v_c, e) \right| \leq \sum_{i=2}^{e} \binom{n}{i} c_i \sum_{b=0}^{e} \sum_{a=\max\{b,e-b\}}^{\min\{a+b+2i,2i-a\}} \sum_{c_o=0}^{\infty} \sum_{i=0}^{e} O(i, a, b, c_o)
$$

where

$$
O(i, a, b, c_o) = [c_{xy} \geq 0 \land c_x \geq 0 \land c_y \geq 0] \left( \frac{2i}{c_x} \right) \left( \frac{2i-c_x}{c_y} \right) (n-2)^{c_{xy}} \left( \frac{2(n-i)}{c_o} \right) (n-1)^{c_o}
$$

and

$$
\begin{align*}
c_{xy} & = (a-c_o) + (b-c_o) - 2i \\
c_x & = 2i - a + c_o \\
c_y & = 2i - b + c_o
\end{align*}
$$
Remember that \( \mathcal{B}(v_c, e) = \bigcup_{a=0}^{e} \mathcal{B}(v_c, a) \) and \( \mathcal{B}(v_j, e) = \bigcup_{b=0}^{e} \mathcal{B}(v_j, b) \).

For a configuration \( v \in \mathcal{B}(v_c, a) \cap \mathcal{B}(v_j, b) \) with \( b = d(v_j, v) \) and \( a = d(v_c, v) \) if \( a < b \) then \( v \) will be reported as the correct communication configuration. Since we are concerned with counting the valid communication configurations that are incorrect fixes, we only consider cases where \( a \geq b \). Additionally, if \( a + b < e \) then the intersection between \( \mathcal{B}(v_c, a) \) and \( \mathcal{B}(v_j, b) \) is empty. These 3 observations combined yield

\[
\left| \bigcup_{v \in \mathcal{V}} \mathcal{B}(v_c, e) \cap \mathcal{B}(v_j, e) \right| = \left| \bigcup_{i \in \{4, \ldots, 2e\}} \mathcal{B}(v_i, e) \bigcap \mathcal{B}(v_j, e) \right| = \sum_{i=2}^{e} \sum_{v_j \in D(v_c, 2i)} \left| \bigcup_{b=0}^{\varepsilon} \mathcal{B}(v_c, a) \cap \mathcal{B}(v_j, b) \right| (2)
\]

We now calculate the value of

\[
\left| \mathcal{B}(v_c, a) \cap \mathcal{B}(v_j, b) \right|
\]

for values \( a \) and \( b \) where \( j = 2i \). Let \( x = v_c = x_1x_2 \ldots x_{2n} \) and \( y = v_j = y_1y_2 \ldots y_{2n} \) with \( d(x, y) = j = 2i \). We want to find configurations \( z = v = z_1z_2 \ldots z_{2n} \) such that \( d(x, z) = a \) and \( d(y, z) = b \). WLOG assume that \( x_k = y_k \) for \( j + 1 \leq k \leq 2n \).

\[
\begin{array}{cccccc}
  &   &   &   &   &   \\
  j & x_1 & x_2 & \ldots & x_j & x_{j+1} \ldots x_{2n} \\
  & z_1 & z_2 & \ldots & z_j & z_{j+1} \ldots z_{2n} \\
  & y_1 & y_2 & \ldots & y_j & y_{j+1} \ldots y_{2n} \\
\end{array}
\]

Now define

\[
\begin{align*}
C_y &= \{ z_k : 1 \leq k \leq j : z_k = x_k \land z_k \neq y_k \} \\
C_x &= \{ z_k : 1 \leq k \leq j : z_k \neq x_k \land z_k = y_k \} \\
C_{xy} &= \{ z_k : 1 \leq k \leq j : z_k \neq x_k \land z_k \neq y_k \} \\
C_o &= \{ z_k : j+1 \leq k \leq 2n : z_k \neq x_k \land z_k \neq y_k \} \\
c_x &= |C_x|
\end{align*}
\]

46
\[ c_y = |C_y| \]
\[ c_{xy} = |C_{xy}| \]
\[ c_o = |C_o| \]

A z-configuration must satisfy

\[ \begin{pmatrix} a \\ b \end{pmatrix} = c_{xy} \begin{pmatrix} 1 \\ 1 \end{pmatrix} + c_x \begin{pmatrix} 1 \\ 0 \end{pmatrix} + c_y \begin{pmatrix} 0 \\ 1 \end{pmatrix} + c_o \begin{pmatrix} 1 \\ 1 \end{pmatrix} \]

since \( d(x,z) = a \) and \( d(y,z) = b \).

This gives us the following 3 equations.

\[ a = c_{xy} + c_y + c_o \]
\[ b = c_{xy} + c_x + c_o \]
\[ j = c_{xy} + c_x + c_y \]

The last equation follows from the fact that \( x_k \neq y_k \) for \( 1 \leq k \leq j \).

If we solve with \( d' = a - c_o \) and \( b' = b - c_o \) we get the following matrix equation:

\[ \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} c_{xy} \\ c_x \\ c_y \end{pmatrix} = \begin{pmatrix} d' \\ b' \\ j \end{pmatrix} \]

By inverting the matrix we can compute values for \( c_{xy}, c_x, \) and \( c_y \):

\[ \begin{pmatrix} c_{xy} \\ c_x \\ c_y \end{pmatrix} = \begin{pmatrix} 1 & 1 & -1 \\ -1 & 0 & 1 \\ 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} d' \\ b' \\ j \end{pmatrix} = \begin{pmatrix} d' + b' - j \\ j - d' \\ j - b' \end{pmatrix} = \begin{pmatrix} a + b - j - 2c_o \\ j - a + c_o \\ j - b + c_o \end{pmatrix} \]

where the number of fields where \( z \) differs from \( x \) but not from \( y \) is \( c_x \), the number of fields where \( z \) differs from \( y \) but not \( x \) is \( c_y \), and the number of fields where \( z \) differs from both \( x \) and \( y \) is \( c_{xy} \). We now look at the different ways of choosing fields in \( z \) that satisfy these constraints. Of the \( j \) fields where \( x_k \neq y_k \) we must choose \( c_x \) where \( z \) differs from \( x \) but not from \( y \). Of the remaining
(j - c_x) we must choose c_y fields. The rest of the c_{xy} fields are pre-chosen. Of the (2n - j) fields
where x and y do not differ we must choose c_o fields where z differs.

The values of the C_x and C_y fields are chosen to be the values of the opposite string, that is, for
the c_x chosen fields the value is that of y and vice versa for the c_y chosen ones. The remaining C_{xy}
fields can take any value except those of the corresponding fields in x and y, which leaves (n - 2)
choices for these c_{xy} fields. The C_o fields can take any value except that of the corresponding fields
of x and y (which are the same!), that is, there are (n - 1) different choices for these values.

By substituting these values in equation (2) and summing over all valid values of c_o (i.e. values
for c_o that produce non-negative values for c_{xy}, c_x and c_y), we get the number of configurations with
distance a to x and distance b to y. The Iverson function \([c_x \geq 0 \land c_y \geq 0 \land c_{xy} \geq 0]\) assures valid
values of c_{xy}, c_x and c_y.

Taking into consideration that \(c_o \leq \frac{a+b+2i}{2}\) and \(c_o \leq 2i - a\) (See equation (3)) we arrive at
\[
\sum_{i=2}^{e} |D(v_c, 2i)| \sum_{b=0}^{e} \min\{\frac{a+b+2i}{2}, 2i-a\} \sum_{c_o=0}^{e} O(i, a, b, c_o) =
\sum_{i=2}^{e} \binom{n}{i} c_i \sum_{b=0}^{e} \min\{\frac{a+b+2i}{2}, 2i-a\} \sum_{c_o=0}^{e} O(i, a, b, c_o)
\]  
(4)

where \(c_i\) are the constants from Lemma 4, and
\[O(i, a, b, c_o) = [c_{xy} \geq 0][c_x \geq 0][c_y \geq 0] 2i c_x c_y (n-2)^c_{xy} (2(n-i) c_o) (n-1)^c_o\]

and c_{xy}, c_x, c_y are given as follows:
\[c_{xy} = (a - c_o) + (b - c_o) - 2i\]
\[c_x = 2i - a + c_o\]
\[c_y = 2i - b + c_o\]
<table>
<thead>
<tr>
<th>$n$</th>
<th>Wrong</th>
<th>Ambg.</th>
<th>Wrong</th>
<th>Ambg.</th>
<th>Wrong</th>
<th>Ambg.</th>
<th>Wrong</th>
<th>Ambg.</th>
<th>Wrong</th>
<th>Ambg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>54.55%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.00%</td>
<td>0.00%</td>
<td>16.44%</td>
<td>8.22%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.00%</td>
<td>0.00%</td>
<td>10.83%</td>
<td>2.17%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.00%</td>
<td>0.00%</td>
<td>7.10%</td>
<td>0.79%</td>
<td>21.40%</td>
<td>1.59%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.00%</td>
<td>0.00%</td>
<td>4.91%</td>
<td>0.35%</td>
<td>14.83%</td>
<td>0.75%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0.00%</td>
<td>0.00%</td>
<td>3.57%</td>
<td>0.18%</td>
<td>10.79%</td>
<td>0.40%</td>
<td>19.89%</td>
<td>0.26%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.00%</td>
<td>0.00%</td>
<td>2.70%</td>
<td>0.10%</td>
<td>8.17%</td>
<td>0.23%</td>
<td>13.51%</td>
<td>0.11%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0.00%</td>
<td>0.00%</td>
<td>2.11%</td>
<td>0.06%</td>
<td>6.38%</td>
<td>0.14%</td>
<td>9.54%</td>
<td>0.05%</td>
<td>19.87%</td>
<td>0.04%</td>
</tr>
<tr>
<td>10</td>
<td>0.00%</td>
<td>0.00%</td>
<td>1.70%</td>
<td>0.04%</td>
<td>5.12%</td>
<td>0.09%</td>
<td>6.97%</td>
<td>0.03%</td>
<td>13.62%</td>
<td>0.02%</td>
</tr>
</tbody>
</table>

Figure 2.17: Failure rate for algorithm - Incorrect fixes and ambiguous fixes.
Using equation (4) we can not compute an upper bound for the fraction (1). Figure 2.17 shows the estimated failure rate for the algorithm. An ambiguous fix is when more than one valid configuration is at the minimum distance.

2.5.2 Millipede Online Protocol Error Detection Module

The MOPED (Millipede Online Protocol Error Detection) module is partially implemented, but not fully designed, specified or implemented.

It accepts a specification of a protocol and at runtime checks every sent message against the protocol rules.

However, there is still work to be done here, but for completeness I will briefly give a few examples of MOPED verification files.

2.6 Expected Future Publications

I expect to write and submit conference papers on both the MOPED module as well as the message module during the next 18 months. A journal paper on the entire Millipede system and the multi level debugging methodology is also expected.
Chapter 3

Review of the Literature

In this chapter I will briefly describe some of the existing approaches to parallel debugging and parallel system development. I will try to point out any shortcomings these tools might have and compare them with the theory of errors and debugging from Chapter 1.

3.1 Program Development Environments

A well known approach to writing programs is to use integrated development environments. Some well known examples in the sequential world include the integrated development environments from Borland (Turbo Pascal 5.0 and up) and Visual C/C++, and Visual Basic from Microsoft. Not only do these environments offer support for program development, but they all come with built-in debuggers. The idea of developing programs and complicated systems through a development environment also extends to the parallel programming domain.

One of the most important tasks of a program development tool is to allow the user to develop programs in a structured way using some high level abstraction, such as graphs, for example. An important side effect of the structure and high level of abstraction is a lowered risk of introducing certain error types. A good example of this is the ability found in certain tools [BB97] to always create dead-lock free message passing code by always ensuring that all send calls are matched with receive calls. The high level of abstraction allows the user to concentrate on higher levels of the program design, for example, the function/control or data decomposition of the program,
depending on which abstraction is adopted by the environment. Often the structure and abstraction level offers relatively easy debugging of certain types of errors within the environment; this is a very desirable quality from a debugging point of view.

Unfortunately, the concepts that make development environments desirable also have downsides. An environment structured around a high level of abstraction is a good tool for program development, if the abstraction of the task at hand matches that of the environment. For example, if the environment is structured around the data flow model and the program being implemented is structured according to the control flow model, then the task of implementing an algorithm that adheres to one abstraction in an environment that does not adhere to the same abstraction becomes complicated and cumbersome. One example of this problem emerges when trying to write programs that make use of explicit message passing, using a tool that supports the data flow model (entire blocks of data flows between functional units in the program representation of the tool). If it is even possible to implement the program it will be conceptually difficult and artificially structured.

The problem with structure and abstraction of the tool not fitting that of the program being implemented is one of the most common reasons for not using such tools [Pan94]. Another reason is that users are often conservative and hesitant to learn new environments [Pan94]. A last example of why environments can sometimes be less desirable to use is the final code generated. If the tool supports a source to source transformation, for example, from the tool abstraction to C source code, this code can be hard to read or illogically structured because of the automated code generation. Even worse, sometimes no source code is available, and that limits further development and maintenance without using the original programming tool.

Some examples of environments that have adopted the data flow model as a main abstraction are CODE [NY93, NC92], HeNCE [ea], and TRAPPER [ea97]. The abstraction is based on data flowing between functional units or entire processes of the system.

Like CODE, HeNCE, and TRAPPER, PVMbuilder [BB97] and VPE [ND94] are other environments that use graphs to represent programs. However, these two tools both allow explicit message passing and the abstraction adopted leans more towards the control flow model than the data flow model.

The abstraction used in ENTERPRISE [SSS90] and FRAMEWORKS [SSG91] is that of tem-
plates. This is a strong concept but unfortunately, in this case, communication specification is completely left to the user, so the number of errors does not seem to be reduced by imposing an abstraction.

All of the tools strive to make parallel programming easier, that is, to reduce the number of errors, and take away much of the work with respect to explicit message passing from the user. Unfortunately, when this is the goal, the user’s freedom and expressiveness gets reduced: the safer you want a development tool to be (the more you want the tool to do for you without having to think for yourself) the more restrictive it becomes, and the more limited the expressiveness becomes. In other words, the higher the level of abstraction and the more rigid the structure of a tool, the smaller the set of easily implementable programs. The greater the set of programs the user wishes to implement, the more general the environment must be. At one end of the spectrum, tools are specifically designed towards a certain type of program with a very rigid structure; at the other end, 100% manually coded programs exist where the programmer himself supplies message passing calls using, for example, PVM or MPI.

3.2 Visualisation Tools

One class of tools that can be used for not only performance tuning but also debugging is the family of visualisation tools. Visualisation tools are categorised by their ability to provide the user with information about a program’s behaviour.

A typical tool offers a fixed set of views, each displaying different information about the system in various ways, such as graphs, charts, and so on. A visualisation tool that supports message passing views can be utilised when the programmer searches for errors involving stray messages or simply erroneous protocol specifications. Such tools are also excellent tools if the programmer is trying to obtain a global view of the entire system.

However, often global views are much too large for a programmer easily to be able to locate errors. These types of tools are faced with the very difficult task of providing a vast amount of information in an easy to understand way. This problem has been addressed by Cherri Pancake in [Pan99], and some of the more serious issues pointed out include not only difficulty with presenting large amounts of data, but also the ability, or lack there of, to zoom into views, extract
lower level information, and to map these displays/views back to the source code which created the error.

This is similar to having gathered data as in dimension 2 from Chapter 1, but the data is not directly understandable, which is caused by the inapplicability of the tool (dimension 1 from Chapter 1). In addition, there are also problems associated with the amount of data that can be displayed, the type of displays, and at least for tuning, the problem of perturbing the execution of the program. The last problem can appear in any tool based on a software monitoring and runtime collection of information; however, it is more critical in the case of performance tuning and identifying performance problems.

Another problem is the lack of user defined views; that is, a lot of visualisation tools support a limited pre-programmed set of views. I believe that this directly contradicts one of the design goals set by Eisenstadt in [Eis97]: the possibility of a tool to offer a view of what the user wants, when she wants it, not just the information that the programmer of the tool thinks might be useful to the user.

Examples of these tools include Paradyyn [MHC94], Vampir [NAW+96], and ParaGraph [HE93]. One visualisation tool specific to PVM is XPVM [KG96], which uses the tracing facilities available in PVM 3.4, and offers a nice graphical user interface to dynamically visualise network status, utilisation, message queues, and much more. These tools all use graphical representations to display program behaviour.

Millipede attempts to solve this problem by letting the user specify query statements based on the messages, data sent/received and program structure/source code. I believe that by allowing the user to create queries directly related to an error, the error can be more easily located.

### 3.3 Extension of Sequential Debuggers

In this section I will look at the family of debuggers that are extensions of well known sequential debuggers. I have divided this class of tools into two categories: debugging environments and N-version sequential tools.
3.3.1 Debugging Environments

A number of debugging environments exist which support parallel debugging. These environments are typically extensions of sequential debuggers. This means that the set of operations available in these tools are well because of familiarity with standard sequential debuggers. These include stepping, breakpoints and variable inspection. The biggest difference is that these tools operate on several processes at a time, thus allowing collective breakpoints over multiple pieces of source code and macro stepping (several processes all stepping through one line of program code at the same time). The force of these tools is their ability to control multiple processes at the same time. This can be a problem for the user; he has to keep track of a large number of processes simultaneously, which blurs the focus of the debugging task. Even though these tools support the common set of debugging activities, they all require the user to learn a new environment with its own graphical interface. Furthermore, the focus is on the the sequential code, not on the entire parallel system. That is, the granularity (as defined in Chapter 1) cannot be varied, but is set to ‘fine grained’; only the sequential debugging task is supported. In a sense, these parallel debugging environments can be said to often suffer from the exact opposite problem as the visualisation tools: the granularity is too fine and the focus is on the source code always. Such tools therefore get placed into the ‘tool inapplicable or hampered’ category of dimension 1. I believe that this category is as big as it is exactly because of the set granularity experienced in most tools.

Examples include DIWIDE [KLK99] and TotalView [Gmb99]. The DIWIDE debugger is a parallel debugger that implements collective breakpoints and macro steps (collectively stepping over program parts). It allows the programmer to treat a collection of processes as one, and allows the user to easily issue global commands and set global breakpoints. TotalView, a commercial product, is a multi process, multi threaded tool for on-line source code debugging.

3.3.2 N-version Debuggers

A naïve approach to parallel debugging is the use of $N$ copies of a sequential debugger like `gdb/dbx` – one for each process. The disadvantage of providing $N$ versions of a sequential tool is the overwhelming amount of information. In addition, how this vast amount of information gets presented to the user is often not appropriate for the task at hand [Pan99]; it is not as easy for the user to
focus on one particular process in the system when attending to all of the them. The complexity of
the program development process alters drastically when parallelism is introduced, and the prob-
lems are heightened by the relative instability of current parallel runtime environments [Pan94].
This suggests that debugging a parallel program while all the processes are running concurrently
may be too difficult and one tailored more to specific processes in the system would be better. In
addition, the granularity cannot be varied and the user is left with the functionality of a sequential
tool, which might not be applicable for a parallel debugging task.

Another apparent lack is the inability to supply different views of whatever information the user
might want; although all variables and program texts are available (which can be a great advantage
when debugging low level sequential code), this information is spread over $N$ windows and not
readily available for queries. It would take an overwhelming amount of time for the programmer
to extract, collect and interpret the information available. In addition, if the focus is on one single
process the debugging views are not needed for the rest of the processes.

An example is pdbx [pdb] for the SP/2, which is a front end for multiple instances of the UNIX
debugger dbx running on multiple nodes on an IBM SP system. As well, p2d2 [Hoo96] is a graph-
ical front end for multiple instances of the gdb sequential debugger, and has successfully been used
to debug systems with as many as 128 concurrent processes. It is possible to design a script for
PVM to allow users to execute gdb/dbx on every process spawned. However, that would require
a script for each sequential tool being supported, and it is more difficult in PVM to conditionally
spawn the debugger for a given set of processes without having to rewrite the code in the original
program.

To avoid the lack of focus and confusion caused by the vast amount of information presented
at a time, Millipede has taken a different approach. If debugging of sequential code is needed, the
Sequential Debugging Module can be used. The exact same behaviour of the debugged process, as
when run with the rest of the system, can be achieved by sequentially debugging the process with
Millipede’s replay mode switched on.
3.4 Replay Tools/Debuggers

Another major class of tools is the family of replay tools which allows the user to animate or replay the execution of a program. Replay tools collect information about the system as the program is executed; that is, messages are collected, time stamped and saved on secondary storage for replay use. When the tool then replays the execution, information about message content and program state is retrieved from the disk. A replay tool is typically considered an off-line tool, meaning it can be deployed once the program has finished running.

As seen before, many of these system have set granularity, thus focusing on, for example, the source code level, leaving the user helpless if debugging on a higher level is needed. The opposite can of course also be the case: when the focus is on the higher level of the system, and mapping the error back to the lower level is extremely difficult, because the tool does not readily support debugging at a lower level.

BUSTER [XWXS96] is one such post mortem replay system; it allows the user to re-execute the debugged program in different modes depending on the amount of control needed, without having to run the message passing system. A system like PVaniM [TSS96] is another PVM based graphical tool that supports both online debugging and post mortem visualisation of a parallel execution.

Other examples of tools that combine the online and off-line strategies of debugging into a more integrated environment are MAD [KV97], PDT from the Annai toolset [CFR95], and PDM [Arv92]. These tools provide more integrated environments for debugging while providing higher level tools for finding and correcting specific errors, such as communication errors. MAD [KV97] is a debugging environment based on event graphs and the manipulation of these. It allows debugging on various levels, from pattern of processes (groups of event graphs), to control flow graphs and source code. PDT is an interactive distributed source-level debugger for distributed memory parallel processes in the Annai [CFR95] tool set, and allows both online debugging and off-line replay. The PDM [Arv92] system is a framework for detecting communication-related errors in concurrent Occam programs running on a Transputer network.

The major disadvantage of these tools is again the massive amount of information and the need to learn a new environment. The replay mechanism is extremely useful. However, unless other
tools can be used in connection with a replay, it becomes virtually impossible to accomplish a specific debugging task unless the tool specifically supports it. Again, if the replay mechanism was merged with some of the techniques described in earlier subsection in this chapter, it would make for a stronger and more flexible tool.

### 3.5 Relative Debuggers

For the sake of completeness I wish to mention the concept of relative debugging. Relative debugging is a technique often used when porting programs to different architectures, thus allowing the execution of 2 different versions of the same program on 2 different machines at the same time. Guard [SA97, WA98] is such a debugger. It will execute 2 different instances of the same program on 2 different machines and thus allows the programmer to compare the contents of variables and more while the programs are executing. Relative debugging is mostly useful for the programmer who ports existing programs to different architectures or operating systems, and thus this technique is not applicable when it comes general parallel debugging.

### 3.6 Language Support for Communication

A different approach altogether is writing programs using languages with built in support for parallelism. A well known example of this is the $\mu$C++ language[BS95] from the University of Waterloo. $\mu$C++ extends C++ with new language constructs to express parallelism and provides a runtime system that runs a programs concurrently or in parallel when appropriate hardware is available. However, the need to debug is still present.

The work in [BK95, Kar95] describes debugging and performance tools for $\mu$C++ in greater detail. A debugging session of a program written in $\mu$C++ is comparable to those found in DIWIDE and TotalView; a front end to a number of instances of well known sequential debuggers like (gdb) attached to each process being debugged.

Other well known examples of languages that support communication include CML [RWZ88] (Concurrent ML), and Facile [GMP89a, GMP89b] (ML with higher-order concurrent processes based on CCS), both functional languages.
However, these systems are of more interest when seen from a theoretical perspective, as they both lack the performance needed for large parallel systems. Lack of performance is a well known side effect of functional programming languages; this is a an unfortunate trade-off as functional programs are more easily verified by program verification tools, thus reducing the need for debugging.

### 3.7 Summary of Related Work

The following points summarise the problems with many of the existing tools:

- Restrictive interfaces that support a number of predefined tasks.

- The data gathered need to be interpreted by the user to map the error back to the cause, which often renders the tool less useful. In other words, the cause/effect concept is not well supported. In [KV97], it is argued that the original source code is a good basis for debugging activities, since it contains the cause of the wrong behaviour.

- A fixed, often small grained, number of tasks are supported. Fixed granularity in connection with restrictions on the interface makes debugging at higher levels almost impossible.

- Information overload: the amount of information presented can be so large that time needed to find the information becomes unmanageable.

What the user wants is not always available in any of the reviewed systems. Each of the systems have strong points and can be very useful for certain tasks. Unfortunately, applying different selections of tools from different tool sets is an impossible task – different user interfaces, different representations, different formats and so on makes changing between tools for different debugging tasks impossible. This means that the user must choose only one, or at best a small number of tools, which might not be preferred for the debugging task.
Chapter 4

Plan for completing this work

In Figure 4.1 the following abbreviations are used:

- **SDM** — Sequential Debugging Module.
- **MDM** — Message Debugging Module.
- **DCM** — Deadlock Correction Module.
- **MOPED** — Millipede Online Protocol Error Detection Module.

Both the **DCM** and the **MOPED** modules are part of the Protocol Debugging Module. The **Dissertation** block covers the writing process associated with finishing this work, and the **Paper** blocks cover the writing of the papers mentioned earlier as future publication.
Figure 4.1: Time line for completing this work.
Bibliography


