Communicating Process Architectures 2006
Peter H. Welch, Jon Kerridge, Frederick R. M. Barnes
IOS Press, 2006

Classification of Programming Errors in Parallel Message Passing Systems

Jan B. PEDERSEN
University of Nevada
4505 Maryland Parkway,
Las Vegas, Nevada, 89154
United States of America
Tel.: +1 702 895 2557; Fax: +1 702 895 2639
E-mail: matt@cs.unlv.edu

Abstract. In this paper we investigate two major topics; firstly, through a survey given to graduate students in a parallel message passing programming class, we categorize the errors they made (and the ways they fixed the bugs) into a number of categories. Secondly, we analyze these answers and provide some insight into how software could be built to aid the development, deployment, and debugging of parallel message passing systems. We draw parallels to similar studies done for sequential programming, and finally show how the idea of multilevel debugging relates to the results from the survey.

Keywords. Parallel Programming Errors, Debugging, Multilevel Debugging, Parallel Programming.

1. Introduction

"If debugging is the process of removing bugs, then programming must be the process of putting them in"; This well known quote from Dijkstra was probably said in jest, but seems to hold some amount of truth.

A number of papers have been written about debugging, both for sequential and parallel programming, but many of the debugging systems they describe are not being used by the average programmer. Numerous reasons for this are given, and they include restrictive interfaces, information overload, and wrong level of granularity. They also fail to take into account the types of the errors that occur in parallel message passing programs by primary focusing on well known sequential errors. In other words, the same level of granularity is used for all error types, which we believe is a big mistake. The first step to a solution to this problem is to obtain a better understanding of the type of errors that programmers encounter.

To better understand this problem, a class of graduate students at the University of Nevada, Las Vegas, answered a questionnaire about their experiences with programming parallel message passing programs. We also asked them to report every single runtime-error they encountered throughout the semester. Along

with each report they submitted information about the cause of the error, how the error was found, and how long it took to find.

In this paper we present the analysis of the error reports and the questionnaire along with a number of suggestions for how development environments and in particular debuggers can be developed around these error types. In addition, we argue how multilevel debugging can be a useful new debugging methodology for parallel message passing programs.

Thus we propose, as tool developers, that we need to understand the types of errors that our clients (the programmers) make. If we do not understand these errors we cannot provide tools and techniques that will carry a big impact.

2. Related Work

2.1. The Parallel Programming Domain

Parallel programming involves a set of components that must each be considered when developing a parallel system. This set, which we regard as the parallel programming domain, includes, among others, the following aspects of the code: sequential code, interprocess communication, synchronization, and processor utilization. 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. In addition, it helps determine where to focus the debugging effort, depending on which component of the domain the programmer looks for errors in.

In [Fos95] a four stage model for constructing a parallel program, referred to as PCAM, representing the parallel programming domain, is suggested. The four components are:

1. **Partitioning.** The computation to be performed and the data which it operates on are decomposed into small tasks.
2. **Communication.** The communication required to coordinate task execution is determined, and the 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 outside the scope of this paper.

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

1. **Algorithmic changes.** Many parallel programs begin life as a sequential program. If parallel algorithms are based on, or derived from, existing algorithms and/or programs, then a transformation from the sequential to the parallel domain 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. For example, the pipe-and-roll matrix multiplication algorithm [FJL+88] does not have a sequential counterpart.
2. **Data decomposition.** When a program is re-implemented, the data is distributed according to the algorithm being implemented. Whether it is the transformation of a sequential program or an implementation of a parallel algorithm from scratch, data decomposition is a nontrivial task that cannot be ignored when writing parallel programs, as not only correctness, but efficiency also greatly depends on it.
3. **Data exchange.** As parallel programs consist of a number of concurrently executing processes, the need to explicitly 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 a separate memory, message passing becomes necessary.
   When message passing systems like MPI [Don94] and PVM [Gei94] are used, the programmer is responsible for a number of different tasks: specifying the correct IDs of the involved processes, packing messages into buffers, using the correct functions to pack the data depending on the type, and assigning tags to the message. In part, the difficulty of using a message passing library like PVM and MPI is the low level of the interface of the message passing system.
4. **Protocol specification.** The protocol for a parallel system is defined as the content, order, and overall structure of the message passing between communicating processes. Along with the data exchange, the communication protocol of the program is a new concept that has been introduced by parallelizing the algorithm.

Figure 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 either be 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

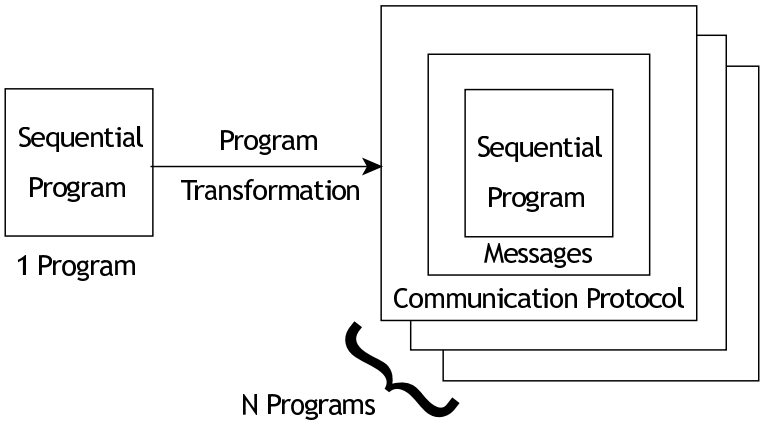

Figure 1. The sequential versus the parallel programming domain.

exchange), and the outer box represents the protocol that the communicating processes must adhere to.

2.2. The Debugging Process

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

1. **Initial hypothesis set.** The programmer creates a hypothesis about the errors in the program, including the locations in the program where errors may occur, as well as a hypothesis about the cause, 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 and the significance of the error.
4. **Hypothesis verification.** The hypothesis is verified or discarded using one or more of the four 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 four, the process is repeated from step two. In the above model, step four, hypothesis verification, is the focus of our work. Step one can in some situations be automated to a certain degree; examples of such automation include the deadlock detection and correction presented in [BW01a].

2.3. The Why, How and What of Errors

M. Eisenstadt describes in [Eis97] a 3-dimensional space in which sequential errors are placed according to certain criteria. This classification shows some interesting results, which we briefly summarize. 51 programmers were asked to participate in a study in which programming errors are placed into a 3-dimensional space. The 3 dimensions are:

- **Dimension 1**: Why is the error difficult to find?
- **Dimension 2**: How is the error found?
- **Dimension 3**: What is the root cause of the error?

For dimension 1 29.4% fell in the category *Cause/effect chasm*. What makes the errors hard to find is the fact that the symptom of the error is far removed in space and time from the root cause. The second most frequent answer was *Tools inapplicable or hampered*, which covers the so called 'Heisen bugs' [Gra86]. It is notable that over 50% of the cases are caused by these two categories. The first category, the cause/effect chasm is greatly amplified in the parallel programming domain, and the second category is, as we have already pointed out, one of the problems we are researching.

Dimension 2, concerned with how an error was found; the most frequent answer was *Gathering data* (53% of answers fell in this category). This category covers the use of print statements, debuggers, break points etc. The second most frequent answer was *Inspeculation*, which covers hand simulation and thinking about the code. 25.5% of answers fell in this category.

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 (in Eisenstadt's study). This result corroborates the result of Pancake [Pan94]: up to 90% of all sequential debugging is done using print statements.

While the use of print statements is straightforward when working with sequential programs, their use in parallel programs is often more complicated. Often, processes run on remote processors, which makes redirecting output to the console difficult. Even when output can be redirected to the console, all processes are writing to the same window, thus making the interpretation of the output a challenging task. This is an example of the information overload theory mentioned earlier. Furthermore, the order of the output (i.e., the debugging information from the concurrently executing processes) is not the same for every run, as the processes execute asynchronously and only synchronize 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 a process crashes, the buffer might not be flushed, thus missing output written by the program. Of course this can be solved by inserting calls to flush the I/O buffers, but if these are missing, the programmer ends up spending time on debugging the code he added for debugging purposes! 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 can then be wasted looking for an error in a place where no error can be found.

The third dimension, the root cause of the error, contains 9 different categories; the most noteworthy is the most frequent one, *Memory*, which covers errors such as overwriting a reserved portion of the memory causing the system to crash, and array subscripts out of bounds. 25.5% of answers fell on this category. The second most frequent root cause was faulty hardware (with 17.7%) and in third and fourth place, with 13.7% and 11.8% respectively, came faulty design logic (Algorithmic design/implementation problems) and initialization, which covers wrong types, redefinition of the meaning of system keywords, or incorrectly initialization of a variable.

Nearly 50% of the errors are caused by the first two categories. This also perfectly agrees with previous studies where tools and runtime systems are 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 analyzing 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 nine categories.

Although all errors essentially trace back to a piece of sequential code that executed on a processor somewhere in the parallel system, we should still consider the errors that occur at conceptually higher levels of the parallel programming domain. By ignoring the higher levels and attempting to use tools from a lower level we often achieve information overload or other problems. Even though a protocol error is caused by sequential code somewhere in the system, such errors are easier found if the level of granularity is that of the protocol. Naturally, it is vital that the tool at this level can map the error back to the sequential code as the correction will have to be made here. (This is one of the main design goals of multilevel debugging)

If we accept the decomposition of the parallel programming domain as we stated it above, as well as the overall debugging technique of hypothesis development and verification, we still need to gather information about the error types like Eisenstadt did for sequential errors. This is the study presented in the following sections.

3. The Framework

The main goal of this research is to clarify a number of subjects related to parallel programming and debugging of parallel programs. First of all, we wish to obtain some insight into the types of errors the programmers encounter, and secondly obtain data about the techniques they used to locate and correct them. We believe that this information serves as a good basis for how programming and debugging tools for parallel (message passing) programs should be developed. It is important to understand the programming domain (in this situation, the parallel programming domain with message passing) in order to make qualified decisions about how to correct the errors.

The subject of error types are useful for a tool developer in a number of ways. First and foremost, if a large percentage of errors are of a certain type, it is

important to tailor the tools to assist the user in locating and correcting this type of errors rather than a different type that might not occur as frequently. Secondly, it gives the tool developer an idea of where the errors are located, that is, are most errors in the sequential code, are they related to the data decomposition, the functional decomposition or could they be relate to the use of the message passing API. Such information is invaluable to developers of programming environments as well. It pinpoints the area where the tool has the greatest chance of having an impact on the development cycle.

One of the main reasons for this research is a result by Cherri Pancake [Pan93] which states that tools for parallel programming/debugging are often only used by their developer. She claims that this is caused by the fact that the tool developer and the tool user might have different foci on what they want/need from a tool.

4. The Error Reports

The programmers were asked to submit a small web questionnaire about all the run-time errors they encountered throughout the semester. These were submitted though a simple web interface, and contained just three questions:

- Describe the bug.
- How did you find/fix it?
- How long did it take?

We attempt to mimic the study by Eisenstadt as closely as possibly by asking how the error was found and what caused it. There questions are close to dimensions 2 and 3 of Eisenstadt's questionnaire.

4.1. The Programs

In this section we briefly describe the six different programs the programmers wrote throughout the semester. The following list (in no particular order) gives a brief description of the programs

- **Equation Solver** — Using one master and n slave processes to solve a upper triangular system of equations.
- **Mandelbrot** — Using one master and n slaves in a work farm model to compute a Mandelbrot set.
- **Matrix Multiplication** — Implement the Pipe-and-Roll [FJL+88] matrix multiplication algorithm.
- **Partial Sum** — Implement a partial sum algorithm that runs in time $O(log\ n)$.
- **Pipeline Computation** — Using functional decomposition, implement a multistage pipeline with dispersers and collectors that allow for multiple instances of some stages of the computation to achieve a good load balance.
- **Differential Equation Solver** — Solve a differential equation using a discrete method.

Depending on the type of the error, we categorize the *"Describe the bug"* question into seven different categories. We chose seven different categories based on the two first categories in the PCAM model [Fos95], namely partitioning and communication. The partitioning is further subdivided into data decomposition and functional decomposition, and the communication is divided into API usage as well as the three major levels of the parallel programming domain: sequential, message, and protocol. Finally a category for errors (*other*) that do not fit any other category was added. In more detail, the seven categories we chose are:

- **Data Decomposition** — The root of the bug had to do with the decomposition of the data set from the sequential to the parallel version of the program.
- **Functional Decomposition** — The root of the bug was the decomposition of the functionality when implementing the parallel version of the program.
- **API Usage** — This type of error is associated with the use of the MPI API calls. Typical errors here include passing data of the wrong type or misunderstanding the way the MPI functions work.
- **Sequential Error** — This type of error is the type we know from sequential programs. This includes using `=` instead of `==` in tests etc.
- **Message Problem** — This type covers sending/receiving the wrong data, that is, it is concerned with the content of the messages, not the entire protocol of the system.
- **Protocol Problem** — This error type is concerned with stray/missing messages that violate the overall communication protocol of the parallel system.
- **Other** — Bugs that do not fit any of the above categories are reported as 'other'. This include wrong permissions on programs to be spawned, faulty parallel IO etc.

We believe that this breakdown will reveal a lot of information about where the bugs are located and where focus should be placed in the development and debugging process. It should be clear that the first 3 items are issues that could be aided in the development process where as the next three should have strong debugging support. This partitioning of course does not rule out development support for message and protocol problems or debugging support for data of functional partitioning.

The base for all these programs was either a sequential program (Equation Solver, Mandelbrot, Differential Equation Solver) or a abstract parallel algorithm. The program were to be implemented in C using the MPI [Don94] message passing interface.

5. The Questionnaire

The second part of the Survey was a questionnaire given at the end of the semester. The objectives of this questionnaire were to discover out what the programmers thought was the hardest topic, to learn about their general debugging habits, and to obtain a picture of the type of errors they perceive as being the most

frequently encountered. Furthermore, we asked for a wish list with respect to the functionality of development and debugging tools. The survey contained the following 6 questions:

1. Please mark the level of difficulty for each of the following points (1=easy, 5=hard):
   - Data decomposition
   - Function decomposition
   - Communication Calls
   - Debugging the code
2. What do you think is the hardest part of developing a parallel program?
3. List the 3 types of errors you encountered the most.
4. What was your main approach to debugging.
5. What sort of programming support would you find useful (not debugging).
6. What sort of debugging support would you find useful?

The answers to these questions should give an indication of what the programmer perceives to be hard, and when compared to the actual error reports, it will show if their perception of parallel message passing programming is correct. In addition, it will be revealed if the errors they think they get most frequently are indeed the errors they reported.

6. Result of Error Reporting

Table 1 summarizes the results of the online error reporting survey and figure 2 shows a graphical representation of the result.

	Data Decomp.	**Functional Decomp.**	**API Usage**	**Sequential Error**	**Message Problem**	**Protocol Problem**	**Other**
Equation	2	0	1	0	1	3	4
Solver ($n_1 = 8$)	20.00%	0.00%	12.50%	0.00%	12.50%	37.50%	12.50%
Mandelbrot	10	7	11	5	5	7	4
($n_2 = 49$)	20.41%	14.29%	22.45%	10.20%	10.20%	14.20%	8.16%
Matrix	9	3	6	2	3	10	0
Mult.($n_3 = 33$)	27.27%	9.09%	18.18%	6.06%	9.09%	30.30%	0.00%
Partial	2	2	3	7	3	8	1
Sum ($n_4 = 26$)	7.69%	7.69%	11.54%	26.92%	11.54%	30.77%	3.85%
Pipeline	0	1	1	1	0	1	0
Comp.($n_5 = 4$)	0.00%	25.00%	25.00%	25.00%	0.00%	25.00%	0.00%
Differential	3	0	9	8	4	9	2
Eq. ($n_6 = 35$)	8.57%	0.00%	25.71%	22.86%	11.43%	25.71%	5.71
Total	**26**	**13**	**31**	**23**	**16**	**38**	**8**
($n = 155$)	**16.77%**	**8.39%**	**20.00%**	**14.84%**	**10.32%**	**24.52%**	**5.16%**

Table 1. Results of the online error reporting survey.

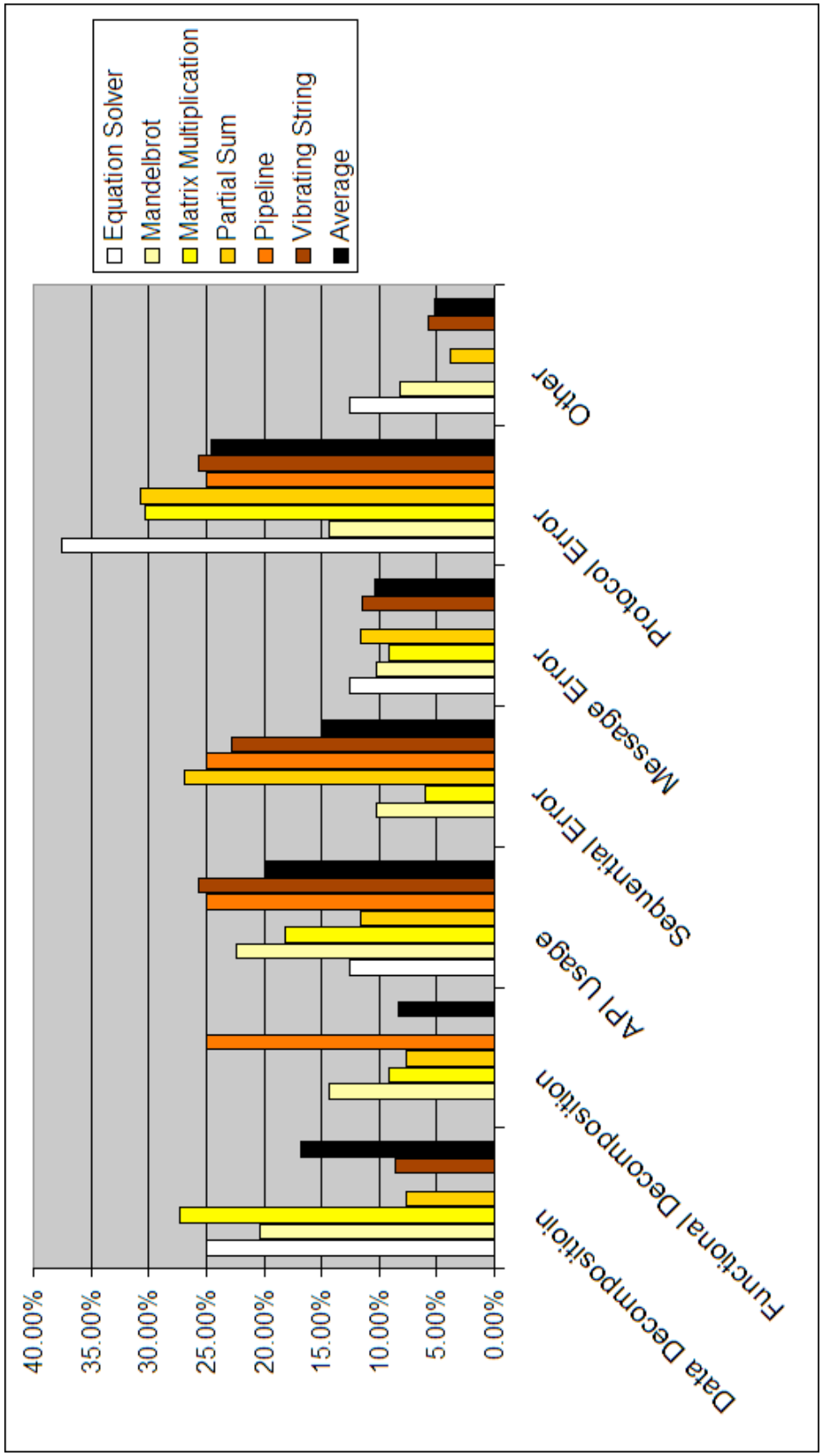

Figure 2. The results of the error reports.

As can be seen, 16.77%+8.39% = 25.16% of errors fall in the decomposition categories, and 14.84%+10.32%+24.52%=49.68% fall into one of the three parts of the decomposition of the PCAM model (all non-decomposition errors should fall into either of these categories! Here we do not count the API usage category). It is notable that almost one quarter of the errors are protocol related. This means that serious support is needed at this level. Approximately 15% of the errors are sequential, and if counting the API usage errors we reach a total of almost 45% of errors that a directly linked to faulty sequential code. This strongly suggests that support for sequential debugging is extremely important; Of course such support should not suffer from the problem current tools do (e.g., information overloading and granularity mismatch).

As a side note, we believe that it would be reasonable to assume that the 20% of the errors that fell in API Usage category will be reduced as the programmer becomes more familiar with the message passing API. We are certain that many API usage errors could be reduced through the use of a development environment that can aid the programmer in choosing the right values/types for the arguments.

6.1. Debugging Time / Print Statements

The third question on the error reporting page asked the programmers to give an estimate of the time it took to locate and correct the bug. The following table shows the result of this question along with the count of how many times the bug was located using regular print statements inserted in the code at strategical places.

Program	**Average Time**	**Print Statements**	**# Answers**
Equation Solver	43 minutes	2 (25.00%)	8
Mandelbrot	45 minutes	22 (44.90%)	49
Matrix Multiplication	47 minutes	14 (42.42%)	33
Partial Sum	63 minutes	19 (73.09%)	26
Pipeline Computation	28 minutes	0 (00.00%)	4
Differential Equation	61 minutes	16 (45.71%)	35
Total	**52**	**73 (47.10%)**	**155**

Table 2. Average debugging time and the use of print statements.

The error reports show a staggering 52 minute average for each of the 155 bugs there were reported. This is a lot higher than we suspected. This is also a good indication that it really is difficult to locate errors and correct them when dealing with a number of processes executing concurrently and communicating asynchronously, especially when not using any parallel debugging tools. For comparison, the average time it took to correct the 23 sequential errors reported was 37 minutes per error.

Table 2 shows that for certain problems (like the partial sum problem), the debugging task was accomplished by using print statements in 73.09% of the time. The average use of print statements for debugging was 47.10%. For sequential programs it was stated in [Pan93] that 90% of programmers still used print state-

ments as their primary debugging tool. The use of print statements as a primary debugging tool for parallel message passing programs can adversely impact the time the debugging takes. Not only can it be challenging to get the output rerouted to the console, but with a number of processes all using the same console, the output will be interspersed, and the interpretation of the output becomes more challenging.

7. Results of the Questionnaire

The first question of the end-of-semester questionnaire asked the students to rate the level of difficulty on a scale from 1 (easy) to 5 (hard) for 4 different topics. The results of this question can be seen in table 3.

Data Decomposition	Functional Decomposition	Communication Calls	Debugging
2.92	3.35	2.31	4.04

Table 3. Average level of difficulty for the topics in question 1 (out of 5 possible).

As expected, debugging proved to be the most difficult task associated with writing parallel message passing programs. This again emphasizes the need for techniques and tools for debugging parallel message passing programs.

The second question on the survey asked what part of developing a parallel program was regarded the hardest. The answers covered all 4 of the topics from table 3 with approximately 25% answering 'debugging'. To quote one of the answers: "Debugging, and debugging, and sometimes debugging."

The third question asked for a list of the three kinds of errors encountered most often. The answers that occurred the most frequent were: Processes die prematurely, incorrect data transferred, mismatch between sender and receiver, sequential errors, and incorrect API usage. The second most frequent answers (to question 3) include deadlock, process rank problems, and message tag problems.

For this question the answers fall into 3 clear categories: errors in the sequential code, errors associated with message content, and finally errors in the overall communication protocol. We will return to this division later and show why this is an important grouping, and how it has been used to develop a new debugging methodology.

The fourth question asked which technique was more frequently used for debugging. Most questionnaires had at least 2 different answers, but a staggering 100% of the questionnaires contained "print statements" as a primary or secondary debugging tool. This of course does not mean that all errors were located by the use of print statements, and reported in the previous section, although not 100% of all errors were caught and corrected using print statements, a large percentage was. The second most frequent answer for debugging approaches was various types of manual code inspection or manual code execution. Both print statements, code inspection and hand simulation are covered by Eisenstadt's two categories "Inspeculation" and "Gather Data", which for sequential programs ac-

counted for 25.5% and 53% respectively. That is, 78.5% of all errors in sequential programs were found using techniques in either of these two groups. According to the error report surveys for the parallel programs, 100% of how the error was found fall into one of these categories.

The last two questions asked what kind of programming and debugging support the programmer would like. Two answers stood out: Integrated programming/development environments and debugging tools specifically tailored to message passing programs, and not just a sequential debugger attached to each process.

8. Conclusions from the Survey

The overall result of the end-of-semester questionnaire was that most programmers still debug like they do when writing sequential programs, which unfortunately is primarily by the use of print statements. Furthermore, the errors seemed to fall into one of 3 main categories: Sequential errors, message errors, and protocol errors. There was an overall agreement that (better) IDEs and specialized debugging tools were the most useful programming/debugging support that a programmer could ask for.

8.1. The Error Reports

According to table 1, the error-type most often reported was 'Protocol Problem'. Surprisingly, the 'API Usage' takes second place, followed by 'Data Decomposition', 'Sequential Errors' and finally 'Message Problem', and 'Functional Decomposition'.

The 7 error categories can be grouped into two groups: i) Pre-programming/ planning problem, and ii) Actual programming errors. The former covers problems that fall in the data decomposition and functional decomposition categories, and the latter cover sequential errors, message problems, protocol problems and other problems. The first group is more related to the theoretical development/planning of the program as to the actual programming. If the data is laid out wrong or if the functionality has been decomposed incorrectly, no debugging can correct the problem. Thus, the second group contains the actual errors that can be corrected in the program text (i.e., which do not require a revision of the actual parallel algorithm).

the API Usage category was intentionally left out of the two categories listed above. We did this for a number of reasons. First of all, one can argue that the problems of using the API will disappear as the programmer gets more familiar with the message passing interface, and thus are not a real threat to programming development. On the other hand, one can argue that they constitute errors that can be fixed in the code, but they should be included in the second category. Often, an incorrect use of a message passing API can be caught and corrected by inspecting content of messages, so for now we chose to leave this category out.

8.2. Program Development

A little more than 25% of errors are attributed to problems associated with data or functional decomposition. This indicates that there is a genuine need for tool and/or techniques in this area to help the programmer reduce the number of errors and possibly reduce the development time.

This is a challenging problem to solve. One way of reducing errors of this type is to use development environments that support certain parallel patterns; however, it is not always possible to fit an algorithm to a known pattern.

Since we are focusing on the debugging issue we will leave this problem as a research topic for the future.

8.3. Program Debugging

The remaining 75% of errors (including API Usage) are associated with actual programming errors, that is, they require the programmer to actually debug the code to correct the error that causes the problem. The number of debugging tools in existence for parallel message passing programs have been unable to fully embrace a general debugging technique for such programs; in general existing debuggers can be divided into two categories: i) N-version debuggers, which are extensions of sequential debuggers where one debugging process is attached to each process in the parallel system. ii) Debugging environments, which are specialized environments that support debugging by the use of break points and macro stepping.

Some of the problem with N-version debuggers is the enormous amount of information that is displayed to the programmer. This often renders the use of this technique useless because of information overload.

The problem with both N-version debuggers and the environments is that their primary focus is on the sequential code. Naturally, all errors lead back to the code, but there is no support for higher levels of debugging, this includes errors related to messages or the protocol. Thus, the programmer is left to perform this debugging though an interface that is not meant for it. This makes debugging a very tedious and challenging task.

The results of the error-reporting page shows that approximately 15% of errors are sequential errors, so the perfect debugging tool of course must support debugging of the sequential code, but also, as almost 26% of the errors are concerned with messages or protocols, they must be capable of operating at higher levels that include messages, their content, and the overall communication protocol of the program.

These observations led us to formulate a new debugging methodologies developed around this break-down of the programming domain. We believe a successful tool must support debugging at the three different levels. At the sequential level it should be easy for the programmer to deploy a sequential debugger/tool or debugging technique related to sequential code without getting information overloading as with the N-version debugging technique. Similarly it should be easy to deploy tools specifically designed to locate and correct errors at the remaining two levels. We refer to this new technique as **Multilevel debugging**. In the following section we briefly describe the idea behind multilevel debugging.

9. Multilevel Debugging

In this section we briefly describe the multilevel debugging methodology. We developed this technique as a potential solution to the shortcomings of the existing debugging tools and techniques. We based the multilevel debugging methodology on the decomposition of the PCAM model as described earlier; this resulted in a bottom-up approach rather than the conventional top-down approach, which had proved fruitless because of problems such as information overloading. Some of the basic ideas behind multilevel debugging are:

- The information about messages and their content, as well as information about the protocol should be extracted from the running program and used when debugging errors at these higher levels.
- Support for mapping the manifestation of the error (the effect) back to the actual code that caused it should be provided.
- Strong support of sequential debugging of separate processes without causing information overloading must be provided.
- Automation of debugging tasks should be done when ever possible. Example of this include deadlock detection and correction.

Some of the general goals for multilevel debugging include

- Computable relations should be computed on request not left for the user to figure out.
- Displayable state should be displayed on request, not left for the user to draw or visualize.
- Views for important parts of the program (key players) other than variables should be available.
- Navigation tools tailored to specific tasks/levels should be available.

We have described a number of tools and techniques from the multilevel debugging framework in [BW00,BW01a,BW01b]. A complete reference can be found in [Ped03]

10. Conclusion

We have presented the results of 2 different surveys, and the result of these have shown that errors fit into 3 distinct categories: Sequential errors, message errors, and protocol errors. In addition, the survey showed that debugging by using print statements is still frequently used, and extremely time consuming.

We introduced a new debugging technique referred to as multilevel debugging, which decomposes the debugging task into three categories: sequential, message, and protocol. The results of the questionnaires and error reports seem to support the decomposition of the PCAM model chosen for multilevel debugging; Since multi level debugging include tools that are tailored to the levels associated with messages/message content and the protocol of the system, we believe that not only will it be easier to locate errors at these levels, but also reduce the time it takes to find and correct the error.

Multilevel debugging was initially developed as a new debugging methodology in [Ped03] and implemented in a text based version for PVM programs. A newer version, also for PVM, with a graphical user interface (Millipede) was presented in [Tri05], and an initial version for MPI (IDLI) has been presented in [Bas05].

11. Future Work

We wish to complete the MPI implementation and deploy it to programmers and have them use it and redo the survey to see if the errors were be corrected faster using these new debugging techniques.

In addition, as the surveys have shown, 25% of errors are associated with decomposition. It is clear that tools to assist the correct decomposition of data and functionality are needed.

Also supporting new tools through the use of recorded data from the program execution remains an interesting area of research. We strongly believe that a message passing library like MPI must incorporate debugging support directly in the library or through a system like Millipede or IDLI. The first step to making debugging 'easier' is to facilitate the extraction of necessary information about messages and the protocol from the message passing system itself.

References

[AFC91] K. Araki, Z. Furukawa, and J. Cheng. A General Framework for Debugging. *IEEE Software*, pages 14–20, May 1991.

[Bas05] Hoimonti Basu. Interactive message debugger for parallel message passing programs using lam-mpi. Master's thesis, University of Nevada, Las Vegas, December 2005.

[BW00] J. B. Pedersen and A. Wagner. Sequential Debugging of Parallel Programs. In *Proceedings of the international conference on communications in computing, CIC'2000*. CSREA Press, June 2000.

[BW01a] J. B. Pedersen and A. Wagner. Correcting Errors in Message Passing Systems. In F. Mueller, editor, *High-Level Parallel Programming Models and Supportive Environments, 6th international workshop, HIPS 2001 San Francisco, CA, USA*, volume 2026 of *Lecture Notes in Computer Science*, pages 122–137. Springer Verlag, April 2001.

[BW01b] J. B. Pedersen and A. Wagner. Protocol Verification in Millipede. In *Communicating Process Architectures 2001*. IOS Press, September 2001.

[Don94] J. Dongarra. MPI: A Message Passing Interface Standard. *The International Journal of Supercomputers and High Performance Computing*, 8:165–184, 1994.

[Eis97] M. Eisenstadt. My hairiest bug war stories. In *The Debugging Scandal and What to Do About It - Communication of the ACM*. ACM Press, April 1997.

[FJL+88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving problems on concurrent processors. General techniques and regular problems*, volume 1. Prentice Hall International, 1988.

[Fos95] I. Foster. *Designing and Building Parallel Programs: Concepts and tools for parallel software engineering*. Addison Wesley, 1995.

[Gei94] A. Geist et al. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.* Prentice Hall International, 1994.

[Gra86] J. Gray. Why do Computers Stop and What Can be Done About it? *Proceedings of 5th Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, January 1986.

[Joh83] W. L. Johnston. An Effective Bug Classification Scheme Must Take the Programmer into Account. *Proceedings of the workshop of High-level debugging. Palo Alto, California*, 1983.

[Knu89] D. E. Knuth. The Errors of TEX. *Software - Practise and Experience*, 19(7):607–685, July 1989.

[Pan93] C. M. Pancake. Why Is There Such a Mis-Match between User Need and Parallel Tool Production? Keynote address, 1993 Workshop on Parallel Computing Systems: A Dialog between Users and Developers, April 1993.

[Pan94] C. M. Pancake. What Users Need in Parallel Tool Support: Survey Results and Analysis. Technical Report CSTR 94-80-3, Oregon State University, June 1994.

[Ped03] Jan Bækgaard Pedersen. *MultiLevel Debugging of Parallel Message Passing Programs.* PhD thesis, University of British Columbia, 2003.

[SSP85] J. C. Spohrer, E. Soloway, and E. Pope. A Goal/Plan Analysis of Buggy Pascal Programs. *Human-computer Interaction*, 1(2):163–207, 1985.

[Tri05] Erik H. Tribou. Millipede: A Graphical Tool for Debugging Distributed Systems with a Multilevel Approach. Master's thesis, University of Nevada, Las Vegas, August 2005.