VISUAL ASSISTANCE FOR
CONCURRENT PROCESSING

BY

ROBERT F. ERBACHER
B.S., University of Massachusetts at Lowell (1991)
M.S., University of Massachusetts at Lowell (1993)

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF MASSACHUSETTS AT LOWELL

Signature of Author: ___________________________ Date: ____________

Signature of Dissertation Director:

____________________________ Date: ____________

Signatures of Thesis Committee Members:

____________________________ Date: ____________

____________________________ Date: ____________
VISUAL ASSISTANCE FOR
CONCURRENT PROCESSING

BY

ROBERT F. ERBACHER

ABSTRACT OF A DISSERTATION SUBMITTED TO THE FACULTY OF
THE DEPARTMENT OF COMPUTER SCIENCE IN PARTIAL
FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF SCIENCE IN COMPUTER SCIENCE
UNIVERSITY OF MASSACHUSETTS AT LOWELL
1998

Dissertation Supervisor: Georges G. Grinstein, Ph. D.
Professor, Department of Computer Science
ABSTRACT

Recent advances in concurrent architecture technology have led to a dramatic increase in the number of types of systems available as well as the number of systems in actual use. In contrast, software technology has not kept pace. Software tools have yet to be developed which fully compensate for the added difficulties and complexities inherent in programming concurrent architectures. Consequently, the tools used to aid in the development and debugging of applications for these types of architectures are often insufficient for the task. This makes it extremely difficult and time consuming to develop correct and efficient concurrent applications. The availability of high end graphics systems with most concurrent hardware provides an opportunity to extend development and debugging tools to use these graphics capabilities, where applicable.

In this research, we explored interactive computational steering, data analysis (as opposed to processor or control analysis), analysis of data operations, often using direct manipulation techniques, all in support of the development of concurrent applications.

We extended interactive computational steering beyond the ability to merely steer data values instantaneously. This resulted in techniques for the steering of operations, allowing the user to modify the application dynamically during execution. We also extended data steering to incorporate a persistent mode. In the persistent mode, once the user has specified the values for designated variable elements, the environment does not allow them to change.
Operation visualization techniques that are merged with the data display provide the user with a single focus and aid in the correlation of data, program status, and loop status information. The environment presents the operation visualizations in such a way that the user can also use them to debug the execution stack.

Finally, we show how users can apply the techniques developed during the course of this research to aid in the comprehension and debugging of concurrent programs. Examples show how the techniques can be used to find potential errors in two example concurrent applications or to verify that errors do not exist.
ACKNOWLEDGMENTS

I owe a great deal to my advisor Professor Georges G. Grinstein for his continued support and encouragement. He got me interested in visualization and in pursuing advanced degrees. He also helped my professional exposure in the field through publishing and conference involvement. Without his prodding I never would have gotten where I am today. I can only begin to describe the impact he has had on my life.

I would also like to thank my thesis committee Professor Haim Levkowitz and Dr. Hugh Masterman for their assistance in this long process. While he could not be on my thesis committee in the end, I would like to express my gratitude to Professor Stuart Smith for sharing his interest in parallel processing and aiding me when possible.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 BACKGROUND</td>
<td>2</td>
</tr>
<tr>
<td>1.2 RESEARCH GOALS</td>
<td>5</td>
</tr>
<tr>
<td>1.3 CONTRIBUTIONS</td>
<td>5</td>
</tr>
<tr>
<td>PROBLEM DESCRIPTION</td>
<td>7</td>
</tr>
<tr>
<td>1.1 PARALLEL AND DISTRIBUTED SYSTEMS</td>
<td>8</td>
</tr>
<tr>
<td>1.2 CONCURRENT PROGRAMMING</td>
<td>9</td>
</tr>
<tr>
<td>1.2.1 Application Comprehension</td>
<td>9</td>
</tr>
<tr>
<td>1.2.2 Application Debugging</td>
<td>10</td>
</tr>
<tr>
<td>1.2.3 Performance Tuning</td>
<td>11</td>
</tr>
<tr>
<td>1.2.4 Summary</td>
<td>12</td>
</tr>
<tr>
<td>1.3 SERIAL APPROACHES</td>
<td>12</td>
</tr>
<tr>
<td>1.3.1 Application Comprehension</td>
<td>13</td>
</tr>
<tr>
<td>1.3.2 Application Debugging</td>
<td>15</td>
</tr>
<tr>
<td>1.3.3 Performance Tuning</td>
<td>17</td>
</tr>
<tr>
<td>1.4 MONITORING VS. MODELING</td>
<td>17</td>
</tr>
<tr>
<td>VISUALIZATION AND HUMAN PERCEPTION</td>
<td>18</td>
</tr>
<tr>
<td>1.5.1 Visualization</td>
<td>19</td>
</tr>
<tr>
<td>1.5.2 Human Perception</td>
<td>20</td>
</tr>
<tr>
<td>1.5.2.1 Pre-attentive Vision</td>
<td>21</td>
</tr>
<tr>
<td>1.5.2.2 Color Perception</td>
<td>22</td>
</tr>
<tr>
<td>1.5.2.3 Texture Perception</td>
<td>24</td>
</tr>
<tr>
<td>1.5.2.4 Edge Detection</td>
<td>25</td>
</tr>
</tbody>
</table>
1.5.2.5 Applied Perception .................................................. 25
1.6 Interactive Computational Steering ....................................... 26
  1.6.1 Examples ................................................................. 27
  1.6.2 History Capabilities .................................................... 28
  1.6.3 Logging ................................................................. 28

3 Thesis Goals ................................................................. 30
  1.1 Primary Goals ............................................................ 30
  1.2 Problem Specifics .......................................................... 31
  1.3 Interactive Computational Steering ...................................... 32

1.4 Data Analysis Techniques .................................................. 32
  1.4.1 Foundation of Techniques ............................................. 33
  1.4.2 Inserting Data Elements at Selected Processors .................... 34
  1.4.3 Particle Tracing .......................................................... 35
  1.4.4 Selecting Sub-Regions of Processors .................................. 35
  1.4.5 Data Operations ....................................................... 36
  1.4.6 Environment Capabilities ............................................. 36

1.5 Analysis ................................................................. 37

4 Previous Work ............................................................... 39
  4.1 Program Instrumentation ................................................ 39
    4.1.1 Breakpoint-based Analysis ....................................... 40
    4.1.2 Trace-based Analysis ............................................... 41
      4.1.2.1 Real-Time Interaction ...................................... 42
      4.1.2.2 Hardware Instrumentation .................................. 43
      4.1.2.3 Software Instrumentation .................................. 45
        4.1.2.3.1 The probe effect ....................................... 45
# Table of Contents

4.1.2.3.2 Distributed Instrumentation ........................................... 46

4.1.2.4 Instrumentation Of Programs ............................................ 47

4.2 Application Comprehension .................................................. 48

4.2.1 Standard Approaches and Techniques .................................. 50
4.2.2 Formal Approaches ......................................................... 51

4.2.3 Application-Specific Approaches ........................................ 51
4.2.4 Three-Dimensional Applications ......................................... 52

4.3 Debugging Techniques ....................................................... 53

4.3.1 Basic Information Representation ....................................... 16
4.3.2 Statistical Displays ......................................................... 17

4.3.3 Code Views ................................................................. 56

4.3.4 Visual Display of Processor Status ..................................... 58
4.3.5 Data Analysis and Dataflow Analysis .................................. 58

4.4 Performance Tuning .......................................................... 60

4.4.1 Processor Utilization ....................................................... 61

4.4.1.1 Processor Utilization Profiling ....................................... 61
4.4.1.2 Processor Status ........................................................ 64

4.4.2 Interprocessor Communication ............................................ 65

4.4.3 Memory Access Patterns .................................................. 66

4.4.4 Three Dimensional Applications ........................................ 68
4.4.5 Simulation ................................................................. 69

4.5 Combined Visualization Techniques ....................................... 69

4.6 User-Defined Visualization Techniques ................................... 70

4.7 Scalability ................................................................. 71
## Limitations of Previous Work

### 5.1 Analysis Domains

### 5.2 Interaction

### 5.3 Views

### 5.4 Scalability

### 5.5 Display Techniques

### 5.6 Application Specific Techniques

### 5.7 3D Techniques

## The Viscon Environment

### 6.1 Environment Architecture

#### 6.1.1 The Visualization Subsystem

##### 6.1.1.1 Data Display Component

##### 6.1.1.2 Operation Display Component

#### 6.1.2 The Instrumentation Interpreter and Parser

#### 6.1.3 The Real-time Data Modification Subsystem

### 6.2 Environment Interface

#### 6.2.1 Main Control Window

#### 6.2.2 Display Windows

#### 6.2.3 Popup Menus

#### 6.2.4 Real-time Dynamic Modification Interface

### 6.3 Instrumentation Instruction Paradigm
### 7 Visualization and Steering Techniques

- **7.1 Visualization Techniques**
  - 7.1.1 Data Representation Techniques
  - 7.1.2 Data Movement Techniques
  - 7.1.3 Operation Representation Techniques
    - 7.1.3.1 Extensions to Basic Techniques
    - 7.1.3.2 Extensions for Detailing Control
- 7.1.4 Techniques for MIMD and Distributed Systems
- 7.1.5 Techniques for Long Running Programs

- **7.2 Steering Techniques**
  - 7.2.1 Program Level Steering
  - 7.2.2 Instrumentation Level Steering
  - 7.2.3 Real-time Data Modification
  - 7.2.4 Persistent Vs. Instant Interaction
    - 7.2.4.1 Instantaneous (Conventional) Interaction Techniques
    - 7.2.4.2 Persistent Interaction Techniques
  - 7.2.5 Dynamic Instruction Modification

### 8 The Analysis and Debugging Process

- **8.1 Example Algorithms**
  - 8.1.1 Even-Odd Exchange Sort
  - 8.1.2 Heat Dissipation
- **8.2 Application Comprehension**
8.2.1 Understanding Algorithm .......................... 128
Component Importance ........................................ 128
8.2.2 Understanding Data Movement .......................... 129
Characteristics ......................................................... 129
8.3 Application Debugging ........................................ 131

8.3.1 Verifying Border and Boundary Integrity .......................... 132
8.3.2 Verifying Algorithm Configuration .......................... 137
8.3.3 Verifying Data Values ........................................ 138
8.3.4 Debugging Loops and Conditionals ......................... 139
8.3.5 Debugging the Execution Stack .......................... 141
8.3.6 Verifying Algorithm Impact .......................... 141

8.4 Identifying and Correcting Errors .......................... 143

8.5 Methodology .................................................. 145

9 Contributions .................................................. 148

10 Conclusion ....................................................... 150

11 Future Work .................................................... 156

12 References ...................................................... 159

13 References Used but Not Cited .......................... 171

14 World Wide Web Sites ........................................ 173

Appendix A ......................................................... 174

Appendix B ........................................................ 190

Biographical Sketch of Author ................................ 203
1 INTRODUCTION

Parallel computers are approaching the point where they can be added to common workstations as coprocessors. Even limited systems can provide the performance needed for interactive computation; the computation executes on a supercomputer and the display is shown on a workstation. The performance of recent workstation and PC technology is sufficient to perform this display in near real-time [Erbac95]. In addition, new software platforms (e.g., MPI and PVM) allow users to simulate parallel architectures on workstation networks. With this increased availability of supercomputers and massively parallel systems, there is an increasing need for technology to assist users in understanding how algorithms operate on the hardware, where errors are occurring during the application execution, and what portions of the algorithm are degrading performance. A more important question, which derives from identification of errors, is why the errors occurred.

With the advent of symmetric multiprocessors (SMPs) in desktop machines, a new generation of concurrent systems is coming into reign. As these systems proliferate, more companies are beginning to provide multithreaded support in their applications to improve performance. While these systems currently provide support for only two to four processors, the time is fast approaching when many more processors will be supported and desired by users for the added performance.
Current programs that support multithreading do so only at a very high level. Take image processing applications as an example. These applications allow the user to work on multiple images simultaneously, allowing a complex task to be executed on one algorithm in the background while the user manipulates another image in the foreground. In addition, since the operating system processes also run on a separate processor, performance of the application and subsequently of the user are improved. On the other hand, this technique does not use the capabilities of an SMP system to its full extent. As the number of SMP systems and the number of processors in these systems grow, the limited support currently provided for these processors will be insufficient. Companies must begin to make more complete use of the processors available in a system. Taking our image processing example from above, what the user really desires when acquiring a multiprocessor system is for the complex task that was executed to complete in its entirety more quickly, ultimately in real-time. This requires subdivision of the problem among the available processors and more of a parallel processing approach to the application design than is currently taken.

As companies begin to design their applications for parallel environments, the number of individuals programming for parallel environments will increase dramatically. Initially, many of these programmers will be inexperienced. These inexperienced users, in particular, will need assistance in debugging and performance tuning their applications. While some applications, such as the image processing example we used earlier, are not difficult to parallelize, they are also one of the simplest cases. Most applications will be much more difficult to program as parallel applications.

1.1 Background

Extensive technology and tools have been developed to aid programmers in developing and debugging serial programs. Other tools aid in analyzing the execution of serial programs and measuring their performance. When this technology is available to programmers, they are
able to develop better software applications. This is the case even more so with concurrent systems due to the complexity and difficulty involved in parallelizing programs. The desire is to eventually automate much of the parallelization, debugging, and performance tuning process (through compiler technology). At this time, however, such capabilities are a long way off.

Caerts et al. [Caert91] discuss the limitations of the sequential debugging process applied to parallel architectures and the need for added functionality when debugging such architectures. The debugging cycle used for serial programs is based on the concept that there is a single thread of execution and that by following this thread of execution it is possible to iteratively narrow in on the problem area. With a SIMD (Single Instruction Multiple Data) architecture there can be thousands of instantiations of each variable, making it extremely difficult to determine the overall effect of each operation. With a MIMD (Multiple Instruction Multiple Date) or distributed architecture there is more than a single thread of control and the cyclic debugging technique fails. Interprocessor communication and large numbers of processors are factors in SIMD systems that increase this complexity by adding influences from outside a process’s own execution. The execution of a program on a MIMD or distributed system is also non-deterministic (each execution of a program can be different); though environments, such as that provided by Data Parallel C (DPC), can simulate SIMD style execution on MIMD architectures.

As a result of the complexity of parallel systems, an enormous amount of data associated with parallel systems (e.g., performance statistics, processor call graphs, variable instantiations, etc.) must be analyzed by users to understand an application and why it does not operate as expected. It takes a long time for programmers to perform this analysis. Thus, because of the large amount of data being generated, visualization is useful to complete this analysis efficiently [Khann91]. Visualization applies well to aiding this analysis because
visualization allows large amounts of data to be examined and analyzed more effectively and more timely by representing it in a form more easily comprehensible by the human perceptual system.

“Visualization is important for exploring the mass of detailed event data collected from a large system. Problems such as long queues and stalled transactions can be spotted quickly. Patterns that are obvious in a display can be hard, if not impossible, to discern in a log file. Unexpected patterns seen in a display can indicate patterns of system activity that no one might even think to look for. The broad view of many events at once is just as important as the detailed views.” [Jakie95]

In addition, the type of visualization used is important to the success of the tool. Clemencon et al. [Cleme96a] urge that the techniques used by tools to present data be easily and quickly comprehensible. The techniques should also match the programmer’s conception of the program and data as closely as possible. This will reduce the amount of interpretation needed and make the analysis process proceed much more quickly.

Moher and Wilson [Moher91] indicate that the debugging process is a necessity that results from the limitations of our reasoning, memory, and communication. This necessity can be relieved through the use of formal development methods. With the development of increasingly complex parallel architectures, the need for debugging is increased correspondingly. Thus, debugging technology needs to be improved to assist programmers and offset human limitations [Moher91]. Debuggers should assist the users in finding bugs by making them more visible [Moher91].

This thesis reviews previous work on the comprehension, debugging, and performance tuning of concurrent applications. We then describe new operation visualization techniques and interactive computational steering techniques. These techniques were conceived in the areas of dynamic data modification and dynamic instruction modification. They are geared towards improving the comprehension and debugging of concurrent applications. The benefits
of the new techniques and how they can be used to identify errors or verify the lack of errors in concurrent programs are demonstrated.

1.2 Research Goals

This research was driven by the need to improve debugging technology for concurrent systems. To this end, we explored the use of visualization techniques in several areas which have received little or no attention. In particular, we applied interactive computational steering to the visualization of concurrent systems, allowing developers to modify the visualizations in real-time. Interactive computational steering has never before been applied in this way.

A second novel area of this research is our concentration on analyzing data and in particular data operations. Most tools, on the other hand, provide an analysis of a concurrent application based on the process or processor. For analyzing data we investigated techniques for visualization of data values, regions of data values, data operations, data flow, etc. Lastly, we explored the modification of data values and instructions themselves as the program executes.

1.3 Contributions

Some of the techniques conceived during this research include:

- Visualization techniques for displaying data values
- Visualization techniques for displaying data operations
- Visualization techniques for displaying data flow
- Visualization techniques for displaying data values, data operations, and data flow in a merged display
- VCR-like controls to control execution rate of a concurrent application
• Direct manipulation techniques for selecting and highlighting multiple independent regions of data variables
• Interactive computational steering of concurrent systems
• Interactive computational steering techniques for changing data values during program execution
• Interactive computational steering techniques for changing actual instructions in a program during program execution

These techniques, taken collectively, provide needed tools for improving the development and debugging process. Many of the techniques have been implemented in a prototype environment to show their benefits and viability. The prototype is designed to effectively use either a SIMD system or a serial system, in the case where a SIMD system is not available. The environment should apply directly to MIMD and distributed systems as well.
2 PROBLEM DESCRIPTION

The algorithm needed to execute a problem on a concurrent system is generally much more complex than an identical algorithm for a single processor system. Consequently, programming concurrent systems tends to be extremely complex for most real applications. Tools are needed to assist in this task, but technology is currently very limited in this area. What is needed is a capability similar to that developed for serial hardware. Because concurrent systems are so much more complex than serial hardware, the tools must take this complexity into account and provide a corresponding increase in technology and capability to assist programmers in developing, debugging, and analyzing software on these platforms. Szelényi and Zecca [Szel79] list several goals of parallelization: to maximize parallelism, minimize overhead of parallel constructs, and balance load among processors. Only when we have a level of technology sufficient to aid in the programming of concurrent computers, allowing programmers to meet these goals, will programmers have the resources to develop efficient and correct applications in a timely manner.

Another area at issue is the number of processors that current programming and debugging tools can handle. Few tools and researchers “… have considered the problems associated with massively parallel architectures where hundreds or even thousands of processors might be involved …” [Thist93]. With the increasing number of systems available
that support this number of processors, the need for tools that provide support for these types of systems is becoming a necessity.

Tomas et al. [Tomas94] provide a background summary of parallel program visualization and characteristics of visualization tools (e.g., BALSA, Paragraph, TRACEVIEW, PIE, VIPS, etc.). Details of visualizing parallel Iterated Defect Correction methods (IDC [Tomas94]) and parallel integration methods are also provided. Kraemer et al. [Kraem92] present an overview of parallel visualization tools with a concentration on the tasks inherent in building parallel visualization systems. A classification of program visualization systems, including characteristics such as scope, abstraction, specification method, interface, and presentation can be found in [Roman93]. Background information related to parallel program visualization, a reference list for many available tools, and discussions of classification systems and of empirical studies is provided in [Hyrsk95].

### 1.1 Parallel and Distributed Systems

A SIMD architecture available for this research allowed us to develop useful techniques without the complexity engendered from the non-determinism inherent in MIMD and distributed systems. This improved the time required to develop and investigate new techniques. These techniques will apply directly to MIMD and distributed systems with minimal modification since non-determinism should not, by nature, impact them. Conversely, visualization techniques developed for MIMD and distributed architectures should also apply to SIMD architectures. It is thus important to discuss previous work in relation to all system types as some of the techniques are applicable to systems on which the techniques were not originally developed.

We have begun research into new techniques to aid programmers of parallel systems. These techniques are based on interactive computational steering and direct manipulation.
1.2 Concurrent Programming

Concurrent systems are much more difficult to understand and program than serial systems. Three areas in particular are affected by the complexity of these types of systems: application comprehension, application debugging, and performance tuning. These areas are all closely related and require many of the same capabilities. Application debugging and performance tuning are both dependent on application comprehension. This is true because it is only through understanding exactly how the program is currently executing that the programmer can improve the program.

An added difficulty of concurrent programming is that a parallel algorithm must be closely coupled with the underlying hardware to achieve acceptable speedup. Merely adding parallel constructs is inadequate [Panca95]. Adding parallel constructs adds to the difficulty of comprehending and correctly debugging parallel implementations.

1.2.1 Application Comprehension

The first area of importance is application comprehension. It is much more difficult for a programmer to understand how an algorithm works when it is executing on a concurrent system. There are three reasons for this. First, on MIMD and distributed architectures the programmer has to keep track of what is occurring during many threads of execution, instead of just a single thread. On a SIMD architecture there can be thousands of processors, each with different data which may need to be operated on in different ways. In effect, the user will need
to analyze when operations are performed on which processors and when the operations are not performed. This in effect generates a complex computation and interprocessor communication pattern. Second, the programmer needs to understand how information is communicated between processors. Third, there can be thousands of instantiations of each variable which need to be kept track of. A field of research has developed to aid user understanding of program execution. This field incorporates visualization paradigms and is termed *program visualization*. “The goal of *program visualization* ... is to help programmers form clear and correct mental images of a program’s structure and function.” [Brown85]

Program visualization has actually been broken down into different categories or levels at which a tool can be applied to aid the user in understanding a program or algorithm. Stasko [Stask92a] defines these categories as follows:

- Algorithm animation - focuses attention on an algorithm and its operations
- Program visualization - aids analysis of data structures, program states, and program code
- Software visualization - aids analysis of computer processes and data in addition to programs
- Computation visualization - provides analysis of both software and hardware views

Stasko uses *Computation Visualization* as a general and encompassing term. I prefer the term *behavioral analysis*, which is used more frequently in the literature and more completely describes the ideas being considered.

### 1.2.2 Application Debugging

A second area of importance, which is very closely related to application comprehension, is application debugging. Application debugging in fact relies on application comprehension but takes the analysis a step further. After the programmer has begun to
understand the algorithm as it executes on a concurrent architecture, additional capabilities
need to be provided to allow the programmer to identify situations where the algorithm does
not operate as specified (either logically or performance-wise) and to determine why. This can
be extremely difficult on MIMD and distributed systems where the operation or result of the
application can vary from execution to execution, due to non-determinism. Concurrent
programmers must also deal with other influences that make concurrent systems more
difficult to program than sequential systems. These include the need for the programmer to
define the interactions and communications between the processors and, in the case of MIMD
and distributed systems, the need to balance the computational load among processors. These
are activities that do not occur in sequential programs [Brown95].

1.2.3 Performance Tuning

The final area of concern is performance tuning. The primary purpose of using a
concurrent system is for the added performance these systems provide [Heath95a, Panca95a].
A poorly performing parallel application can be considered incorrect. Great care needs to be
taken when designing a concurrent algorithm to ensure it is finely tuned for the concurrent
system under consideration or performance may be degraded. As an example, consider a
distributed system. In this case, the interprocessor communication should be reduced as this is
usually the most costly operation in such a system. Performance tuning can become extremely
difficult with the complexity of many of today’s systems. This is perplexing since performance
tuning often needs to be done on a per system basis. Thus, there is a need for performance
tuning tools to aid in determining exactly why an application is performing poorly and to
help determine what modifications would best improve the applications performance.
“Visualization of concurrent program performance is critical for fast, correct debugging”
[Lehr89].
The most common cause of performance degradation is an algorithm design that requires a communication pattern not directly supported by the hardware. The actual communication pattern of an algorithm on real application data is often difficult to determine without the aid of tools. Performance may also be reduced because of too much communication on the whole or performing operations for which the system is not optimized. In some cases, the algorithm under investigation may need to be completely redesigned to more closely map to the concurrent system.

The number of visualization tools being made available for performance tuning concurrent systems is increasing dramatically. For this reason, Heath et al. [Heath95a, Heath95b] proposed an abstract high level model as a framework for the visualization of performance data. They describe guiding principles and foundational concepts. They also show that lessons learned from scientific visualization apply to performance visualization, particularly the visualization of multi-parametric data.

1.2.4 Summary

It is clear that users need new tools for developing applications for concurrent systems. Consequently, for this research we will provide application debugging capabilities through the use of interactive computational steering techniques and associated visual techniques to aid users in developing applications for concurrent systems. Since the three areas of research are all closely interrelated we must examine previous research applied to all three areas, as some techniques are applicable across all three areas of research.

1.3 Serial Approaches

Significant research efforts have been applied to the development of tools to aid in the debugging and analysis of serial programs. These tools allow programmers to examine most details of the execution of the program under consideration. While many of these tools are still
text oriented, with limited display capabilities, there has been a trend towards more visually oriented tools with interactive capabilities. The newer tools are important to this research because many of the lessons learned from them also apply to concurrent systems. In addition, some of the techniques can apply, with suitable modification, directly to concurrent systems.

1.3.1 Application Comprehension

A small class of tools is available to assist users in comprehending the execution of an algorithm. These tools generally perform behavioral analysis of one form or another to help convey to the user exactly how the program operates. While these tools are not debuggers, their ability to assist a programmer in understanding how the algorithm is actually working does assist in the debugging process. Such assistance can be considered a precondition to effective debugging and algorithm improvement. This portion of the debugging process is extremely important, in particular, for concurrent systems because they are so much more difficult to understand.

Because of their linear nature, serial programs are less difficult to understand than concurrent systems and thus extensive capability for the application comprehension portion of the debugging process is generally not used. This explains, in part, the small number of tools currently available. Programmers, rely on print statements and other debugging tools to aid in gaining an understanding of the program. They very rarely use tools designed specifically to aid in understanding the program. One area where these specific tools are used is in the teaching of programming. Two tools, Balsa and Zeus, are tailored to this area. Several other tools have been developed that are intended for the generation of 3D animations to aid in comprehension, including: Polka-3D, GASP, ANIM3D, and Pavane.

Balsa [Brown85] is an early algorithm animation system. Its goal was to provide a system that would allow users to more fully use the capabilities of graphical workstations as a
tool to aid in the understanding of program execution. As this was a new area of research during the development of Balsa, students using the system during course work required assistance from the instructor in understanding the animations. This situation has been improved in more recent research. The need for instructor aid in understanding the visualizations demonstrates the need for more powerful techniques that offer a more complete depiction of the program being animated, such that little if any annotation is needed with the animation system. Balsa used various visualization techniques, though data structure visualization is its primary focus. Considerable programming is required for each application to be animated. The need to program each application independently is primarily a result of the need to use different graphical techniques for each algorithm, even when the algorithms are closely related, and the need to calculate display parameters before the actual animation begins. Techniques have since been developed which may be used over a variety of applications. Such techniques generally rely on the use of icons or glyphs to represent data parameters being generated by the observed program. Environments such as Maritxu [Zabal92a, Zabal92b, Zabal93] use these types of techniques.

Zeus [Brown91a, Brown91b, Brown92] is a more recent system geared towards providing program animation. Program animation techniques have been shown to be useful in understanding and analyzing algorithms [Brown91b, Brown92]. Zeus allows the user to define events and visual effects to accompany each event. Zeus also provides state cues which change the representation of an object based on the current state of that object. As an example, a sequence of nodes representing the sorting of sub-regions of a file will at first display the nodes as round—representing their unsorted state. After the sub-region represented by a node is sorted, the representation of that node is changed to a rectangle. Brown et al. found that it is more effective to teach an algorithm by first using a small sample with extensive labeling and to move onto larger examples without labels after the initial concepts have been grasped. They also discuss the use of data designed to make an algorithm fail. Using preconceived data
to make an algorithm fail is difficult to do in a concurrent system because there is so much data involved that generating sets of data would require its own debugging environment. Zeus employs many visual techniques, each programmed specifically for a particular algorithm. These visual techniques consist mainly of graphs, trees, statistical displays, etc.

Algorithm animation greatly aids in the understanding of an algorithm by displaying the state of the algorithm continuously during the course of execution. By showing how the state is changing the user gains insight into how the algorithm achieves its result. Several animation systems have been developed to provide this capability. Polka-3D [Stask92a] is a simple animation system that incorporates the ability to animate 3D scenes. GASP [Tal94] was designed specifically for the animation of geometric algorithms—limiting its applicability. ANIM3D [Najor94] provides library primitives to aid in generating 3D animations of more general algorithms. Pavane [Cox91] provides several primitives within the environment that are designed for 3D animation. The use of 3D animations is still in the early stages and much work needs to be done. None of these systems provides support or investigates the use of 3D animations for concurrent systems.

1.3.2 Application Debugging

Application debugging consists of removing logical errors in a program or deviations from the program specification. To remove these types of errors, it is first necessary for the user to understand how the program is actually executing (see section 2.3.1). Debuggers generally provide additional tools to examine lower level details of an algorithm than is generally needed for pure application comprehension. Such tools include the ability to examine or set specific variables, examine the execution stack, examine data structures, etc.

Years of research into debugging technology have led to the “cyclic” approach to debugging [Utter91]. Using this technique the programmer executes the serial program being
debugged and stops the program at points in the program, examines the values of particular
data values, and successively narrows down the location and cause of an error.

Scores of tools are available to aid in the debugging of serial code. This is not the case
with concurrent systems. Because MIMD and distributed systems consist of many individual
execution streams running simultaneously, the standard breakpoint method of debugging
used in serial approaches may not apply directly to concurrent systems. In the case of SIMD
systems, there are so many instantiations of each variable that it is impractical to halt
execution at any one point and determine the status of the execution by examining variable
values, which is necessary for the cyclic debugging approach to succeed. However, this is
logically possible in SIMD systems.

While the debugging technology for serial programs is very robust, the technology
actually used in debuggers varies widely. The standard UNIX debuggers (e.g., dbx and gdb) are
fairly limited in their display and interaction techniques. They are mainly text-based with
little program animation capabilities, which severely limits their usability. Using the emacs
interface to the standard UNIX debuggers provides much better display capabilities and
interface mechanisms. In addition, it adds algorithm animation capabilities. Other debuggers
have been developed to provide much broader capabilities for analyzing and debugging
programs, their data structures, execution flow, etc..

VIPS [Isoda87, Shimo90, Shimo91] attempts to aid in the debugging of linked-list
structures. It can be very difficult to comprehend the nature of a linked-list, especially for
newer students of computer science. VIPS displays linked lists as syntax trees. It allows
sub-regions to be selected for closer analysis—allowing values stored in each node to be
examined. VIPS attempts to visualize list structures such that incorrect pointers will generate
a display that is easily identifiable by the user.
While VIPS, at first glance, may not appear to apply directly to the debugging of concurrent systems, there are applications to which it is appropriate. For example, the linked-list visualization technology could be applied to concurrent systems in the representation of network and/or processor interconnection topologies. This would work particularly well with concurrent systems that allow the user to specify the topology (e.g., a hypercube system is often visualized as a tree of nodes in which each node is a processor).

1.3.3 Performance Tuning

Performance tuning is less an issue with serial programs than it is with concurrent programs, but it is a concern that has warranted the development of tools specifically designed for its resolution. The primary tools currently available (e.g., prof and gprof) are non-visual and describe the execution time of functions as well as the number of times those functions are executed. More recent tools act as shells for prof and gprof and provide more visually oriented displays of the results.

Improving the performance of a concurrent algorithm is often dependent on the developer understanding what is actually happening to the data during the course of execution. The developer must be able to modify the algorithm to use capabilities of the hardware that are more efficient but modify the data into exactly the same form. Modifying algorithms to use different operations or capabilities while generating identical results is heavily based on comprehension.

1.4 Monitoring vs. Modeling

Monitoring a concurrent algorithm generally implies that an actual program is being examined and analyzed while it runs on actual concurrent hardware. This gives real-life measurements of what is occurring during the execution of the program. Modeling, on the
other hand, generally implies that an actual program run is not being used. Instead, methods of estimating the performance characteristics and execution behavior are used, including:

- Simulations of the program. Simulation detail can vary from merely simulating the program under analysis itself to the detailed simulation of hardware characteristics.
- Formulaic representation of the program’s execution
- Combinations of the previous two methods

Since we wish to apply our techniques primarily to the debugging and analysis of the actual programs to aid in their improvement, we will consider monitoring approaches and not consider modeling during this research. Pancake et al. [Panca95b] discuss the problems of monitoring and modeling performance and the limitations of each.

Tools are becoming available which model or estimate the performance of sections of parallel code, such as P^3T [Fahri95] and SimOS [Rosen95]. For certain programs these tools are beneficial in that they can identify when and where bottlenecks are occurring and can aid in determining data distribution strategies. Unfortunately, these tools aid only in performance tuning. They cannot aid in application comprehension or debugging. In addition, these tools cannot aid users in gaining a greater insight into how an algorithm is executing on concurrent hardware. This failure to provide insight into an algorithm prevents improvement of the algorithm when a different implementation is needed. While these tools can identify where performance bottlenecks are occurring, they cannot aid users in determining the best way to resolve them. In many cases the best way of resolving bottlenecks is dependent on understanding the nature of the performance problem and how the application works. Thus, there is a great dependence on application comprehension, which these tools do not provide.
1.5 Visualization and Human Perception

Debugging of serial systems is generally performed using the cyclic technique. With a concurrent system this debugging technique is not sufficient. This results from several factors:

- With concurrent systems there can be tens, hundreds, even thousands of instantiations of each variable. This makes it extremely difficult to examine the value of a variable as would be done in a serial system.
- Interprocessor communication adds an additional complexity to concurrent systems not present at all in serial systems. Identifying the sending processor, receiving processor, and transmitted data is important to understanding the effect of the communication.
- With many processors the user must keep track of multiple threads of execution. The effect of conditionals can make it extremely difficult to conceptualize what is occurring across the entire processor array.

1.5.1 Visualization

Text can be used to display information pertaining to the values of variables, the processor ID’s involved in a communication, and the context under which a SIMD processor array is operating. The context is determined by those processors which execute TRUE for a conditional. Processors in the current context execute a given operation. While MIMD architectures do not have a context in the sense that SIMD architectures do, the rest of the data needing analysis is similar.

Unfortunately, because of the amount of information that is displayed it can be difficult if not impossible for the programmer to interpret this quantity of information efficiently. This is because textual data is serial in nature and we are attempting to analyze concurrent data. “Textual presentations of data describing the execution of a parallel
computation system are inherently sequential and can be difficult to assimilate” [Kraem92]. For this reason researchers have begun examining the use of visualization as a method for representing the textual information. Visualization allows the human visual system to examine and interpret large quantities of information in parallel. This reduces the time required to analyze the data and also more closely matches the underlying concurrent architecture. The need for visual-based debugging in parallel systems is described by Stasko et al. [Appel93] as follows:

“Parallel programming has an advantage over serial programming in that many computations can be performed simultaneously, thus reducing total computation time. Visual-based debugging systems offer this same advantage over traditional text-based ones: a larger quantity and variety of information (as opposed to a single stream of text) can be presented concurrently, which can be exploited by the human visual perception system.”

Mukherjea et al. [Mukhe93] describe the importance of visualization to aid in comprehension in another way with an analogy to the adage “seeing is believing”, indicating that visual representations are more easily grasped and understood. It has been noted, however, that the adage “seeing is believing” implies credibility more than understanding and that the adage “a picture is worth a thousand words” better implies understanding. With the complexity of concurrent systems it is important to make the representation of the data as easy to understand and interpret as possible.

1.5.2 Human Perception

Successful visualizations require an understanding of the human perceptual system: just because a technique displays data in a graphical form does not mean the display will be useful. Displays need to be designed to harness the capabilities and overcome the limitations of the human perceptual system. Much of what humans perceive visually is done through pre-attentive vision, the processing performed by the eye before the brain performs active
processing [Gross94]. This includes perception of color, textures, and edges, items key in our iconographic displays.

The way we interpret a display is dependent of our conception of the data and the display, our background and previous experience, and our cultural experience. This ultimately effects our understanding and extraction of information from the data. While perception forms a foundation for everything we see, our background and cognitive interpretation of an image or of color also effects the meaning we yield from that image. Colors may have different meanings to individuals with differing backgrounds. Artists and photographers use color for mood or environmental effects while cartographers use color to represent particular land features [Earns95]. This will effect the way they interpret the image.

Tufte [Tufte97] discusses issues involved in providing more effective information. In his discussion, He notes that it is usually sufficient to provide only the smallest noticeable difference to make data recognizable. Providing greater changes tends to distract from the data more than it aids. Providing parallel images in time or space aids comparison, with images parallel in space the more effective. Thus, having two images side by side can aid the user in understanding their differences. This supports the value of direct comparison between two displays or images as well as the potential value of linked displays.

Another aspect of human perception is the ability for missing information or gaps to be filled in subconsciously. For example, when considering line drawings, most of the information tends to be concentrated around points of maximum curvature [Hende93]. Straight lines are not as important for determining the meaning of an image as the curves or bends. Consequently, large amounts of the straight lines can be left out and the human visual system will interpolate the information that is available and fill in the gaps. This applies to other types of displays as well but is not as clearly illustratable. This can be both a hindrance and a boon. Information or gaps can be filled in when they should not be, generating
structures that do not exist. On the other hand, it allows displays to provide additional information without loss of visual acuity, e.g., dashed lines will be just as visible as solid lines.

1.5.2.1 Pre-attentive Vision

We are able to discern information or details, particularly shapes or patterns, without actively focusing attention on the differentiating characteristics [Jules83]. This is termed pre-attentive vision and relates to our ability to perceive information even before cognitive processes begin interpreting an image. An example of the type of information that is discernible pre-attentively is line orientation [Picke88]. Details on the types of information that are identifiable pre-attentively are discussed in Beck [Beck74], Treisman and Gormican [Treis88], and Enns [Enns88].

Pre-attentive vision contrasts with the remainder of human vision which relies on active cognitive processes to identify structure. This can be envisioned as an active search or investigation of an image to identify characteristics and meaning whereas pre-attentive vision relies on the fact that the users focus will be drawn to characteristics without thought.

Gibson [Gibso61, Gibso66, Gibso79] argues that the human perception system is geared towards identifying the source of stimuli and their behavior. Pickett, Grinstein, and Levkowitz [Picke70, Picke88, Picke90, Levko90a, Levko90b, Levko91, Levko97] in their series of papers on iconographics argue that this fundamental nature of perception should drive the development of displays so that the pre-attentive vision process can be used to identify automatically the source of visual and auditory stimuli. The goal is to remove much of the slower analysis, which tends to be consciously steered [Enns88].

1.5.2.2 Color Perception
Color perception has received an enormous amount of attention in the literature. The correct use of color can be a great aid to the understanding of information. The inappropriate use of color, on the other hand, can be more distracting than useful [Earns95]. For example, varying hue (e.g., the heated-object color scale) has shown to be useful to identify particular data characteristics. However, it can be distracting or misleading in other cases as perceptually structures may be identified that have no corresponding structure within the data itself, see Rogowitz and Treinish [Rogow93]. This can lead to misinterpretation of the data.

Color perception is complex. Brightness perception, like most perceptual characteristics, is not a linear function of intensity. In fact, exponential distributions match the visual system better [Hende93]. This contrasts with most display technology, which implement linear scales for intensity.

Color perception is worse at the peripheries of vision [Earns95] which indicates that color can be used to focus user attention of particular regions of interest.

The characteristics of the rods and cones within the eye also impact the way color should be used. The cones that are sensitive to blue are the scarcest of the cones in the human eye causing humans to be relatively insensitive to changes in blue [Earns95]. These cones are also much more evenly distributed across the retina than the other cones. Consequently, blue is appropriate for large areas or for backgrounds but not for providing detail. Cones sensitive to red and green, on the other hand, exist in larger numbers and are more centrally located [Earns95].

It is not often obvious how to use various combinations of color as colors can appear to change depending on the size of the color patch and the color(s) over which the patch is placed [Earns95]. Earnshaw lists examples of good and bad color combinations [Earns95]. It is often
best to use a neutral background to prevent too many discrepancies in the way particular colors are perceived. Bad color combinations can be more fatiguing in addition to being deceptive. Colors at opposite ends of the wavelength spectrum force the user’s eye to refocus constantly [Earns95].

The number of colors that can actually be differentiated between varies slightly from user to user but numbers in the 1,000,000 color range for trained colorists [Tufte90]. Untrained individuals will be able to differentiate between far fewer colors. The human visual memory is much lower, about 20-30 colors. This implies that 20-30 is the maximum number of colors that should be used in legends and other encodings that require the individual to remember the meaning of specific colors, though for best results even fewer should be used [Tufte90]. Tufte also indicates that using color on gray or muted backgrounds is particularly effective for highlighting information and that the best choice of colors are those that appear in nature due to their familiarity and natural coherence.

This understanding of color perception is important to any work that incorporates the use of color. This thesis depends on color for many aspects of visualization, particularly to provide legend information and highlighting. Since so few colors should be used for legend information it is important also to use shapes to distinguish between characteristics. This is done in the operation visualization, in which the color identifies the class of operations being visualized. Additional information as to the exact operations being used is presented as the shape of the visualization. Since color perception is weaker at the peripheries it is important to have the operation visualization close to the data visualization. We incorporated the operation visualization directly into the data display window. This provides a correlated display that incorporates both techniques and locates the techniques within close proximity, helping to maintain the user’s focus.
1.5.2.3 Texture Perception

Texture perception is another pre-attentive capability of the human visual system. It refers to the repetitive patterns seen in natural scenes. An example of texture are the strands in a carpet that provide a texture based on the variation in size and direction of each of the strands. Brushing portions of the carpet changes the direction of some of the strands and changes the perceived texture.

The human visual system is capable of rapidly identifying small changes in texture. These changes can be used to discern information or changes in meaning over small areas [Gibso50, Gibso61, Gibso66]. Thus, texture is an important component to informational displays and the recognition of data or objects [Picke70]. The five limbed geometric icons developed by Pickett [Picke70, Picke88, Picke90] are specifically geared towards providing information through texture and harness this pre-attentive texture perception. In these icons, the statistical properties of the data are observed as qualities of the displayed texture.

1.5.2.4 Edge Detection

An issue closely related to texture perception is edge detection. Edge detection also occurs pre-attentively and represents the ability of the human perceptual system to identify boundaries between different textures, or slight changes in contrast, hue, intensity, and saturation. While even small changes can lead to boundary identification, the changes must be abrupt. Gradual changes will appear as a continuous gradient or as an image with no discriminable boundary. In addition, black and white images provide more boundary information than can be discerned with colors [Hende93]. The boundaries are sharper and more precise with black and white.
1.5.2.5 Applied Perception

Levkowitz et al. [Levko88, Levko90a, Levko90b, Levko91] developed a “color icon” that allows multiple parameters to be mapped to a single display. The color icon is a 4×4 or greater box of pixels in which application parameters or variables are mapped to the color attributes of each corner of the box. The remaining pixels are interpolated from the corner values. The user may select the color model to use for the icon (e.g., LHS, RGB, GLHS [Levko87, Levko93, Levko97], Munsel Book of Color [Levko92]). Since each icon displays up to nine parameters in a merged display, the color icon can represent a much greater amount of information than strictly single parameter displays. This creates a texture of color that can reveal structures not observable when viewing each of the parameters in isolation. The difficulty then lies in identifying mappings between data parameters and color parameters that will visually bring out any characteristics. Since there are so many mappable parameters in the visualization system, this can be a time consuming and tedious process.

Pickett developed a family of five limbed stick figures [Picke70, Picke88, Picke90]. With these icons, the data parameters are mapped to the angles between the limbs. Since the limbs can be connected in a variety of ways there are actually many possible icons. Pickett introduced a sample family of 12 icons. Parameters can also be mapped to the length and intensity of each limb. Many of the characteristics discussed in relation to the color icon also apply to the stick figure icon. Both these and other perceptually based techniques generate integrated displays that attempt to harness pre-attentive vision. Our work will also generate such displays.

When developing our visualization techniques we attempted to provide perceptually oriented displays. These are integrated displays that use color to highlight aspects of interest with lines and edges used to represent additional information. Lines and shapes are used in the operation visualization techniques with color used to represent broad categories of
operations. This helps reduce the number of colors the user must remember. We also use correlated displays to maintain the user’s focus on a smaller portion of the screen. This prevents the user from using peripheral vision to identify structures or color. The user may also specify the range of expected values. This allows the user to focus on that range of values. This provides better contrast by mapping a smaller number of values to the range of available colors.

1.6 Interactive Computational Steering

Most program visualization environments to date have been designed around a static execution philosophy. In other words, once the user has started executing a visualization scenario the parameters of the scenario and the parameters of the visualization system are fixed until the scenario has completed execution. At this point the user may change the parameters and re-execute the scenario in the hopes that the results are improved. This has been termed tracking [Mulde95a].

In terms of program visualization this means that if the user is watching the execution of a program and is attempting to examine one particular period of execution time, then the user must constantly restart the program with new parameters and wait for that period of time. This can be extremely time consuming and inefficient, particularly for long running programs.

With computational steering, the user can change parameters of both the visualization scenario and the executing program while the program is running [Liere96a]. This provides much quicker response and provides greater insight into the effects of the parameters on the execution of the program. The user will be able to see the effect of the parameters as an animation of the changing visual display. Computational steering as described by Atwood et al. [Atwoo95, Atwoo96] is
“... the ability to receive a continuous visualization of data as the program executes, coupled with the ability for the programmer to interactively modify any aspect—not just the visualization or input parameters—of a program at any time and immediately see the effects without restarting the computation.”

1.6.1 Examples

An example of how visualization parameters would be applied to computational steering is the following: visualization systems often allow the user to control the output range of the three color channels (red, green, blue). Computational steering would allow the user to change these output ranges dynamically as the program executes. Thus, the user could see the effect of reducing or increasing the amount of each channel in the visualization display. This capability would allow the user to very quickly identify optimum ranges for a particular simulation.

Changing simulation parameters is a bit more complicated. While, it is possible for any variable or data array to be modified, it doesn’t always make sense all the time. In particular, if the user was analyzing a sorting algorithm that was part of a larger application, it would be reasonable to modify the array before entering the sort. This could be used to test the sort for particular boundary conditions. It would not make sense to modify the array during the sort as this would generate an incorrectly sorted array. It would be reasonable to modify the array within a sort if it is a specific portion of the sort algorithm that is being tested.

Thus, a computational environment must provide the user with the ability to modify any parameter or array at any time. The user must decide if the changes being made will generate the desired results.
1.6.2 History Capabilities

Atwood et al. [Atwoo95, Atwoo96] described the ability for their environment to allow the user to go backwards and forwards through the execution history of a program. They call this time travel. In their environment all variables are defined as formulas. This makes keeping track of the execution history rather simple but is limited in capability. This environment allows the user to analyze in detail specific portions of a program and thoroughly understand why it operates as it does.

In a massively parallel environment we are generally dealing with an extremely large amount of data. This data would need to be stored in order for reverse execution to take place. This would require an enormous amount of storage and would also significantly slow down execution. This technique may be suitable if the user wishes to examine a very specific portion of code. This capability should not be enabled by default but should be provided as an option should the user need this capability after narrowing down the area of interest.

1.6.3 Logging

Mulder et al. [Mulde95a] described a logging facility by which the values of selected variables, parameters, and arrays are saved. This allows the state to be restored at a future time and also allows for a visualization of the entire execution over time to be generated. This is similar to the previous section except there is no intent on allowing the user to reverse the execution of the program. Thus, the situation is less complex but would still require an enormous amount of data to be stored in a massively parallel environment.
3 THESIS GOALS

The three primary areas of research for the improvement of concurrent algorithms and applications (application comprehension, application debugging, and performance tuning) are very dependent on the capabilities available to the user for aiding in understanding the algorithm or application. In addition, with the number of parameters available to the user and the length of execution runs of an application, it can take an enormous amount of time to analyze an algorithm or application and understand why it behaves as it does. Interactive computational steering provides capabilities for allowing the user to guide the computational process during the execution of the program. Interactive steering assists the user in directing the execution of a program to areas of interest or to where a problem may be occurring. The problem is that little or no research has been done concerning the visual techniques used in computational steering with respect to concurrent computation. What techniques make visually evident the patterns and structures of activities occurring on a concurrent platform? Do these techniques lend themselves to interactive steering? What criteria should be used to determine the applicability of the techniques?

1.1 Primary Goals

This research addressed the problems involved in developing software for concurrent platforms and visualization techniques that aid in developing such software. While a large
amount of research has been completed to improve the technology available to programmers developing applications for concurrent systems, there are still major areas where this technology is lacking. This research was directed at these areas in the hope that the problems associated with programming concurrent systems can be reduced.

Our emphasis was on investigating and developing interactive computational steering techniques to aid users in quickly identifying patterns within the execution of a program. These patterns will show the behavioral characteristics of the program as well as unexpected communication patterns, results, execution stacks, and use of the interconnection network. While these techniques allow the user to plan the direction of analysis, techniques are also needed to present the status of the execution. For this we developed new visual techniques to aid in user comprehension of concurrent systems.

1.2 Problem Specifics

Because of the novelty of interactive computational steering for the assistance of programming concurrent systems, there are a number of questions that must be resolved as to its basic applicability and usefulness in programming concurrent systems. This research was directed at analyzing some of these questions and resolving them. The questions of particular interest during this research were:

- What difficulties arise when providing interactive computational steering techniques in tools designed for aiding in the development of concurrent applications?
- Do the implemented visualization techniques lend themselves to interactive computational steering? What characteristics are needed in the visualization techniques to assure they do?
1.3 Interactive Computational Steering

Through interactive computational steering, the user is able to alter the execution of a program. Allowing the user to quickly direct the computation to the desired area of interest in this way requires considerable interaction capabilities. In addition, the display must be updated frequently enough for the user to comprehend the current status of the application and how changes in parameters are affecting the computation. In our case, how do changes in parameters affect the patterns or structures in the visual display? This allows the user to steer the computation towards areas of particular interest by changing parameters in a more composed manner.

Implementing an interactive computational steering environment for a concurrent system required that we implement a trace-based approach and provide real-time interaction, as discussed in section 3.1.2.1. The requirements for the display update rate implies near real-time display updates.

Real-time dynamic displays provide the user with a continuously updated display of the status of the program under investigation. By examining this dynamic display, it is possible to observe trends, directions of flow, and other characteristics of the algorithm. The user can also identify how parameters affect the display and find patterns and structures by directing the flow of the computation towards displays that exhibit interesting or inconsistent characteristics.

• What visualization techniques show structures and patterns in concurrent data and the operations being performed on this data?
1.4 Data Analysis Techniques

Our work differs from most previous work in the focus of our research efforts. First, our use of interactive computational steering in a concurrent environment has not previously been considered. Second, the focus of the visualization techniques to which we applied interactive computational steering differs from previous research. Most previous work centers on analyzing processes or processors. We concentrated on providing techniques for examining and analyzing data, independently of the processors and processes they may be associated with at any particular time. In this way, the data associated with a set of processors may be selected and traced as it is transferred to other processors or modified by computation. Thus, we can trace and examine the data values as well as where they are located.

Rather than just observing characteristics of the data, we allow the user to observe the operations being applied to the data and how the data is affected by these operations. Very few systems provide the ability to visually display the values of variables and follow their flow through a concurrent system. Even fewer systems provide the ability to change variable values (especially for large concurrent systems) to specific values or visual representations. The next extension to this work we provide is the visual display of operations being performed on data, providing insight into how data values are being generated, in a more visual form than can be provided by a purely text oriented display.

1.4.1 Foundation of Techniques

The techniques we explored are based on the concept that there is generally some visible evidence in the result data that an error exists within the program under investigation. This is especially true when the data can be displayed graphically, e.g., tracing NANs (Not A Number), zeroes, particular values, values within or outside of a given range, etc. Furthermore, propagation properties will generally transmit errors through a larger subset of
data and make the error more identifiable as a discontinuity the longer the application executes. Since such errors are identifiable through analysis of the resulting data, we have chosen to concentrate our analysis on the data within the processor array, rather than on the processors, processes, or application code that tend to be the primary focus of most tools.

An example of this effect can be seen in a demonstration we developed for the Terasys workstation which shows a matrix transposition algorithm executing on a SIMD architecture. The hardware initially provided for the demonstration contained a break in the parallel prefix network, which is used for communication. Any attempt to communicate data through this break resulted in a zero value. Performing the transposition required numerous shifts through the break. Each shift pushed all previously generated incorrect values further away from the source of the error while at the same time generating a new incorrect result. This single break in the communication network resulted in a large black line appearing in the transposed image encompassing nearly the entire height of the image. This method of identifying discontinuities which represent errors should be applicable to software errors as well as hardware errors. The characteristics of the structure being examined will usually help discern whether the error is hardware or software related.

1.4.2 Inserting Data Elements at Selected Processors

Often it is advantageous to use statistically controlled data in the examination of an application. Using such data allows specific variables to be tested for errors with values that should cause exceptional conditions. To this end, the user must be allowed to change data values in selected processors to specific values or to a specific representation. Changes to data values should be done both numerically and visually. Visually, the user should be able to select specific attributes of the visual representation. This will allow the user to observe and follow sets of data very easily. By allowing modification of variable values, we can execute the application with particular data values to identify patterns or structures that may not be
brought about with natural data. This will also make it simpler to track data value movement since the user will be able to change particular elements to be more visually identifiable than surrounding data.

1.4.3 Particle Tracing

Another useful technique is the ability to follow the movement of particular elements as they are transferred to other processors. This is similar to the tracing of particles in a field. We examined the ability to select a group of processors and follow the data elements currently contained within those processors.

An important concern that we examined was the ability to identify the actual data values as the elements are being transferred between processors and manipulated. Consequently, we looked at tracing selected elements not only by identifying them as solid colored sub-regions in grayscale fields but also by highlighting the selected elements with colored scales in grayscale fields.

In effect, with this highlight-based tracing technique we are using grayscale to identify the instantiated values of the data variables in the image for a particular variable while we are using a separate color scale (e.g., a red scale) to identify the values of selected elements. Thus, we can identify the values of the elements in the selected region as well as the non-selected region and quickly discern the difference between the two.

1.4.4 Selecting Sub-Regions of Processors

As discussed in section 4.2, most current environments are lacking in their ability to select processors that require further investigation. Selection capability must be improved upon by allowing the user to interact with the display and select the desired processors as the
program executes. By interacting directly with the display during execution in this way we hope the user will be able to accurately select the desired processors for further analysis.

1.4.5 Data Operations

An area that has received little attention is the visualization of the operations being performed on the data. Previous work has looked mainly at the operations that directly affect the utilization of processors (e.g., waits for locks). We extended this concept to general operations to aid in determining how data values are modified, not only where in time erroneous values can be identified—which is the extent of previous work.

While developing visual techniques for the analysis of data operations, we also examined the relation of the data operations to the data values resulting from the operations. Displaying this relationship will require displays that represent multiple pieces of information (e.g., an icon representing each process can be displayed using a two parameter icon with a circle to represent the operation and a line projecting from the circle to represent the data value).

1.4.6 Environment Capabilities

Providing the user with the ability to select subsets of processors requires that the user be able to temporarily pause and continue execution of the program. Because the techniques we incorporated into our environment required that certain capabilities be provided by the environment, it is noteworthy to mention some of the interface mechanisms that were added to meet the demands of the techniques we incorporated. While these capabilities aren’t all new they are important enough to the success of the environment to be noteworthy. Such capabilities include:

- Start, pause, resume, and halt execution. While this is a common metaphor, particularly in the analysis of log files, it is an important feature for this research.
• Select and highlight multiple sets of data elements independently
• Change the method used to highlight sets of data elements
• Change application variable values to specific values or ranges. Variables are those used by the application under investigation.
• Change the visual representation of sets of data elements
• Identify the parameters that are to be applied to characteristics of the selected visual technique.

All of the above capabilities take place in real-time (i.e., take effect immediately) to maintain interactive computational steering.

1.5 Analysis

It is important to understand that identifying errors through the use of iconographic displays (e.g., by noticing striations or other patterns in the visual displays) generally requires the application of domain knowledge. Without the application of domain knowledge it can be impossible for the implementor to know for certain whether the application is truly generating correct results. In our techniques, we are asking that the domain knowledge be applied to a resulting visual image rather than raw data. While this may require a learning curve for the domain expert it will ultimately reduce the amount of time required for the domain expert to analyze the results. This is particularly true given the amount of data being generated by today’s scientific programs. This fact applies to serial programs as well as to concurrent programs. The resulting data can be just as confusing to someone who is not an expert in the field no matter what type of system the program was written on. It takes a domain expert to decipher the data and determine whether or not the results are correct.

It is noteworthy to mention that nearly all the techniques discussed in this research will apply directly to serial platforms as well as to concurrent platforms. In fact, the
environment developed as a test bed for these techniques incorporates the ability to analyze serial programs in addition to the SIMD programs which were the primary focus of this research.
4 PREVIOUS WORK

While the technology for assisting programmers on serial platforms is well developed, the technology for concurrent platforms still needs much research. Methods need to be developed to provide concurrent platforms with many of the capabilities that were described for serial platforms. Because of the complexity of concurrent systems and the fact that there can be thousands of instances of a variable, techniques for analyzing concurrent systems tend to rely much more heavily on visual techniques.

4.1 Program Instrumentation

A useful environment for aiding the user in understanding, debugging, or performance tuning an algorithm must provide a method of acquiring details of the executing algorithms. Acquiring such details generally requires placing instructions into the algorithm code to pass the required information to the visualization environment. Adding such instructions is termed “instrumentation” of the program.

Generally, instrumentation instructions are inserted into the program before program compilation. Another technique for instrumenting a program, provided by most serial-based debuggers, is to have the user halt the program at specific points during the execution and then have the user enter commands to retrieve the desired information [Panca92, Utter91].
This is termed breakpoint-based analysis. The user is effectively entering the instrumentation commands dynamically during the execution of the program.

When instrumenting a program, it is important to have a clear idea of what operations to instrument. Zernik et al. [Zerni91] formally describe the concepts of work and time that can be used to determine instrumentation procedures. Work is any sequence of operations that the programmer identifies as an important increment in the progress of the program. Time is important because it orders events and because it specifies execution rate.

McDowell et al. [McDow89] describe the concepts of breakpoint-based debugging and event-based debugging, as well as their impact on the probe effect and subsequent effects. Event-based debugging requires that instrumentation instructions be inserted into the program code at appropriate locations where data describing the program execution needs to be gathered. The probe effect is the change in execution behavior caused by the instrumentation instructions. Breakpoint-based debugging completely halts the execution of the program and goes well beyond the effects generally referred to with the probe effect.

4.1.1 Breakpoint-based Analysis

Breakpoint-style debuggers are the most common type used in serial algorithms. With this technique, the user specifies exact positions in the program where execution is to be halted. Once halted, the values of various program variables may be examined and analyzed. The user may also specify watch points, which identify variables that are to be monitored during program execution, and have their values displayed whenever they are touched or pre-specified conditions are met.

Prism [Sista92] provides a breakpoint-based debugging environment derived from dbx, in which the results of an operation are initially displayed in their textual form. This text may be piped to a visualizer module that displays a visual representation of the results.
4.1.2 Trace-based Analysis

Trace points are locations of instructions or data that generate a message whenever they are accessed [Panca92]. Trace-based debugging is derived from the concept of watch points discussed previously [Panca92] except that the messages are generally not viewed directly. Instead, the messages are either saved to a file (termed a trace file) or translated into a visually oriented display.

Trace files came about as a mechanism for recording the large amount of data generated by recent concurrent architectures. Without the visualization techniques currently being developed, this data cannot be analyzed dynamically and requires large amounts of time to examine completely. By storing this information in a file, the data can be analyzed repeatedly off-line to find errors in the program execution. Unfortunately, these files generally become so large that they are rarely used—analyzing the files appears too daunting. Early visualization techniques were applied directly to these trace files and the need to reduce the time required to fully analyze them [Malon91a, Malon91b].

Thistlewaite and Johnson [Thist93] describe an environment that attempts to reduce the number of primitive events in a trace file by deducing compound events, representing higher level concepts, from the trace file. The environment also recognizes adjacent repetitions of the same compound event. This modified trace file may then be analyzed with any trace-based debugging tool. This environment greatly reduces the complexity of trace files but is limited in that it may remove details of importance. It is also very dependent on the capabilities of the trace-based debugging tool used to analyze the resulting trace file.

Another issue that is being dealt with by many researchers is that many tools are being developed either to generate trace files or to read them but that no consideration is being given to the lack of standardization of these trace files. The formats of the trace files
generated by the trace generating tools are all different and the analysis tools that read trace files read only one or two formats. This lack of standardized trace file formats greatly limits the portability of the tools. Glendinning et al. [Glend92] describe the use of trace filters to convert trace files from one format to another, thereby providing an environment where ParaGraph (an application analysis environment) may be used on a wide variety of systems by merely converting the resulting trace files into a format understandable by ParaGraph. Using such conversion tools is still limiting in that filters must be written for every trace file format.

Riek et al. [Riek92] go a step further than Glendinning et al. and define a specification for a monitoring system. By providing a standard, the authors hope to provide consistency among applications and architectures. The specification defines the events to be generated, the format of the events, and other relevant or necessary information.

Most of the systems described in this chapter use trace-based approaches with trace-files. Two principal mechanisms are used in generating trace files: hardware-based approaches and software-based approaches.

4.1.2.1 Real-Time Interaction

Two primary mechanisms exist by which program analysis may be performed when using a trace-based approach. Each has advantages and disadvantages. One mechanism is examination of trace files generated during a particular run of a concurrent program (e.g., the data generated from tracing is immediately saved to a file). The other mechanism is immediate analysis of the traced information as it is generated.

The use of trace files allows the developer to examine a particular run many times and makes it simple to implement VCR-like controls to move forward and backward through the application’s execution history. VCR controls permit very detailed exploration of a particular run. Unfortunately, using trace files makes it difficult to examine a large number of runs and
prevents easy examination of different parameters, and it may miss problems that occur sporadically. It also does not support human perception in the computational process.

Real-time examination, on the other hand, does not allow the same execution to be examined repetitively. Not being able to reexamine a specific iteration may make it difficult to repeatedly find a particular run with a problem. This is necessary due to the need to identify more precisely where a problem is occurring and why. It is also needed to verify that a problem has been resolved through changes to the program. Real-time examination does, however, provide the user with more control over the execution and allows a more interactive analysis to be performed. Application parameters can be adjusted to highlight a problem area much more quickly. As a program is run, the user can slowly change parameters until unexpected or interesting characteristics are seen. Dynamic updating of parameters cannot be done in the trace file approach, which requires a full application run for each parameter setup. In addition, unlike interactive real-time analysis, the trace file approach cannot be used to find problems with very specific parameter combinations in a timely manner. Re-executing a program with different parameter settings can be extremely time consuming.

While it may be possible to provide an environment that includes characteristics from both trace-based and real-time examinations, it would be difficult to do so without detriment to the performance of the environment. In addition, because parameters of the execution may be constantly changing, the storage requirements may be prohibitive, particularly if a large number of runs is performed.

4.1.2.2 Hardware Instrumentation

Hardware approaches incorporate modifications into the actual concurrent hardware to perform the collection of data values or to assist with the collection of data. Hardware techniques are very rarely used because of their cost and difficulty of implementation. These
techniques are also limited in the types and quantity of information they can gather. On the other hand, hardware techniques provide information not available with software techniques, such as the locality of memory references or the number of cache misses.

Systems that require the capabilities provided by hardware instrumentation will often use hybrid approaches, which provide many of the advantages of the hardware implementation but require some modification to the software application. The software will generally execute a small atomic instruction that causes the hardware monitor to operate. These types of systems are not nearly as expensive as full hardware implementations but are slightly more intrusive. The MSPARC performance monitor [Harde92], the HMON Environment [Dodd92], the Multikron hardware [Harde95], and the Paragon Performance Monitoring Environment [Ries93] are based on the hybrid concept. Problems that need to be examined carefully when using hardware assistance for tracing are:

- What kind of information will this provide the user?
- Is it adequate?
- Is it useful?
- Will there be a need for higher level information that cannot be provided by hardware-only approaches?

Harden et al. [Harde95] present a design that is tailored to making it easier to integrate software and performance monitoring hardware. This design is designed for promoting a standard in performance monitoring hardware and software. Because of the amount of data generated by today’s systems, it is recommended by Harden et al. that hardware assistance be provided to help reduce the overhead and obtrusiveness of gathering the needed data [Harde92, Horie93].
4.1.2.3 Software Instrumentation

Software instrumentation is the most common method of acquiring the needed data from an executing concurrent application. In this approach, the software application has procedure calls placed at appropriate locations that direct the application to transmit values of pre-specified variables to a given collection agent (usually one of the processors on a MIMD architecture or the host processor on a SIMD architecture).

4.1.2.3.1 The probe effect

Instrumenting an application in software has a side effect—often termed the probe effect or the Heisenberg Uncertainty Principle as applied to computing systems. When the program is modified to send messages to the collecting agent, we are changing the program’s execution behavior from that of an un-instrumented program. Thus, we are actually debugging a slightly different application, which can cause difficulties. The “perturbation” caused by instrumenting a program can change the order in which events occur [Lehr89, McDow89]. In addition, by forcing the program to execute more slowly (while it executes instrumentation code) we may remove contention for resources such as a global bus. Reduction of resource contention is a consequence of reducing the frequency of accesses to resources [Goldb91]. Different processors can also execute instrumentation code a different number of times, changing their relative speeds [Goldb91]. It has been shown that adding any amount of instrumentation can change the execution patterns of a program. Merely adding a print statement can cause a malfunctioning application to execute correctly [Harde92, McDow89]. For these reasons, it is important that if monitoring code must introduce timing errors that it not hide these timing errors [Dodd92].
MTOOL [Goldb91] attempts to reduce or eliminate problems encountered from perturbation by using minimum block counting techniques to reduce the time spent executing instrumentation instructions.

The Paragon Performance Monitoring Environment [Ries93] allows for the specification of time ranges and processors for which trace data is to be collected. Specifying that only pertinent information be recorded reduces the amount of data that is generated and reduces the perturbation. Reducing the amount of data being recorded does not, however, allow the user to examine other aspects of the environment that may influence the problem areas indirectly.

PICL, which is used by ParaGraph, stores the trace data in the local processor’s memory until the program has completed execution, at which time the data is saved to disk [Heath91]. This reduces the amount of perturbation by not constantly sending trace data across the interconnection network during program execution. This technique does not, however, remove the limitations of a trace based approach and does not allow real-time interaction.

4.1.2.3.2 Distributed instrumentation

Instrumenting a massively parallel machine has an even greater effect than instrumenting a serial or parallel machine with two to four processors. The most common methods used for instrumenting programs send the traced data to the host machine, where the information is either stored on disk for later retrieval or analyzed immediately for display. Sending all the traced data to a single machine (the host machine) leads to an enormous amount of information being sent to the host processor to generate each display. For a dynamic system, the host machine must then process the data, calculate the appropriate representation, and display the results. Since all nodes are attempting to send data to the
same host processor, the execution of the algorithm must be halted while all processors wait for their turn to transmit data to the host. An alternative is offered by Poinson et al. [Poins93] in which trace data is stored on the individual nodes of the concurrent machine, where possible. When a display is required, the appropriate view specification is transmitted to all nodes containing trace information. These nodes then generate only the portions of the display for the data they posses and transmit the results back to the host processor. Thus, only information needed for the display being generated is transferred and all processors share in the computation of the image, rather than the host processor being responsible for all activities. Because the view information is transferred to the individual processors, any information not needed by the current view is not sent to the host processor—reducing the amount of communication and computation by the host processor to a reasonable limit.

Node Prism [Sista94, Sista96] is another system that incorporates an environment that allows distributed instrumentation, though it is intended primarily for debugging. The goal of Node Prism is to provide a scalable system, both in the number of processors that can be represented on the screen and the time required to generate the graphical representation. The Paragon Performance Monitoring Environment [Ries93] and the Annai integrated tool environment [Cleme95, Cleme96a, Wylie96a, Wylie96b] also provide distributed monitoring capabilities.

4.1.2.4 Instrumentation Of Programs

Once the method of instrumenting the program is determined, the issue of which events and operations to instrument remains. The PEPP tool [Quick94] attempts to provide automatic instrumentation of programs. It is based on a model-driven approach. The user defines a model to be applied to the instrumentation algorithm and determines which events are to be instrumented. Different models may be used to define different levels of abstraction.
or for different debugging goals. To change the instrumentation, the user merely needs to call up another model and re-instrument the program.

Riek et al. [Riek92] define several areas where data should be collected and will likely prove useful: event semantics, control-flow, data-flow, communication, process-scheduling, monitoring, and related information.

The amount of time required to instrument a program can become excessive, especially for large programs. The programmer must make decisions about what events to instrument and where to instrument them, which can become extremely tedious. An approach offered by Topol and Stasko [Topol94] is to implement the instrumentation primitives within the distributed system primitives (e.g., each concurrent command will automatically execute the appropriate instrumentation commands as well as the operations to perform the desired concurrent operations). Having the instrumentation primitives tightly integrated with the system in this way reduces the overhead of the instrumentation instructions, which reduces the perturbation caused by the operations.

PEP [Szelé91] provides interface routines that may be inserted into the program code to dump the values of specific variables to the trace files.

### 4.2 Application Comprehension

The reason that algorithms designed for concurrent systems are hard to understand is because the user must keep track of many threads of execution, variable instantiations, and conditional processor execution. With all the research that has been done concerning the comprehension of serial algorithms, it is important to look at how much of this experience will apply to concurrent systems, if any. As mentioned previously, the primary method for aiding in comprehending concurrent programs is through forms of algorithm animation. Application
comprehension is often a precursor to program debugging and performance tuning. Algorithm animation techniques often need to take a more visual approach, resulting from limitations of textual representations (as discussed in section 2.5).

Part of the difficulty of comprehending concurrent programs arises from the difficulty of converting an algorithm specification to a concurrent program and from dissimilarities in their specification mechanisms. Very often, an algorithm specification will include graphs as well as pseudo-code. Attempting to implement these types of specifications in a concurrent program can be extremely difficult. The Poker environment [Snyde84] provides a mechanism by which the user can use graphs to describe aspects of the algorithm, such as interconnection patterns, communication patterns, etc. Using graphs to specify algorithm characteristics helps reduce the dissimilarities between the two mediums and the amount of debugging required after implementation.

Kimelman et al. [Kimel91, Kimel94] describe the importance of dynamic visualization for revealing trends and behavioral phenomena that may not be revealed at the end of execution in a static display. They also point out that many performance problems may be results of the complex interaction of various activities and not the individual activities or their cumulation. For this reason, the use of dynamic techniques is extremely important to future research in this area.

Smith et al. [Smith91] describe the importance of providing continuous, smooth updates of graphical displays to aid in the exploration of large sequences of data. With continuous updates, it is much more intuitive for users to identify trends and anomalies in a data set. Stasko and Kraemer [Stask92b] also discuss the importance of smooth animation over discrete animation. Smooth animation more clearly illustrates the program execution and is much more easily understood by the programmer.
Brown et al. [Brown91b] point out several concerns in performing smooth animations that apply directly to our work. In order to maintain smooth animation it is important to update the display frequently, at least once every tenth of a second. On the other hand, if the changes from one frame to another are too small, it will be difficult, if not impossible, for the user to identify a change. This situation can occur if color changes are beyond the ability of a human to discriminate.

4.2.1 Standard Approaches and Techniques

Maritxu [Zabala92a, Zabala92b, Zabal93] provides a system for examining and analyzing the run-time behavior and performance of concurrent applications. Maritxu focuses on analysis of processors, rather than processes. Maritxu relies on the ability of the human perceptual system to identify inconsistencies in a large animated display. Maritxu’s analysis is done with an “icon”, onto which parameters are mapped (parameters in this case would be data associated with the processor for which the icon stands). Icons and their layout may be defined by the user. These icons allow the use of color, algorithm animation, and visual overloading (encoding multiple parameters in each icon) to generate large composite images that the user can use to identify characteristic changes in the program execution through pattern recognition techniques.

RP3 [Kimel91] was designed as a visualization environment with a wide variety of tools to analyze instruction execution rate, page faults, locks, state transitions, etc. of all levels (e.g., hardware, OS, application, etc.) of the concurrent system being investigated.

PV [Kimel94] is a trace-oriented environment for program visualization that is suited to the analysis of complex programs. PV provides several visualization techniques for analyzing program execution, including scheduling views, activity views, kernel performance statistics, hardware performance statistics, active loop of executing function, memory
allocation, memory possession, and memory status. Where appropriate, unique colors are used to distinguish different activities, tasks, or processes. PV is unique in that it provides methods for analyzing applications at several different layers (e.g., hardware, OS, application). PV also allows for correlation between the different visualizations, which is rarely provided in other tools. Correlation refers to the ability to identify where the same process or variable appears in each of several visualizations.

4.2.2 Formal Approaches

Pavane [Cox91] takes a more formal approach to the description of the visualization process. In Pavane, a series of formal rules specifies the mappings between various state-spaces. The starting space generally consists of characteristics of the program considered important, while the final space is the display space—a representative animation of the program characteristics as modified by the formal rules. This is similar to the mappings of program characteristics to graphical objects used in most visualization environments, with the added advantage that it is formally defined and allows more than a single rule to define the final mapping. The rules allow the user to define different visual representations to be generated for different actions occurring within the computation.

IVE [Fried92] also describes a formal process for the design of various types of animations. Designing animations is done through assistive design. The user creates an initial rough representation of the visualization model, using a typical modeling application. This draft is then characterized by the user by describing system properties. The system then produces a set of designs that fit the characterization as generative object grammars. Lastly, the user is allowed to provide constraints on the grammars that are displayed and the user can select a grammar that appears to be promising.
4.2.3 Application-Specific Approaches

Stasko and Kraemer [Stask92b] argue for the use of application-specific visualization paradigms to aid in the debugging of concurrent programs. By application-specific, they mean that for each class of algorithms (e.g., sorting, graphing, etc.) a specialized visualization paradigm should be used. To aid in this task, the environment should make it simple to design and implement new visualization paradigms. Application-specific paradigms are necessary for the debugging of concurrent programs because of the need to reveal the inherent semantics and operation of the program, which cannot be fully examined from predefined library visualization methodologies. Stasko and Kraemer also discuss the importance of the visualization paradigm being capable of illustrating concurrent activities in such a way that the programmer understands the characteristics of the activities.

The environment developed as a test bed for these methodologies is POLKA. While POLKA does provide all the capabilities discussed, the authors admit it is hard to use. The difficulty with using POLKA primarily arises from the need to program the visualization paradigms in C++ using the POLKA environment.

Mukherjea et al. [Mukhe93] also describe the importance of application specific views and the need to simplify the animation design process to allow development of animations to be completed in a more timely manner. Providing for easier development of animations will allow application specific animations to be used during the debugging of applications, rather than to just aid in comprehension. If the process remains time consuming, users will not use this type of capability. Lens, the environment developed as a test bed for these techniques, provides a visual programming interface to an animation development environment that provides only the most often used attributes and capabilities (reducing the complexity and development time). The user also works with familiar programming concepts, rather than graphical primitives; the user therefore needs no familiarity with graphics libraries to develop
animations. Lack of user familiarity with graphics libraries is another reason for allowing the use of external visualization environments (e.g., AVS or APE).

### 4.2.4 Three-Dimensional Applications

Most of the environments described in this paper provide only 2D animations. Recently, 3D animations have been gaining interest, resulting from their ability to represent more information and more adequately represent certain types of algorithms (e.g., geometric algorithms) [Stask92a]. Stasko points out several types of features that may be mapped to the various axes to emphasize the need for additional dimensions (e.g., Value, Position, History, State, Aesthetics, etc.).

Hackstadt and Malony [Hacks95] discuss the rapid development of more advanced techniques for the visualization of concurrent systems through the use of external visualization systems (e.g., IBM’s Visualization Data Explorer). They extended the concept of a Kiviat diagram to three dimensions. Normally, a Kiviat will overwrite itself through time as it is updated. The 3D version (Kiviat tube) will expand into the Z dimension rather than overwrite itself. A version of the Kiviat tube was made semitransparent, with a slice capability, to allow one particular time segment to be examined in detail without being obscured by the remaining time segments. Using a visualization system allowed this technique to be implemented much more quickly than could have been done from scratch through graphics libraries. Consequently, using a visualization system allows various techniques to be examined and tuned much more quickly and effectively.

The Annai/PMA (Performance Monitor and Analyzer) environment [Cleme95, Cleme96a, Cleme96b, Wylie96a, Wylie96b] provides several techniques through the use of three-dimensional graphics. Annai uses three-dimensional bar charts and three-dimensional surfaces to represent data values and distributions of matrices.
4.3 Debugging Techniques

Debugging is usually performed hierarchically. The user starts with a general view of the program and slowly narrows down the location of an error [Utter91]. Any concurrent method must allow the user to work in a similar manner. That is, the debugging tool should allow the user to concentrate on specific areas of interest and view more detail when needed, but also provide a wide scale view that shows little detail but of a large number of processors. While providing multiple levels of detail has been relatively easy to accomplish in serial systems, it is very difficult in concurrent systems. For this reason, tools for concurrent systems generally provide more visually-oriented approaches. While many of the systems described in this section do provide visual displays, they do not provide the capabilities needed to fully debug concurrent systems. They are also lacking in that they do not provide visually-oriented interaction techniques that are needed to easily and fully use the capabilities of the given tool.

A detailed description of many of the characteristics of parallel debuggers, their capabilities and their shortcomings, can be found in [McDow89]. This survey also describes characteristics of displays used to represent information associated with parallel debuggers. Pancake [Panca92b] discusses the need and benefits of graphics and visualization for debugging parallel applications along with some of the current uses of these techniques.

4.3.1 Basic Information Representation

An important concern when designing a visual display is the ease of understanding the display and the ease with which the behavioral patterns it represents are discerned [Khann92]. Many techniques have been developed for representing information gathered during program execution. The techniques vary from simple lights, representing on/off
In the following subsections, some of the previous work performed in several of the major areas of analysis is described.

Kimelman et al. [Kimel91] go a step further by listing specific areas where analysis should be provided. These areas include:

- processor utilization
- page faults
- instruction execution rate
- memory accesses
- barrier synchronization
- context switches
- processor load
- processor status
- process status
- cache misses
- memory allocation
- parallelism
- use of locks
- procedure execution
- program states
- data operations
- interconnection network
- traffic

In the following subsections, some of the previous work performed in several of the major areas of analysis is described.
4.3.2 Statistical Displays

A simple technique for debugging concurrent systems is to provide visual displays that directly correspond to the statistical data generated during program execution (e.g., histograms or directly plotted graphs). These methods allow the programmer to interpret data much more quickly but are limited in the kinds and quantity of data they can display. In particular, while a histogram can show the amount of parallelism used in a program, it cannot show when this parallelism occurred. A graph can show when the program obtained a certain level of parallelism, but not why [Lehr89]. Many systems provide these basic techniques, including PIE [Lehr89], ParaGraph [Heath91], RP3 [Kimel91] and Annai/PMA [Cleme96a, Cleme96b, Wylie96a, Wylie96b].

Pavane [Horie93] displays the load of each processor, number of active processors, and the number of idling or communicating processors as graphs. Glenn and Pryor [Glenn91] describe an environment that uses a state diagram with most of its nodes capable of dynamic animation, allowing it to show characteristics of the program execution—number of streams in each state, work size, simulation time, and streams waiting at barriers. MTOOL [Goldb91] provides histograms of time spent doing various tasks—primarily applied to finding memory bottlenecks.

Traceview provides a more ample set of statistical displays, using Gantt charts, to display the event history of a trace file [Malon91a, Malon91b]. This system provides density bars with a value density or a point density function and provides average curves to deal with any data density problems. The color map used for the density bar is user selectable. Gantt charts in this environment may be used to display state transitions and/or the number of times a state is entered. The actual mapping of data parameters onto the Gantt chart is done by the user.
4.3.3 Code Views

Another simple form of visual analysis, used in many serial approaches, is the use of highlighting to show the current line of code being executed, as well as the variables currently being accessed [Brown85, Isoda87, Kimel94, Lehr89, Ries93, Shimo90, Sista92, Szelé91, Utter91]. Highlighting the code being executed greatly aids the user by providing a context under which the visualization is executing and identifying exactly where a problem lies in the code when a problem is observed in the visualization. While code highlighting could be applied in a limited form to SIMD architectures, it could not be applied at all to MIMD architectures. Code highlighting cannot be applied to MIMD architectures because each process in a MIMD or distributed environment can be running a different code segment. Thus, the environment would need to display the code executing on every process, which can be overwhelming. Although it is possible to display the code associated with a single process or processor, a truly effective method for displaying the code would also show the correspondence between the code and the processor on which it is running. Other tools provide more limited capabilities. Such systems usually display just the code of a procedure that appears to be of interest, resulting from a bottleneck of some form. MTOOL [Goldb91] falls into this category. PF-View [Utter91, Panca92] extends the concept of code viewing in two ways.

- By providing the textual view of the code in a hierarchical representation, showing only the main statement blocks and allowing the user to expand those statement blocks to look at progressively more detailed listings of the code.
- By providing a high-level visualization of the code. Generating this visualization is done by generating schematic diagrams, representing serial sections of code as rectangles with single arrows, parallel loops as ovals with multiple arrows, and parallel cases as polygons with multiple arrows. Color is used to represent the execution stage of the particular section of code. Execution status of individual processors in the individual sections of code is also provided, identifying locks.
Sigma [Guarn89] provides more extensive capabilities associated with code views. These capabilities are afforded by providing complete project management and database management capabilities. Through the code view, the user may select functions or variables and query nearly any information concerning those objects (e.g., “Where was this variable initialized or last modified?”, “Which routines modify or use this variable?”, and “Draw this function’s static call graph.”).

The Annai/PMA environment [Cleme95, Cleme96a, Wylie96a, Wylie96b] provides a slightly modified version of the code view browser. The Annai environment provides a structural view of the source code. The details may be hidden or seen by expanding or collapsing routines or loops. This provides the user with an idea of what is currently executing without completely overburdening the user with information. The user can gain the jist of what is executing very quickly.

4.3.4 Visual Display of Processor Status

Another method of providing information about the execution of an algorithm and its efficiency is to display the status of each process or processor during the execution of the application. This data could include information on whether the processor is executing, stopped, idle, waiting for a lock, or finished executing.

Faust [Guarn89] provides call graph displays (trees of function executions) which indicate the function each processor is currently executing, as well as the function hierarchy that led to that function’s execution. A combined view displays the call graph of all processors in a merged tree.

Node Prism [Sista94, Sista96] provides a type of display similar to Faust, called a “where tree,” in which each node has a number of children equal to the number of different functions executing on each processor. Node Prism provides more information than Faust
through the use of zooming. The highest level of detail, also provided in pop-up windows, provides information on the function name, line number, processors involved, etc. Sub-trees may be arbitrarily iconified and/or zoomed.

Many other systems support this type of analysis in one form or another. Pavane [Horie93] displays a grid of processors with the color of the grid square indicating the processor’s status. PEP [Szelé91] provides a dependency graph showing the dependencies and logical connections among processors (e.g., waiting for locks or ending tasks).

4.3.5 Data Analysis and Dataflow Analysis

The ability to examine data values and change those values during program execution to aid in understanding the algorithm execution and to test specific characteristics of the algorithm has also been widely used. The Poker environment [Snyde84] provides the ability to change data values in a concurrent environment during execution. Unfortunately, this capability has seen little use in more recent systems and has not been extended to more visually oriented environments. Changing data values during execution could prove to be a very valuable ability. By allowing the user to specify not only a variable’s value explicitly but also the color that the value should generate, the programmer will very easily be able to watch the flow of data through the processors and observe where and how the value is modified.

Prism [Sista92] and the C* Data Visualizer [Jourd90] provide displays of data values of selected variables. The displays consist of simple textual representations as well as graphical representations, assigning colors to the processor values. The color scale used to represent data values may be filtered to highlight specific ranges of values. Other methods of representing data provided by Prism include: thresholding, simple graphs of data rows, surfaces generated by mapping a parameter to the elevation of pixels in the color representation, and vector plots.
Any serial debugger that wishes to be given serious attention must include the ability to display the data values associated with variables and to specify the values of those variables. Examining data values during program execution allows the programmer to follow the execution of the program under different conditions to determine where potential errors can occur. Displaying data values is much more difficult in a concurrent system because there can be any number of instances of a particular variable. A SIMD architecture with 32K processors will have 32K instances of each variable. Poker is one environment that allows values to be peeked and poked into variables of concurrent systems. While the environment does provide tools to reduce the amount of typing required to assign values to all the processors, it is limited in the number of processors that can be displayed, a result of the amount of information displayed with each processor and the space thus required [Snyde84].

Seenet [Becke94] provides methods for analyzing network traffic that may be applicable to concurrent systems. The communication of data between nodes is represented by half links between the nodes, with the half link beginning at the originating node. The color and thickness of the half link redundantly represent the volume of traffic. This type of display is termed a link map. Becker et al. also presented a type of rectangular glyph in which the height of the glyph represents the volume of outgoing calls and the width represents the volume of incoming calls. This type of display is termed a node map.

Friedell et al. [Fried91, Fried92] developed the IVE environment to visualize the behavior of massively parallel programs. The environment provides several data analysis techniques. The techniques are primarily based on the use of color fields and elevation parameters to provide informative displays. They use these techniques to display processor values, processor partitions, and nearest neighbor communication.

The Annai/PDT (Parallel Debugging Tool) [Cleme95, Cleme96a, Cleme96b, Wylie96a, Wylie96b] provides several techniques for displaying data values. These techniques include
three-dimensional bar charts, three-dimensional surfaces, and tabular readouts of the data
(for detailed analysis of data values).

4.4 Performance Tuning

As mentioned previously, the performance of serial applications isn’t given nearly as
much attention as concurrent applications, though there are several tools to aid in the
performance tuning of serial algorithms. Concurrent applications, on the other hand, are
designed specifically for the added performance the concurrent system will provide. Because of
the complexity of concurrent systems, it is also much more difficult to get the desired
performance. Consequently, a wide variety of tools have been developed to aid in the
performance tuning of concurrent algorithms. As with the debugging of concurrent systems,
the tools being designed for performance tuning are visually oriented.

Several areas exist in which visual analysis may be applied to performance tuning. These include processor utilization, memory access patterns, and interprocessor communication. ParaGraph [Heath91] is noteworthy because it provides 25 displays, divided into categories of processor utilization, interprocessor communication, and task information. ParaGraph also allows the creation of user-defined visualizations for application-specific views. Hyperview [Nicho90] insulates the data presentation modules from other parts of the environment, particularly data filtering modules. This insulation allows any type of data to be displayed with any display technique. Hyperview also provides extensive display capabilities, including the use of dials, bar charts, LEDs, Kiviat diagrams, matrix views, xy plots, contour plots, and strip charts. Data is associated with the appropriate display technique through views.
4.4.1 Processor Utilization

Processor utilization applies visualization techniques to the analysis of processor usage and attempts to assist the programmer in determining locations in the program where inadequate parallelism occurs. This can be applied in three ways: global analysis, local analysis, and post-mortem analysis. In global analysis, the performance of the program at a particular time is represented by a single entity—no matter how many processors are in the system. Local analysis represents the performance of each processor at every point in time by a separate entity. Post-mortem analysis represents performance of the algorithm as a whole (from start to finish) as a single entity. Most of the techniques described in this section may be used for either local or global analysis.

4.4.1.1 Processor Utilization Profiling

Processor utilization is a simple aspect of performance tuning incorporated into most tools. This technique is adapted to analyzing how many of each processor’s CPU cycles are used during different phases of execution. Points during the execution of a program in which processors are not being used can indicate a problem in the algorithm design at those points.

SMILI [Khann92] is an analysis technique that uses Chernoff-like faces [Chern73] to represent characteristics of the performance of the algorithm at each point in time. The faces are used to display multi-parametric data by mapping different performance parameters to different aspects of the face; this mapping may be done randomly or with the intent of generating specific faces for desired characteristics. Observing changes in the face should identify approximate locations in the algorithm where performance is poor. This technique is very dependent on the characteristics being used to determine good/bad performance. It is also very limited in that it cannot aid the user in finding exactly where the problem is occurring or why. SMILI also suffers from the need for the user to learn the meaning of the faces every time
the mappings between parameters and face characteristics change. This can take time and lead to confusion. The most useful application of this technique is when attempting to identify values of control variables that affect the parallelism and granularity of an algorithm. In this way, a Chernoff face may be computed to represent the performance characteristics of the algorithm under different control values. The best control values may then be found by narrowing in on the best face.

One of the earlier systems developed for processor profiling is PIE [Lehr89]. PIE provides a histogram display in which the vertical axis consists of the individual processes being observed and the horizontal axis represents the time in milliseconds. Colored rectangles are used to represent the activity each process is performing. A unique color may be assigned to each thread to allow tracing of that thread. ParaGraph [Glend92, Heath91] provides many capabilities, including:

- Feynman displays that show the number of processors in the busy, overhead, and idle states
- Utilization displays that show the percentage of time each processor was in each stage for the entire program run
- Gantt charts that display the utilization of individual processors
- Task histories that show the tasks executed by processors in bar charts of different colors—tasks are user defined
- Utilization meters that show the percentage of processors in each of the three states
- Space-time diagrams that show processor utilization and communication patterns
- Kiviat diagrams that describe processor usage
- Animations that show communication patterns
- Communication pattern displays using a matrices of senders vs. receivers
• Communication volume displays using meters or histograms
• Trace records show the current event record
• message lengths
• concurrency profile
• simulation clock

Prism [Sista92] extends the concept of processor utilization displays by providing extensive profiler reports. The execution time of functions, with or without children, usage of interprocessor communication, and usage of other resources are provided. Unfortunately, the display mechanism provided is limited to simple histograms.

PEP [Szelé91] displays statistics on the number of parallel statements executed and the time spent executing those statements, which can be used to estimate overhead. Bar charts are provided to display the number of executions of a loop, the number of iterations of the loop (minimum, maximum, and average), number of processors used, and the number of chunks per processor. Overhead arising from inefficiencies (e.g., waiting for tasks, locks, events) is represented as charts using both time and percentage of time. Bar charts also provide insight into the execution of subroutines, including total execution time, time spent in loops, and overhead from waiting.

The Paragon Performance Monitoring Environment [Ries93] provides a modified version of ParaGraph to aid in performance tuning. It also provides modified versions of prof and gprof. All these tools are distributed. The environment can provide utilization information about all levels of the hardware, including: the processor, message passing processor, memory, and Bus and Network Interface Controller.
4.4.1.2 Processor Status

An extension of processor utilization profiling is the analysis of the processor status during various stages of execution. This technique provides more detailed information as to what each processor is doing at different stages during the execution of a program. It can identify bottlenecks and more accurately describe the state of each processor than is possible with processor utilization profiling.

The C* Data Visualizer [Jourd90] provides bitmap displays representing the current context of the system such that active processors are white and inactive processors are black. Context displays allow the user to gain an understanding of which processors are active at any time by merely glancing at the display.

Impact, a tool provided with Faust [Guarn89], displays the current event each processor is executing, as well as the duration that event was executing for. The display may be probed to gain the exact numerical time for which the event executed, as well as the event identification string. The display takes the form of time lines for each processor, with user-defined symbols placed on the timeline to represent the different executing events. The distance between symbols provides a representation of execution time.

The Pandore II environment [Barea94] displays lattices of the program execution. Each node represents an instant in the execution time and each edge from the node represents one thread of execution. This lattice clearly indicates where parallelism occurs and, what is more important, where it does not.

4.4.2 Interprocessor Communication

Interprocessor communication is any communication that occurs between multiple independent processors in a concurrent system. This includes transmitting application data
and result data between processors, transmitting flow control information, and transmitting error information. Displaying interprocessor communication is generally a more difficult problem to resolve than processor utilization. It can greatly affect the performance characteristics of the resulting program. Consequently, several tools have been designed either directly targeted for this application area or with some facilities for analyzing the interprocessor communication.

One method of representing interprocessor communication is by lines and arrows. Blocks are used to identify individual processors. Lines are drawn between two processors that are communicating with each other. Arrows are used to indicate the direction of the communication. The simplicity of this technique along with its well understood meaning makes it easy to use. It is particularly effective when used for subsets of processors. Radar [Panca92] is one tool that provides this type of display.

The Task Force Performance Monitor [Utech92] was designed specifically to analyze interprocessor communication on concurrent architectures. This tool is trace-based and provides only a single visual technique for examining interprocessor communication. The technique consists of a graph, similar to an electrical signal timing diagram, in which the level of “current” indicates whether the processor is reading or writing and the thickness of the graph indicates when the reading or writing takes place. These displays are similar to Feynman diagrams with additional information available. Feynman diagrams represent communication between processors as a graph with time on the horizontal axis and the processor ID on the vertical axis. Lines connect the sender and receiver of a communication. The Annai/PMA environment [Cleme96a, Cleme96b, Wylie96a, Wylie96b] provides a similar display to show interprocessor communication with additional information on memory usage.

Glenn and Pryor [Glenn91] describe an environment that provides a grid display of all processors on the system. Each grid entry displays the number of conflicts generated from
interprocessor communication. Only a single message may be sent on one port during any clock tick. If more than a single message is required to go to the same port, a conflict has occurred and consequently a bottleneck. A color mapping is used to represent the number of conflicts.

Jakiela [Jakie95] describes the visualization of queuing delays in distributed environments, which can greatly reduce the overall throughput of the system. Simple scatterplot displays with the transaction start time on the y-axis, transaction end time on the x-axis, and node ID represented by color shows when and where queuing delays are most severe. Jakiela also examined the use of timelines to show when transactions began and ended during the application’s life cycle.

4.4.3 Memory Access Patterns

Another technique used to analyze the execution behavior of a concurrent program is examining the access patterns of shared variables. This includes examination of which processors read or write the data values and how the data values change over the course of execution. Most of these techniques are applied to matrix algorithms. Techniques used to analyze execution behavior include displaying the color associated with a particular processor that accesses a shared data value and using a logarithmically scaled color map to represent the data value intensities [Horie93, Panca92]. The memory access patterns of a program are very important to its performance because they can identify bottlenecks where multiple processors are attempting to access the same variable simultaneously. Understanding how algorithms access memory, how multiprocessors sharing data affects access patterns, how the memory hierarchy impacts effectiveness, and ultimately, how the access patterns of the algorithm and multiprocessor architecture affect performance are necessary for ensuring an optimally performing algorithm has been developed.
The SHMAP tool [Donga90] is tailored for the analysis of memory access patterns. The tool initially generates a trace file representing the accesses made to the memory hierarchy. When the analysis portion of the tool is run, the user may define the system—allowing the user to analyze the memory access behavior of the algorithm on a number of different systems. Unfortunately, the system is limited to analyzing systems with 16 processing elements or fewer and with a single cache level. The visualization SHMAP provides for analyzing main memory consists of two displays, one for loads and one for stores. Similar displays are provided for cache memory. Each display contains a grid representing the main memory. Whenever a memory access is made, the appropriate grid location is highlighted with a unique color, representing the processor making the access. The color used to represent a memory access will fade to black over a short time to indicate the duration since the memory location was last accessed. Memory accesses occurring immediately after each other to the same memory location forces the system to flash the location, which alerts the user to the fact that a second access has occurred. Statistics are also provided on the number of access as a historical display. This provides feedback on the total number of accesses made to each memory location and the number of active processors.

Dongarra et al. [Donga92] describe the MAPA environment, which analyzes memory access patterns of both local and cache memory. In particular, it is applied to matrix accesses. A grid of processors is displayed with all values initially white. After an access, the memory position is made black. It will slowly lighten again until the next access. This allows analysis of locality of reference. Cache memory can be analyzed with a similar display, which colors read locations yellow and written locations red.

Glenn and Pryor [Glenn91] describe an environment that provides a grid display of all processors on the system. Each grid entry contains several letter “M”’s, one for each memory module. The color of each “M” represents its activity. A highly accessed module is colored red.
The modules may then be queried (probed), allowing the user to select from all variables in that module to determine what is being accessed frequently and why. This is useful for reducing memory bottlenecks and hot spots.

### 4.4.4 Three Dimensional Applications

Reed et al. [Reed95] have developed the AVATAR environment, which displays performance data in a 3D volume. The goal of their system is to aid users in understanding the performance effects of software changes. Understanding these performance effects is accomplished by generating a 3D volume, representative of the performance, which is updated dynamically as the user changes system and application parameters.

AVATAR also provides the capability for viewing this volume in immersive environments. The performance data is organized into a volume by generalizing the concept of a scatter plot. First, a scatter plot matrix is formed of all possible scatter plots. Each scatter plot consists of a different combination of parameters mapped to the X and Y axis. This scatter plot matrix is then generalized to a scatter plot cube in which each scatter plot is replaced by a 3D scatter plot, using a third parameter as the Z axis, and the matrix itself is extended to three dimensions. The user may then move through this volume to analyze the performance data. Color, labels, and transparency aid users in identifying their position within the cube and understanding the performance data. The resulting environment shows data clustering across several dimensions, which greatly reinforces the performance data upon the user.

Marc Brown et al. have extended the Zeus system to incorporate three dimensional interactive animations [Brown93a]. In this environment the user can animate 3D visual representation of serial algorithms. By interacting with the display the user can rotate the visualization to examine it from different angles. the user can also move and scale the scene. Brown et al. explored using 3D graphics in three different scenarios:
1. Representing inherently two dimensional structures.
2. Incorporating multiple representations of an object in a single display.
3. Providing a history function of two dimensional views.

Finally, a 3D sonification capability is provided which provides a redundant mapping of the data. Sonification can be used to identify changes in characteristics, to help in locating the user's current position within the environment, as a probe for areas of particular interest, or as a generalization of the performance characteristics of the algorithm as a whole.

### 4.4.5 Simulation

A final method of examining the performance characteristics of an algorithm is through simulation. Through simulations, the impact of network layouts and processor topologies can be examined before the changes are made. Simulation is particularly effective when a network of workstations is to be reconfigured and used as a distributed environment. The reconfiguration of the network is time-consuming and it would be useful to prove the effectiveness of the new layout. The Network Architecture Simulation System (NASS) provides this type of simulation environment [Nicho90].

### 4.5 Combined Visualization Techniques

Most of the tools reviewed in this thesis provide only one or two presentation or analysis techniques. Providing so few techniques is limiting because debugging concurrent algorithms often requires access to several techniques and it is difficult to determine beforehand which technique will be the most useful. Tools need to be developed which incorporate examples from many techniques [Jourd90, Panca92].

Environments are also being made available that consist of several tools, each of which is fairly specific and tailored to debugging one particular area of interest. The PM
parallel system profiler [Nicho90] falls into this category. It consists of the following three tools: Xtool for profiling program execution and providing information as to which routines are heavily CPU based, Ctool for providing communication statistics, and Etool for providing event-based characteristics.

The PARADE environment [Stask95] consists of several different tools which have been combined into a single environment to provide more capability. The tools used in this environment include Polka, Polka-3D, the animation coreographer, and several different instrumentation techniques (such as conch).

### 4.6 User-Defined Visualization Techniques

An issue of importance in the design of a visualization system is what visualization techniques to include in the environment. Because it is impossible to incorporate every imaginable technique, many systems include the ability for users to define and incorporate their own display methods into the environment. The ability for users to define their own visual techniques is important because it lets them design displays which more closely represent problems of importance in specific applications [Heath91, Heath95b]. The use of user-defined visualization techniques can aid in the comprehension and debugging of more complex and varied algorithms. For this reason, the ability to tie the debugging environment to an external visualization system (e.g., AVS and APE) is extremely important. Interoperability between debugging tools must also be examined. Systems that currently incorporate this ability are: VIPS [Isoda87], ParaGraph [Heath91], and Traceview [Malon91a].

Sistare et al. [Sista92] described the ability to export data to AVS as a possible future effort. Using AVS would provide many more visual techniques to the analysis of data for very little investment. With AVS’s many tools the user will be able to generate visualizations that are specifically designed for a particular application.
4.7 Scalability

The ability for a visualization environment to support concurrent architectures with an arbitrary number of processors, data variables, or interconnection network is very important. This problem area is generally termed “scalability”. With the increasing number of processors in today’s concurrent systems, it is important for visualization environments to provide techniques to handle both large and small numbers of processors. While representing small numbers of processors is not a problem, dealing with large numbers of processors brings up many concerns that must be dealt with.

Two problems, in particular, are encountered when dealing with larger numbers of processors. The first is the increased time required to gather the data from each processor for the generated display. The increased time results from the number of processors probed and the increased data generated. The second problem is the number of instantiations of each data variable to be analyzed, which becomes incomprehensible very quickly [Horie93].

Methods for dealing with the scalability problem and for allowing more processors to be analyzed include:

- the use of views (section 4.8)
- more appropriate use of interaction techniques
- extensive use of color and sound to more fully use the perceptual capabilities of the user
- distributing computation of the resulting display across multiple nodes

The C* Data Visualizer [Jourd90] provides scrollbars to allow more data to be analyzed than will fit on the display screen. The data may also be filtered such that only desired ranges of values are displayed.
Sarkar and Brown [Sarka94] describe a more effective method of displaying large numbers of processors through a fisheye lens filter. This filter enlarges the area of interest, making it more visually identifiable, while reducing outlying data. The farther away from the central area of interest, the smaller the visual representation of the data. A fisheye lens filter is more effective than displaying an area of interest in a separate window, because we do not remove the user from the context under which the application is operating.

A problem related to scalability is the use of virtual processors. A system that uses virtual processors must allow for visualizations at both the virtual processor level and the physical processor level [Szelé91].

4.8 Views

Many systems discussed in this thesis provide some sense of “views”. Views are different windows into the application being examined. Each window generally visualizes a different aspect of the application. Views are necessary because of the difficulty of displaying all appropriate information of a complex algorithm or even multiple aspects of a simple algorithm [Brown91b]. With the amount of data being generated by today’s concurrent systems, the difficulty of displaying the needed information is increased dramatically and it is impossible to provide all the required or desired information in a display that remains meaningful, useful, and uncluttered [Heath91]. Attempting to pick out important details from complex displays that attempt to display all relevant information becomes so difficult as to make the system unusable [Brown91b, Jourd90].

The exact specification of a view varies from environment to environment. LeBlanc et al. [Lebla90] describe a view as a method of emphasizing or ignoring selected information and describe what information is to be displayed. The visualization of the system can then describe
how to display the information. Systems such as Traceview [Malon91a, Malon91b] define a
view through the beginning time, ending time, and event filtering applied to a trace file.

LeBlanc et al. [Lebla90] discuss the importance of allowing arbitrary views to be
constructed. Currently, most systems provide only limited view capabilities and only a fixed
set of preconceived views. Unfortunately, views designed for one specific problem area will not
apply to other problem areas. This problem is compounded by the fact that most tools cannot
be used with each other, as they are language or architecture specific. Combine these facts
with the knowledge that no single view can adequately reveal all unexpected behavior and
there is obviously a need for general purpose view construction capabilities. Broad view
capabilities would allow users to create multiple views focusing on the problem at hand and
create new views as the need warrants.

Another concern discussed by LeBlanc et al. [Lebla90] is the availability of selective
refinement. As the user narrows down the problem area and wishes to focus in on fewer
processes, the environment under which the user is working should allow the user to define
appropriate views that will only incorporate the desired processes and if possible provide more
information than could be presented with large numbers of processes being displayed.

LeBlanc et al. [Lebla90] described a formal framework for the description of views and
a method for using these views. The framework is based on process interaction, the process
state, and time. Examples Leblanc et al. provide of views defined under this framework are as
follows:

- program view
- anonymous process view
- interprocess communications view
- data parallelism view
- processor view
- external view
- systems view
- logical time view
- physical time view
- synthetic view
- infrastructure view
The methodology consists of four phases of debugging/performance tuning of parallel programming:

1. Does the program work and how does it behave?
2. What causes the behavior seen in phase 1?
3. Determine general performance characteristics.
4. Tune the program’s performance.

The problem with this methodology is that it does not put enough emphasis on performance tuning. The performance of a parallel program is paramount. In addition, a poorly performing parallel algorithm may need to be completely redesigned from the ground up to improve its performance characteristics. Any visualization paradigm may be used within the guidelines defined by the framework and methodology.

Each window in POLKA [Stask92b, Topol94] is considered a different view and may provide different presentations of the application under investigation.

Zeus [Brown91b, Brown92] provides views in an algorithm animation system. Each view in this system is a different visual technique for examining the data (e.g., a different way of looking at the data).

VIPS provides an added capability for dealing with views. When multiple views of the same structure are active, selecting a node from any of the views will highlight the corresponding node in all appropriate views [Shimo90]. This highlighting scheme provides a mechanism by which the user can determine the relationship between several different views.

VIZIR [Hao96] provides a tool integration environment. Standard tools such as debuggers, ParaGraph, AVS, Matlab, etc. may be incorporated into the environment to aid
4.9 Interaction Capabilities

While the visual capabilities provided by the tools already mentioned are our primary concern, the methods provided for interacting with these visual displays are also of importance. Without sufficient and adequate interactive capabilities, the user will be unable to use the tools to their fullest extent. It is important to understand the capabilities each system provides to the programmer to control the environment and whether or not they are sufficient. What capabilities are lacking and why are they important?

Several interaction capabilities are provided by most systems, including a menu interface for basic operations, and the ability to control the execution speed of the algorithm. These are so common we won’t discuss them further. Pancake [Panca92b] discusses the importance and uses of Graphical User Interfaces and direct manipulation capabilities in these types of environments.

Becker et al. [Becke94] discuss the importance of dynamic interaction with program parameters to fine tune the analysis process. Gu et al. [Gu94a] discuss the importance of interactive program steering to researchers, especially of concurrent systems, and provide an extensive bibliography in this area. Glendinning et al. [Glend92] discuss the implementation of a real-time version of ParaGraph as a needed future enhancement. They state that there is no reason ParaGraph could not be used to visualize data in real-time as long as the display workstation is fast enough.

PIE [Lehr89] allows the programmer to select nodes in the graphical display (both of the program structure and the processor utilization chart) and will display the associated portion of the actual application code.
Horie and Ikesaka [Horie93] provide interaction modes in which debugger commands apply either to a single processor or to all processors. These interaction modes need to be extended to views such that views can be specified and debugger commands can be applied to specific views without affecting the other views.

Prism [Sista92] allows the user to select regions of the program code and apply operations to the selected code. The user may also point to any processor in a graphical representation to pop-up a window displaying the exact values of variables associated with that processor.

Heath et al. [Heath95b] discuss the importance of a synergistic feedback loop to the creation of successful tools. In this type of loop, as the tool produces images the user guides the tool through the selection of views, parameters, and options to efficiently identify and locate performance bottlenecks. A synergistic feedback loop is similar to what is often termed “interactive computational steering”, in which the user controls the direction and aim of the upcoming computation based on the results of previous computation. Unfortunately, as discussed by Pancake et al. [Panca95a], most tools do not provide this capability: “to date, however, performance evaluation tools have focused on predicting or measuring an application’s behavior with respect to one set of inputs or runtime settings at a time; it is up to the programmer to apply the tool to a variety of these settings and infer any general properties.”

Gu et al. [Gu94b, Eisen96] described the Falcon environment, which provides for the on-line monitoring and partial steering of parallel programs. The concept here is that many applications have a variety of options that affect the system utilization of a parallel system. By modifying these parameters as the application executes, better performance can be achieved. Only partial steering is provided since the programmer must identify what variables may be modified and where through the insertion of additional code.
Paradyn [Kunch96] allows the user to steer programs in order to improve their performance at runtime. When performance bottlenecks are identified the execution can be modified to improve the performance, potentially removing the bottleneck. This improves performance and helps the program to complete execution quicker.

Ungar et al. [Ungar97] discuss the importance of the interface and the concept of immediacy. Basically, they explain that the interface should be transparent, effortless, and not interfere with the user’s focus on the data. Immediacy extends to time, space, and semantics. Visual updates must occur immediately after an event in the program occurs that requires an update. Objects in the program that are closely related should be visually close. Object closely related semantically should also be visually close together; an instruction, the data used in the equation, and the result of the equation are all semantically related.

4.10 Perceptual Issues

As discussed by Brown and Hershberger [Brown92], “algorithm animation extends the exploration of a program’s behavior into the world of sensory perception.” Since most of the previous work on visualization techniques uses algorithm animation or dynamic animation in some form, we must be observant as to how color and sound are applied to best improve the perceptual representation of the animation. The problem here isn’t merely to generate an animation system but to provide an animation system that uses color or sound effectively to improve the visual representation. The environment also attempts to provide more information than could be realistically presented in a simpler system with its limited screen real-estate.

Since the purpose of these environments is to aid in the understanding of complex phenomena, the displays must be easy to understand [Heath91]. Color should be used to more effectively apply the perceptual capabilities of the human visual system. The goal is to provide
more information while making the displays easier to understand without overloading the user or the display capacity. Explanations of how to effectively use color and analyses of the human visual system can be found in [Hende93, Gross94, Levko97].

Color, as used in Zeus [Brown91b, Brown92], represents state information. Color requires fewer pixels than geometric shapes to represent information. Consequently, more objects can be displayed on the screen at one time. The authors also claim identification of global pattern changes should be more observable when color is used rather than monochrome. Color is used for highlighting, to link multiple views (using the same color coding in each view), and to represent time order (using a color spectrum).

The ParSee environment provides a very interesting use of color [Prest96]. In this environment the authors followed the philosophy that lighter is better and darker is worse. Through the use of a subtractive cyan-magenta-yellow color scheme they represent individual metrics as primary colors and multiple metrics with composites. In this way a portion of a program with multiple reasons for not achieving parallelism will be a composite of the appropriate primary colors, resulting in a darker hue. This makes problem areas quickly identifiable as well as indicating the reason for poor parallelism. Finally, saturation of the primary colors is used to differentiate an additional performance metric.

Zabala et al. [Zabal93] provide background on the human visual system and the importance of tailoring graphical displays that attempt to display large amounts of information to the use of pre-attentive vision. They also discuss the use of sound to present information and the benefits of providing redundant information by both sound and image. They designed the Maritxu environment to take advantage of this technology. Maritxu provides color to increase the number of visual dimensions presentable. In other words, Zabala et al. use a color scale of mostly gray values with color hues at the bottom and top, representing undesirable conditions.
4.11 Representations of Time

Executing a program animation system dynamically may lead to several problems. First, in an environment that displays a visualization of the program as it executes, the performance characteristics and execution behavior are very likely being modified by performing that visualization. Second, the animation rate may not be related to the application execution rate. This is because generating each frame of the animation generally requires the same amount of time while the amount of time between application frames varies.

Analysis of a large algorithm in progress will include portions of time during which a large amount of activity is occurring and periods during which no interesting activity is occurring. Using a proportional time scale maintains performance characteristics but leads to large periods of time during which the user must wait while nothing interesting is occurring. An alternative is to use a nonproportional time scale. This reduces the amount of time spent watching nothing happen but removes the proportionality of time between events [Panca92].

The most effective method of controlling the animation rate may be to allow the user to specify the animation speed and ensure the user is aware of the current rate. In this way, the user may speed up the animation to bypass periods of inactivity, but maintain some knowledge of how much execution time really passed between useful events.

When the display is being updated, should the processors continue executing? If yes, how should the data generated during the display time be handled? We don’t want to lose that information as it could be important. Zhang et al. [Zhang92] used the averages of recorded values occurring since the last display update. Averaging data loses information and generates approximate values that are inappropriate for some cases. If execution is halted, the execution characteristics of the application will be perturbed and any contention for resources will be
removed. Removing resource contention will make it difficult to determine where problems are occurring.

Kraemer and Stasko [Kraem94] discuss another relevant concern. When two events in a concurrent system have statistically identical time stamps, how should these two time stamps be ordered (e.g., by time stamp, adjusted time stamp, serially, or for maximum concurrency) and how should the timestamps’ animations be displayed? If the animations for the two events take significantly different amounts of time to complete, how should the animations be displayed and does the method chosen for the animation affect the method for animating events with identical timestamps? How should long periods of inactivity, interspersed with short periods of high activity be dealt with? How does the use of animation sequences of different lengths affect the logical timing of the program? These concerns must be examined when generating a program animation system for any reasonably complex concurrent system. Kraemer and Stasko introduced the Animation Choreographer, which controls the animation of a program animation environment. It allows the user to select options that affect the ordering of events and animation rate and explore the various alternatives, allowing the user to determine which options work best for a particular situation.

Kraemer and Stasko [Kraem98] go on to describe the characteristics a visualization environment should provide to accurately represent applications executing on concurrent systems. In particular, they find that visualizations for concurrent systems, in relation to time, should be reorderable, synchronizable, and independent. They go on to describe these terms and how they should be applied for different tasks the programmer may be performing.

ParaGraph [Heath91] is concerned with the correct order of events and not with the relationship between execution time and display time. Consequently, ParaGraph merely updates the display as quickly as possible, with little or no proportionality. Heath et al. found that close correspondence is not necessary for this type of visual analysis to succeed.
5 LIMITATIONS OF PREVIOUS WORK

The environments discussed in the previous section contain a variety of limitations or problems that reduce their usefulness and effectiveness for a variety of problem domains, architecture types, and user capabilities. Areas of limitations include lack of scalability, limited display techniques, insufficient user interaction capabilities, incomprehensible displays, excessively cluttered displays, and excessive computational requirements for displays. This section will discuss some of the limitations of the environments. In many cases, the limitations are general characteristics that affect many of the environments. These characteristics will often be discussed in relation to particular example applications. This section does not provide a complete discussion of the limitations of these environments.

Currently, most of the available tools are primarily designed for computer scientists and are difficult for domain scientific users to understand [Panca95a]. With current applications of advanced technology, such as virtual reality and immersive capability, this situation has been made worse. The lack of standardization of tools across multiple platforms and the steep learning curve of each different tool undermines much of the usefulness of the tools [Panca95a].

Kimelman et al. [Kimel91, Kimel94] describe limitations of most of the recent work in program visualization environments for concurrent systems. This work has mainly been
limited to communication or other aspects of parallelism with little examination of other aspects of system behavior (e.g., hardware level, system-level, application level activity, runtime library, etc.). PIE does provide some consideration for this activity but is mainly limited to visualizing context switching and does not allow multiple visualizations to be displayed simultaneously for correlation between displays. This chapter discusses limitations of some of the environments discussed in this thesis. We will attempt to provide tools that meet some of these limitations.

5.1 Analysis Domains

One of the more serious limitations of existing systems is the lack of support for a variety of problem domains. As mentioned previously, application comprehension, application debugging, and performance tuning are all closely tied together. Improving an application in one area generally requires knowledge of the other areas. An environment that does not provide sufficient information in other areas is limiting the applicability and usefulness of the environment.

ParaGraph [Heath91] has just such a limitation. It is limited to analyzing performance data. While analyzing performance data is often useful, it does not provide program comprehension capabilities, which will make it difficult for the user to determine specifics about any performance problems found during the execution of a program.

Another example is Maritxu [Zabal92a, Zabal92b, Zabal93]. While Maritxu provides some of the more advanced techniques for analyzing and debugging processors, it does not provide techniques for analyzing data independently of the processors. This can make it difficult to determine how the program is manipulating data and what form the resulting data is taking. Providing such capability would facilitate greater insight into the program and how the results are being formulated.
5.2 Interaction

We have previously mentioned the importance of user interaction capabilities for these types of environments. It is important to provide sufficient interaction such that the user can finely control the execution of the program and specify any necessary constraints or mappings for the display of associated data parameters or techniques. Unfortunately, most environments are lacking in their user interaction capabilities. Environments generally refine areas of specific interest to the research being examined and neglect the user interaction capabilities of the environment. This greatly reduces the usefulness of the environment.

One specific area where this limitation is seen is in the inability of the user to dynamically control the visualization of the program during execution. This is seen in environments such as Maritxu [Zabal92a, Zabal92b, Zabal93] and PV [Kimel94]. This limitation prevents the user from adapting the visualization to the execution and being able to dynamically narrow in on problem areas. This occurs when a visualization environment provides only post-mortem analysis. As a consequence, the user requires much longer time periods to identify specific parameter combinations that cause problematic executions.

Paradyn [Mille95] attempts to automatically identify performance bottlenecks in a running program. In order to accomplish this goal Paradyn needs to be rerun for different parameter settings. For a long running program this can be extremely time consuming. Paradyn is also limited in that it cannot identify every type of performance problem and can only identify the slowest portion of the current algorithm. It will not identify situations where modification to or re-implementation of the algorithm would offer greater performance. Interactive steering of the program while it is executing would provide a much more efficient approach. In this way, if the analysis appeared to be focusing on an area of little interest the user could adjust parameters accordingly and direct the analysis to a more appropriate area.
While Node Prism [Sista94] allows the user to employ relational expressions to select any subset of processors, it does not allow the user to select processors visually. Most recent tools being developed to aid in programming concurrent processors use graphical displays to visually represent the execution of a concurrent program. It is visually that the user will identify characteristics in subsets of processors that they will want to investigate further. Identifying processor ID’s from a visual representation would require the probing of the display, the determination of an appropriate relational expression, and finally the entering and evaluation of the relational expression. While this approach is useful for much of the debugging phase, there is also a need for methods of selecting subsets of processors interactively with the display. Node Prism displays all current subsets in a window, with selected nodes highlighted, which may be zoomed and panned. Node Prism does not provide a mechanism by which data values may be changed.

5.3 Views

The applications being developed for recent concurrent architectures are extremely complex, consisting of numerous parameters, large numbers of variables, and complex communication patterns. These combined with the complexity of the concurrent hardware itself, consisting of hundreds or even thousands of processors, result in an exceptionally complex environment. Attempting to display all the information associated with an application execution simultaneously would be futile. The amount of information would make any display incomprehensible. Some environments instead attempt to reduce this information by allowing users to specify the parameters that are actually to be examined. Very few allow the specification of the processors that are to be visualized. Even fewer allow both of these specifications concurrently. An example of an environment suffering from limited use of views is ParaGraph [Heath91]. ParaGraph merely looks at a view as another display technique.
applied to the trace data and doesn’t explore some of the capabilities that the use of views could provide.

5.4 Scalability

A large class of techniques exists which, while presenting useful information for a number of processors, does not scale well to larger numbers of processors because of the amount of screen real-estate required to represent information. These techniques are effectively limited in the number of processors for which the tool is realistically usable. The method discussed in section 3.4.2 for representing interprocessor communication with lines and arrows [Panca92] cannot clearly represent large numbers of processors because of limitations of screen real estate and the size of icons. Other techniques that generate cluttered and incomprehensible displays when used with large numbers of processors include Kiviat diagrams and Gantt charts [Horie93, Khann92].

Other techniques attempt to associate a processor ID with a color value. With large numbers of processors it becomes difficult to remember which processor is associated with which color. It also becomes difficult to differentiate between close shades of a particular color. Consequently, while this technique is effective for smaller numbers of processors, it begins to fail for larger numbers of processors.

Faust [Guarn89], as discussed in section 3.3.4, displays call graphs to present the execution history of a program. The representation of call graphs inherently requires a large amount of screen real-estate. This becomes particularly problematic when the call graph becomes large and deeply nested. With multiple processors all executing different call graphs, the problem is compounded even more.

The environment described by Glenn and Pryor [Glenn91], which is used to identify memory bottlenecks, represents memory modules in a concurrent hardware with letter “M”’s.
This provides very little added benefit, wastes valuable screen real-estate, and limits the number of processors that can be realistically displayed. The attempt at using a meaningful representation may aid the user in identifying context but greatly reduces scalability. A different method should be used to identify context.

5.5 Display Techniques

Many environments attempt to provide the user with a variety of display techniques to apply to the data being gathered from the concurrent system. These displays can vary from simple black and white meters to complex iconographic displays with considerable use of color and sound. The problem is that most systems, while providing more than a single display technique, do not provide sufficient capability to truly aid the user in a variety of tasks. ParaGraph [Heath91] provides many displays but they only incorporate simple techniques (e.g., black and white histograms and charts) and could be greatly improved with more advanced techniques (e.g., using color and sound).

While providing advanced visual techniques (as seen in Maritxu [Zabal92a, Zabal92b, Zabal93]) or large numbers of visual techniques (as seen in ParaGraph [Heath91]) is not strictly necessary, it is necessary to provide sufficient capability to make an environment useful. RP3 [Kimel91] is limited to simple lights, meters, bar charts, and graphs. PV [Kimel94] provides only simple histograms with a variety of mappings. The C* Data Visualizer provides only textual and nine gray level representations of data values. The limited display capabilities provided with these systems limits their applicability and the complexity of the applications that they can be used to represent. While Maritxu [Zabal92a, Zabal92b, Zabal93] and Zeus [Brown91a, Brown91b, Brown92] use color to improve their visual presentation, the use of this color is rather simple and limited. More extensive use of color could be used, as described by Levkowitz [Levk088, Levko90, Levko90b, Levko91, Levko97].
5.6 Application Specific Techniques

While application-specific views are important, as discussed in section 3.6, it is also a necessity that generating and describing these user defined techniques be simple and efficient. Environments that require extensive programming or modification to incorporate such capabilities will prevent the technique from being useful. Environments such as RP3 [Kimel91], ParaGraph [Heath91], Polka-3D [Stask92a], and Anim3D [Najor94] have this failing. RP3 also requires extensive familiarity with the internal workings of the software, adding to the complexity. Lastly, environments that require extensive programming for specialized views will effectively require that the visualization technique be debugged at the same time that the application code is being debugged. While this is not as important for educational uses, it becomes significant for development uses, in which the capabilities of the environment will be used to their fullest extent.

5.7 3D Techniques

The 3D visualization techniques discussed in section 3.2.4 provide more visually pleasing techniques than are seen with many environments. This approach comes at a cost. Generating the 3D displays for these types of techniques is much more computationally intensive than standard 2D techniques. The cost is compounded by the need to provide interactive techniques (e.g., capability needs to be provided to allow rotation of the 3D space). 3D interaction allows the user to view the space from all angles. Observing the display from different angles may provide more comprehensible results or reveal hidden features. A need also exists to be able to examine the 3D space from various angles while the space is being updated as a computation progresses. This can become confusing because the user is attempting to view the 3D space as it is changing. Consequently, there is a much greater burden on both the hardware and software environments when supporting this type of technique.
The approach presented by Reed et al. [Reed95], as discussed in section 3.4.4, can suffer from an extremely cluttered appearance that can become very confusing to the user. Also, the user is required to explore the data to gain an understanding of many of its details. This can become time consuming, which may preclude the technique from being used. In addition, this type of environment requires that the user be able to maneuver through the volume smoothly or the user will become lost and confused easily. The interactive requirements of the environment put a large demand on the computational power of the underlying hardware. Consequently, users will require fairly high end graphical workstations with stereo capability to fully use the techniques proposed by Reed et al. Currently, the needed hardware is too expensive for a debugging environment. Organizations will be unwilling to provide such expensive resources to debug applications.
6 THE VISCON ENVIRONMENT

This chapter provides an overview of the environment, termed VisCon (Visualization of Concurrency), developed to test the techniques investigated during the course of this research. The VisCon environment and the techniques it incorporates aid the user in understanding, testing, and debugging concurrent applications. The ability to both test and debug a program from within an integrated environment is important for the correct and efficient analysis of the program [Loure97]. This chapter describes the architecture and interfaces inherent in the environment.

6.1 Environment Architecture

Figure Env-1 depicts the architecture for the VisCon environment. The user must modify the application to incorporate instrumentation instructions to identify the activity being performed by the application at any point in time. The implementation of full instrumentation instructions best serves the capabilities provided by the VisCon environment. This is in contrast to merely identifying interesting events as is used in many program animation systems. The capabilities integrated into this environment really require per instruction instrumentation to achieve their maximum benefit. The instrumentation instructions describe the instruction just executed and pass pointers to the variables or data
the operation used. This not only allows the visualization system to access the data but also allows modification of the data.

The visualization environment itself has three principal components. These components include the visualization subsystem, the instrumentation interpreter and parser, and the real-time data modification subsystem.

6.1.1 The Visualization Subsystem

The instrumentation instruction parser provides the visualization subsystem with pointers to the updated data and display window. Parsing and interpretation of instrumentation instructions generally results in the calling of this subsystem to represent events visually. The environment interprets the data parameter settings and other given information to determine the type of information to display and the type of visual display the environment should provide. The principal component of the visualization is the display of data; primarily, concurrent data being operated on by the concurrent system. Additionally, the architecture incorporates visualization of the operation currently executing as well as the execution stack. The data and operation visualizations are each in separate components of the visualization subsystem.

6.1.1.1 Data Display Component

The data display component performs the actual display of the individual elements of a particular visualization. Routines provide the ability to redisplay the entire data set or select subsets of the data set—a performance consideration. Additionally, this component will determine the appropriate highlighting (tagging) of a data element and will call appropriate subroutines to display the data element appropriately. A later section discusses tagging in detail.
The instrumentation instruction parser usually calls this component as a consequence of its analysis and usually indirectly through the encapsulating visualization subsystem. It is callable directly, however, when the user is interactively modifying data values. In this case, the data modification interface will identify the subset of data values to update; this provides up to date displays without significantly impacting performance.

### 6.1.1.2 Operation Display Component

The operation display component is responsible for keeping track of the execution stack and displaying this stack as well as the currently executing operation. This predominantly involves a mechanism for interpreting what instruction is currently executing and how to display this operation. The execution stack keeps track of such items as function calls, iterative loops, and conditional constructs.

### 6.1.2 The Instrumentation Interpreter and Parser

The instrumentation interpreter and parser actually consists of four components. Instrumentation instructions call the instrumentation parser that then parses the instrumentation instruction to determine what has occurred and what corresponding action the environment must take. This process is the aggregate of several components, including, the parsing of the actual instrumentation instruction, the parsing of user identified modifications to the instrumentation instruction, the parsing of user directed data modifications, and the parsing of user identified modifications to display updates.

The instrumentation instruction parser is the foundation of this subsystem. It takes the basic instrumentation instructions, determines what they represent, and decides what actions to take. The parameters provided by instrumentation instructions include information needed to accomplish these tasks. The environment provides the user, however, with the capability to modify many of the default responses to specific instrumentation instructions.
Consequently, the user’s interaction with the environment interface directly impacts the remaining three components of this subsystem.

The instrumentation modification parser compares the current instrumentation instruction with a list of modifications the user has instantiated to determine if the user has indicated that the current instruction should be altered. If this is the case, then the result of an alternate instruction or set of instructions, that the user provides, replaces the results of the original instruction by passing the results back through the instrumentation instruction.

The data modification parser, in addition to interacting with the environment interface, also interacts with the data modification interface. It is the data modification interface that actually provides this component with knowledge as to what data the user has chosen to modify and in what way, i.e., to what value. When the application under examination modifies a data value, this component will compare the location of the value with a list of addresses that the user has chosen to modify. If the user chose to modify this particular address and that modification is persistent, then the component will undo or prevent the application from changing the data value. Often this will result in the component resetting the data value to that specified by the user.

Finally, the display update modification parser will determine if the user has disabled default display updates or forced display updates at locations where they would not normally occur. This parser directly determines if the environment should perform a display update or not. If the environment must perform a display update then it calls the visualization subsystem.

### 6.1.3 The Real-time Data Modification Subsystem

This subsystem is responsible for handling user specified modifications to subsets of variables. It consists of three components and must provide its capabilities in real-time.
whether the application under examination is executing or not. The environment interface provides the means for the user to specify the data modifications. The data display component of the visualization subsystem updates the visual display to provide the user with an up to date representation of the data values.

The data modifier is the interface between the visualization environment and the application under investigation. It accesses the addresses of the variables chosen for alteration and does the actual modifications. It gets the variable’s addresses, subset ranges, and new data values from the data modification interpreter. It also retrieves the current data values when the data modification interface needs them for display by the visualization subsystem.

The data modification interpreter converts actual data values to the representative values used for display, and vice versa. The visual representations need the data values scaled to a reasonable range for display, whereas the actual data values may contain arbitrary values.

The data modification interface provides the interaction capabilities needed to allow users to identify what variable subsets to modify, what values to set them to, and whether to use persistent or instantaneous modifications. This component also provides feedback to the user, identifying both textually and graphically what the new values are.

6.2 Environment Interface

The environment interface provides the menus, options, and interaction capabilities that allow the user to direct the visualization and execution of the application under investigation. The graphical user interface was developed using the motif toolkit, motif, X11, and Xt. The interface consists of two main components, the main control window and the display window—of which there may be any number. Additionally, popup menus allow access to additional capabilities within the display windows. A final interface component allows
dynamic modification of data values within the display window during execution of the application under investigation.

### 6.2.1 Main Control Window

The main control window, figure Env-2, provides the main GUI interface for interacting with the VISCON environment. Choosers provide the ability to select the application under investigation, the type of application (serial, SIMD, MIMD, distributed), and the visual class (TrueColor, PseudoColor, DirectColor) and depth (32, 24, 16, 8). VCR like controls provide the user with the ability to start, stop, rewind, step forward, speed up, or slow down the application’s execution rate. The interaction mode chooser allows the user to select whether or not data modification made in the display window are to be instantaneous (discrete) or persistent (continuous) modifications. A later section discusses this in more detail.

![Fig. Env-2: The main GUI control window of the VisCon environment.](image)

Through this interface the user also has access to the file menu, figure Env-3. This menu provides the ability to create new display windows, listed as visualization windows in the menu, save and restore configuration information, and set global options.

The save and restore configuration option allows the saving of all information dealing with the current investigation for later retrieval. This capability stores every parameter and user modifiable setting. This incorporates such information as:

- The algorithm under investigation.
• The size and placement of all display windows.
• The algorithm type, algorithm, and visual class of each display window.
• Parameters and options set on a per display window basis.
• The current setting for all global parameters.
• The interaction mode.
• Persistent data modifications instantiated. This does not save instantaneous data modifications as they are completely dependent on when the user makes the modifications. Determining when the user makes instantaneous modifications would require storing the values of nearly all variables at the time the user makes the modification. Persistent modifications rarely suffer this effect.
• Regions tagged for identification.
• Modifications to instrumentation instructions.
• Display updates removed and initiated by the user.
• Overrides of application operations with user specified operations or equations.

The global settings option allows the user to set options and parameters related to the use of color and how colors are allocated. This primarily relates to how many color cells the environment saves for the interface and whether or not the display window should have color cells saved to prevent noticeable palette switching when changing focus between display windows, the main VisCon window, and other X applications. The “Display Operation Text” option enables or disables the display of a textual representation of the instrumentation instruction currently executing. The text bar at the bottom of the main control window presents the textual representation.
6.2.2 Display Windows

The display window, figure Env-4, is the primary method of providing information to the user. The majority of the window provides the data display. The example given here also shows an example of a typical region selection. The bottom of the display window provides status information, including: the algorithm under investigation, the visual class and depth, and the (X, Y) position of the cursor.

The user may create as many display windows as necessary for a given investigation. Each window may examine a different algorithm. In fact, each window may examine a different type of algorithm or class of algorithms. This allows the user to compare several algorithms running under different architectures, even serial architectures, all within the same environment. The algorithms will execute concurrently, each instrumentation instruction will pass control to the next algorithm in a cyclic fashion. This forces a type of synchronization but does not adequately represent the true performance of each application since each application executes a single instruction, no matter how slowly or quickly that instruction executes. One algorithm may execute an addition operation while another executes a string comparison.

Additionally, the user may create multiple windows each of which is representing the same algorithm. Since most of the modifications the user can make to the instrumentation instructions are all window dependent, the user can use multiple windows to represent multiple views of the status of the algorithm. The user can remove display updates from one window to have it focus on a particular segment of the application, the segment currently
under investigation. While other windows can either display the entire execution or focus on other areas of interest. Each window may also focus on different variables. This provides the user with unparalleled ability to select and isolate views. The use of views in this fashion closely matches the way users debug programs, concentrating on one small portion of the application at a time. This allows the user to verify the correctness of the portion of the application under investigation before moving on to another part. This lets the programmer focus attention on only small segments of code at a time and become familiar with its behavior very specifically, which helps the debugging process progress more quickly.

The user can interact with the display window through a sequence of popup menus, to control what, when, and how to display information. The next section discusses these popup menus in detail.

Fig. Env-4: A typical display window with status bar and selected region.

6.2.3 Popup Menus

Popup menus are the primary method by which the user interacts with the environment. The most important popup menu, figure Env-5, provides capabilities for selecting the visualization technique, the parameter mappings for the visualization technique, window option selection, and tagging of selected regions.
The parameter mapping popup window, figure Env-6, allows the user to select the variables within the application under investigation that the environment is to map to each of the parameters associated with the currently active visualization technique. This popup actually generates a hierarchy of menus. At the first level is the association between individual parameters, to which the user will map variables, with the active visualization technique. The next level lists the actual variables available for mapping as well as the variable’s initialization function within the program. The function is necessary since multiple functions may have identically named variables. It also aids in providing context to the user. Variables with a function name of “VISCON” are internal variables to the visualization environment made available to provide additional capability. An example of this is an incremental array that simulates an axis in visualization techniques that inherently do not have any (e.g., the scatter plot visualization technique).
The tag selection menu, figure Env-7, instantiates a highlight or tag on the currently selected region of data in the data display window. The user can set this highlight to be either a single constant color or to a color scale. For each of these options a choice of three colors is available (red, green, or blue). This technique identifies data movement and the impact data elements have on the computation of the remainder of the data set.

![Figure Env-7: Popup menu providing tagging. This example shows the types of tagging that are available.](image)

The options menu, figure Env-8, provides a few options for controlling the size of the display window and the corresponding block size of each data element. This basically allows for the scaling of the window or visual display.

![Figure Env-8: Options dialog box for a display window allowing selection of sizing parameters.](image)

The instrumentation control popup, figure Env-9, provides capabilities for modifying the instrumentation instructions with the goal of changing the behavior of the environment. This interface allows the removal of default display updates and the insertion of others by the user. This allows customization of where display updates occur. The frequency of display updates is also settable, figure Env-10, either per display update or for all display updates. This allows for greater performance by not updating the display during every iteration of the application. Additional commands allow for modification and restoration of application instructions. This restoration is applicable to either the current instruction or to all
instructions. The modification of application instructions uses the instrumentation instruction to replace the results of the original instruction with the results of a user specified instruction or equation. A final option allows for the removal of any persistent data modifications made by the user.

Fig. Env-9: Popup menu for modifying instrumentation instructions.

Fig. Env-10: Dialog box for setting frequency of display updates.

6.2.4 Real-time Dynamic Modification Interface

Real-time modification of data values requires a specialized interface, shown in figure Env-11. This interface allows the user to modify dynamically the data values of the elements within a selectable region or of a single element if the user does not select a region. While this modifies the in memory copy of the data values it is assumed a copy of any unreproducible data resides on permanent storage. After activating this mode, moving the mouse to the right increases the values while moving the mouse left decreases the values. The visual display of the selected elements updates as the values change to provide visual feedback as to the new values. Additionally, the display provides a scaled representation of the element clicked on and its surrounding elements. This scaled representation provides greyscale representations as well as textual representations of the values. Since the background of the text box is much larger than the individual pixels in the visual display, the specific values of the elements are much more easily grasped. As values change, the display updates to correspond with the new
values of the elements. A slider provides a representational percentage value of the central value with respect to a user specified range.

![Visualization Window](image)

**Fig. Env-11:** An example of the interface for dynamically modifying data in real-time.

Modifying data values in this way allows programmers to insert random elements into the executing environment. This can simulate inserting particles into a field or allow examination of specific conditions by setting data element values to preconceived values. Both cases provide the user with the ability to test different aspects of the environment under a controlled experiment.

### 6.3 Instrumentation Instruction Paradigm

This environment uses a full instrumentation paradigm. This means we attempt to place an instrumentation instruction in association with every operation in the application. This results in the placement of display updates wherever the application modifies a variable. In some cases it may be unacceptable to update the display at some of these locations. On the other hand, it may be desirable to update the display at the end of a loop or function where the program does not modify a variable and thus would not normally place a display update. For this reason, we allow the default instantiations of display updates to be modifiable dynamically during execution.
Instrumentation instructions include details that increase the amount of information that is analyzable and resolvable. Each instrumentation instruction includes a unique identifier to differentiate instructions from one another. This identifier allows the user to selectively remove, modify, or add capabilities, i.e., the user can remove default display updates and add others. Additionally, instrumentation instruction calls include information as to what functions are executing and when functions, loops, and conditionals terminate.
7 VISUALIZATION AND STEERING TECHNIQUES

This chapter briefly describes the visualization and steering techniques this research has led to. We discuss some of the reasoning behind each of the techniques here. However, the next chapter provides most of the discussion as to the benefits and possible uses of these techniques.

7.1 Visualization Techniques

Countering the complexity of concurrent systems requires assistance from development tools and debuggers specifically geared towards reducing this complexity. There has been much work on developing visualization techniques to represent concurrent data. Consequently, we do not concentrate as much effort on this area but rather on less investigated areas; areas particularly suited towards aiding our interactive steering approach.

Errors in a program are usually first identified by the results the program generates. It is through the data values being generated by an application that a programmer must identify what is going wrong. Unfortunately, many tools for concurrent systems do not allow programmers to follow data values and observe where the generation of incorrect data values occurs. By visualizing the data values, the programmer can identify when the generation of
incorrect data occurs. By following these incorrect values back to their source, the programmer can identify exactly when the generation of incorrect values occurs and even when the generation of the components of that value occurs. By visualizing operations simultaneously, the programmer can identify how the component values generate the resulting values through sequences of operations. The importance of this task is to identify visualization techniques that can help to highlight data values that are outside the user’s expected range of values. Since unexpected values will change from application to application and even from variable to variable, the programmer must have enough understanding of the application to be able to specify values that the environment should flag or generate highly contrasting displays for. For the most part, the user’s expected range of values will be a single continuous range. On occasion, however, this will not be the case and visualization techniques should provide the ability to handle noncontinuous or multiple ranges of expected or unexpected values.

7.1.1 Data Representation Techniques

Representing data is the most basic approach to assisting programmers of concurrent systems. We provide two simple techniques for representing data, greyscale and scatterplot approaches. The greyscale approach, figure Tech-1, represents the value of each data element as a greyscale value with intensity in the range of 0..255. The limitation of this technique is that it may only represent a single variable and the technique must operate on a rectangular array. If the data is not representative of a rectangular array, i.e., a single linear array, then the visualization technique must force the data into a rectangular format, using row major order.
The second visualization technique incorporated into the VisCon environment is a scatterplot technique, figure Tech-2. This technique requires two variables; each variable maps the location on an axis where an element is to be drawn. A third variable can control the intensity of the drawn pixel. In the example provided here, only a single effective variable is available in the quicksort algorithm. The internal incremental array provides for mapping along the orthogonal axis. To be effective, variables mapped onto a scatterplot technique must exhibit a wide range of values and a high disparity, otherwise too much clumping will occur. This technique works well for the quicksort algorithm, in which there is only a single linear array. It will not work as effectively for the heat dissipation algorithm, in which the variables are inherently rectangular.

7.1.2 Data Movement Techniques

Of key importance to any user or programmer of concurrent systems is the characteristic data movement of the application under investigation. This is due to the importance and impact of the communication network available and how well the data
movement of the algorithm matches the communication network. To examine the data movement of the application we provide the tagging capability already discussed.

The environment can provide tagging by maintaining a separate memory space matching the number of elements in each observed memory space in the executing program. This memory space indicates the tag state of the corresponding element. The tag state indicates if the element is tagged and what type of tag to apply, i.e. its color and opacity.

The instrumentation instructions will indicate what variables or variable elements are being used to modify a resultant value. The tag state of the resultant will be the largest tag state of any of the operands. In this way, there can be a hierarchy of tag states, identifying characteristics of the tag to be applied, i.e., its color, opacity, etc. Instrumentation instructions apply corresponding data movement or initialization instructions to the tag data as was performed on the application data. When displaying data values, the program checks each variable or variable element for its tag attribute. If the tag attribute is active then a separate display routine displays that variable. The time to display the values will compare closely. For parallel architectures, identical parallel data movement commands execute on the tag data as execute on the program data.

The method behind the tagging employed is that the user selects the initial elements of a variable for tagging. From then on, any other variable that gains its value either in part or in totality from a tagged element becomes tagged itself. Tagging in this way can be viewed as contagious since tags are passed through any form of association. This helps identify data movement when the application assigns key variables to temporary variables, to new variables belonging to a different processor, or to intermediary variables used during the transfer of a variable element. This data movement exploration technique is extremely powerful and novel to this environment.
As discussed previously, a constant color, figure Tech-3a, and a color scale, figure Tech-3b, are available to highlight tagged data elements. Each of these techniques has advantages and disadvantages which make them more useful under different circumstances. The constant color technique is more easily observable than the color scale when the selected data elements have very low values that may be indiscernible from black or other very low values. The disadvantage is that the actual values of the highlighted data elements are imperceptible, figure Tech-4a. The values of the data elements are obscured by the use of a consistent color. The color scale, on the other hand, allows the actual data values to be observable. The disadvantage is that with very low values for the data elements it can become difficult to determine which elements are actually tagged. The tagged elements blend in with other values with very low values that are not tagged, figure Tech-4b.

![Fig. Tech-3: Examples of tagged regions. (a) shows a constant color. (b) shows a color scale.](image)

![Fig. Tech-4: Examples showing the advantages and disadvantages of the two tagging techniques. (a) shows a constant color. (b) shows a color scale.](image)
7.1.3 Operation Representation Techniques

A second completely novel technique provided by this environment is the ability to represent operations and execution stacks visually in a graphical form rather than just textually, which is the extent of previous work. The advantages this provides, include:

- Interpretable visually in parallel, rather than serially as with text.
- Maintains user focus on a single display by adding additional visualization techniques to represent other features or attributes, rather than having a separate window as with textual representations. Losing focus of the display window can cause the user to miss interesting events, due to the quick execution of concurrent systems. It also forces the user to manually correlate the data represented in multiple windows.
- Does not interfere with the data display.
- Immediate identification and classification of instructions.
- User specification of which instructions to display and which to ignore.
- Simultaneous display of execution stack with current operation.

Example operation visualizations appear in figure Tech-5. With this technique, the data display window maintains its consistency without interference from the operation visualization that the same display incorporates. Each rectangle around the border of the display represents an instruction on the execution stack. This is similar to concepts from the art world in which a frame around a picture helps to maintain the user’s focus on the picture. The instructions closest to the center are the most recent, ensuring the user does not have to look far for the most recent instructions. The currently executing instruction is the innermost rectangle. Each operation or type of operation has a unique appearance, representative of the operation characteristics. Dashed lines represent conditionals, green lines represent function calls, and yellow lines represent iterative loops.
7.1.3.1 Extensions to Basic Techniques

These operation visualization techniques extend easily to provide additional visual capabilities, as shown in figure Tech-6a. The previous examples only showed the classification of types or groupings of operations. The straight lines shown previously are not a limitation of this technique. Symbols representative of the type of operation are also usable; e.g., infinity symbols can be representative of while loops. Individual operations are also presentable. In this case, the color can help in differentiation when the same symbol has multiple purposes. A yellow dash may represent a subtraction operation while a grey dash represents a conditional in general. Characteristics of the dash (e.g., length and thickness) may vary to identify different types of conditions (e.g., if, else if, for loop controls, and while loop controls) or between different instantiations of the same type of conditional; i.e., the representation can differentiate between two if statements. Another example is in the use of a plus sign to represent an addition operation or a mathematical function in general.

The rectangular format is modifiable further. As the examples show, control of the intensity of the rectangle can identify values of interest that relate to the operation. For conditional loops, shown in red, the left hand side represents the current value of the loop control and the right hand side represents the termination condition. Similarly, for “if” constructs, represented as lines of grey dashes, the left side represents the left hand side of the conditional equation and the right side represents the right hand side of the equation.
Mapping the appropriate value to the intensity or saturation of the corresponding portion of the rectangle provides this representation simultaneously.

Providing a visual representation of the parameters of an operation provides additional interactive capabilities. In particular, the user should be able to probe the display and get feedback as to the actual values of the variables involved in the control expression, figure Tech-6b. This feedback can also provide the relative values used to calculate the actual intensity or saturation. The user can modify this probe value, or perhaps the control equation itself, to change the values within the actual control equation.

These operation visualization techniques provide the user with the ability to identify the current status of loops and correlate the loops with the data currently being displayed. In a standard debugging process the user would have to print out the textual values of the loop elements and attempt to correlate them with the program and the data values within the program. This can be extremely difficult and tedious. The operation visualization techniques provides this correlation visually, greatly simplifying the user's task.

![Diagram](image)

**Fig. Tech-6:** Examples of more integrated operation visualization techniques and probing of visual representation.

### 7.1.3.2 Extensions for Detailing Control Expressions

Control expressions in conjunction with complex constructs, such as “if,” “while,” “for,” etc., provide the control methods of a program. The instrumentation instructions for
these constructs will identify the control expression. Ultimately, many individual component variables make up the control expressions; each of which may have its values analyzed in a normal fashion. An effective way to represent control expressions uniquely is as a triplet of visually representative values, indicative of the left and right hand sides of the control expression under investigation and its result. The type of equality expression is representable by a border around the triplet, again using the picture frame metaphor, figure Tech-7. A solid border around the right side indicates “less than”. A solid border around the right side and a dashed border around the left side indicates “less than or equal to”. A solid border around both the left and right sides indicates “equality”. No border indicates “inequality”. A dashed border around both sides indicates “less than or greater than”. A solid border around the left side indicates “greater than”. A solid border around the left side and a dashed border around the right side indicates “greater than or equal to”.

For serial portions of code, the control expression visualization incorporates a single example of this technique. For parallel portions of code, particularly in the SIMD case, the techniques will use a per processor representation using a data visualization paradigm, i.e., a display with a representational visualization for each processor in a matrix format.

![Fig. Tech-7: Examples showing the different conditional types.](image)

The display should allow direct interaction to provide feedback as to the exact values of the control expression and the component variables, in contrast to the representational value provided by the visual display. This visual feedback should provide a mechanism for changing the values of any of the component variables, resulting value, or the algorithmic expression through the use of direct manipulation and interactive computational steering.
Finally, an optional toggle should allow the user to turn an indicator on or off for when data values have changed between probings. This indicator is representable as a small modification to the normal display; i.e., a small uniquely colored square in the upper left corner of the representational visualization of the control operation. The environment removes the flag or indicator after a probing, only to restore it when the value changes yet again.

Many iconographic techniques can apply in a similar fashion; each attribute of the icon taking on an attribute of the information to be displayed. In fact, this technique is describable as an icon.

### 7.1.4 Techniques for MIMD and Distributed Systems

MIMD and distributed systems differ from SIMD systems in the number of processors, the amount of work each processor performs, and the lack of synchronization. To handle these differences, the visualization techniques must apply to slightly different architectural considerations. The synchronization issue is rather easy to handle. For the most part, the visualizations for each processor in a MIMD or distributed system can be calculated and displayed independently. The visualizations are not dependent on one another. The key here is that the visualizations should remain separate. Different regions provide each processor with its own visualization, even when they are displaying the same variable, figure Tech-8. In the case where synchronization is desirable, to display a variable’s data elements in a single unified display, barrier synchronization is useful to force the program into a synchronized state. In a unified display, it may not be clear which processor owns which data elements. This can become just another mappable parameter for the visualization technique, i.e., another limb in the stick figure icon.

Choosing whether to use barrier synchronizations or not in the above example is dependent on what type of analysis is being performed. For application comprehension, to aid
understanding of the application’s behavior, it is usually more beneficial to perform the barrier synchronization. This is seen in most if not all algorithm animation systems which are specifically geared towards aiding comprehension. While this type of animation can be useful for debugging, it can be misinterpreted and can hide important information. The problem lies in the fact that performing the barrier synchronization eliminates the “state” of the processors that may be causing problems to occur, removing inconsistencies between the processors that are being communicated. This makes it difficult to identify what may be causing errors. Thus, for application debugging it is best to not perform barrier synchronizations.

A second problem of interest is communication between processors. While communication between processors at the general level is easy and is doable in many ways, i.e., a border of the processor display indicating data being sent to a particular processor and a different border indicating data being received from a particular processor. For a small number of processors even the typical lines and arrows approach is appropriate. The difficulty comes when the environment must identify the actual data elements being moved. In this case, the visualization technique must indicate where each data element originates from and how recently. Since data transmission will seem almost instantaneous in an executing program, we must provide some persistence in the display of data communication to provide the programmer or user with time to absorb, analyze, and understand the data communication that took place. Again, this is representable simply as additional mappable parameters to the visualization technique.
Fig. Tech-8: Example showing extensions to MIMD and distributed systems. Lines and arrows show communication network. Grey indicates no communication. Red scale indicates communication and relative value of data.

7.1.5 Techniques for Long Running Programs

The techniques in this thesis become particularly important for very long running programs, since these programs are even more tedious and time consuming to comprehend and debug. As a consequence of the length of execution times and the inability for users to be available for the entire execution time, additional capabilities are necessary to assist users in analyzing these types of programs.

As a consequence of the time required to update the display, the user should have the ability to reduce the number of display updates; this is the capability discussed in section 6.2.3 in relation to frequency of display updates. Disabling the display update instrumentation instructions will cause this to occur but it should also be accomplishable through other means. One such technique is the ability to have an instrumentation instruction only update every N calls to that instrumentation instruction, based on the ID associated with the instrumentation instruction. This will greatly increase the performance of the analysis process as the program will no longer force a display update every single variable change, a very time consuming process. A second technique is to globally force all display update instruction to execute only every N times. This is accomplishable in one of two ways. First, the environment can associate
a separate counter with each display instruction or a single counter with all display instructions as a whole. The user should be able to select between the techniques. Figure Env-10 shows a dialog box that provides these capabilities.

An added complication lies in the fact that the user may not be available for the entire execution of the program; some scientific programs take days or weeks to execute. In this case, the user must be able to identify when the program should stop executing. This can be done by specifying desirable values for variables, such as loop counters, error values, array elements, etc. In the case of array elements or other variables, it could be a check for an unexpected value. While storing a log of the entire execution run of the program is usually unfeasible, it is reasonable to log the last several minutes of execution. This will identify to the user when an error occurred and potentially show where and how the error occurred. Even if only the location of the error is identifiable it will provide the user with a time frame on which to focus and an initial set of data values to work with. By using the ability to change the configuration of the system, the user can put the environment into a state identical to what occurred far into the execution.

Ultimately, the user must identify what to flag as an event worthy of investigation, i.e., the types of values to flag. This is primarily a consequence of the fact that for different applications, the same values or events may be either expected or unexpected. It is up to the programmer or domain expert to be familiar enough with the application under investigation to be able to specify what constitutes an interesting or unexpected result for that application. A further complication is the difficulty in differentiating between interesting events and unexpected events.

When a condition the user specifies occurs, the program will most likely need to pause and wait for user intervention such that the user can investigate the event. In some cases, such as a very long running program, the user may wish the environment to continue...
executing but also to log characteristics of the event. This could require storage of large amounts of information, including the values of all data elements and variables at the time of the event.

### 7.2 Steering Techniques

This section describes ways in which the capability and usefulness of the visualization techniques already discussed improve with the incorporation of capabilities through which the user can interact with the visualization techniques. We provide interactive computational steering techniques that the user applies directly through the visualization. We discuss modification of both the data and the operations within the program under investigation.

An important goal for our work is to provide techniques that will allow the user to interact directly with the executing program and make changes without having to restart the program. Our desire is to provide capabilities to aid the user in gaining an in-depth understanding of the program in a timely fashion without having to waste considerable time editing, compiling, and rerunning the program. To this end, we explored the applicability of interactive computational steering to aid in the debugging of concurrent programs. We have explored the use of steering to different levels of the executing program.

#### 7.2.1 Program Level Steering

At the highest level, the user can slow down or speed up the execution rate of the program. A VCR like control panel controls the execution rate. The VCR controls provides the ability to increase or decrease the execution rate by factors of 1/100 or 1/10, to stop the program, and to restart the program. While the user can slow the program, even to a complete stop, we currently do not allow the program to run in reverse. Allowing reverse execution is a possibility since we are fully instrumenting the program under analysis but would require enormous amounts of data storage that we currently are unable to provide. Providing reverse
execution would aid usability and should be available in a complete environment. This is not novel and should not be difficult to incorporate given the storage caveat.

Providing the user with the ability to control the execution rate of the application under investigation presents additional issues. This particularly relates to the need to execute the program at an appropriate rate while maintaining effective user interaction and includes the following issues:

- Application Instructions take varying amounts of time to execute.
- Instrumentation instructions themselves can take a very long time to execute or almost no time.
- Calls to the X toolkit can be very slow, particularly queries.
- We must continue to provide sufficient interaction, no matter what rate the program is currently executing at.
- We do not want to significantly inhibit the execution rate with excessive checks for user interaction.

This means that we can not merely pause the application under investigation to slow down its overall execution rate. We must continue to check for user interaction and incorporate the time required to perform those checks into the delay. In fact, queries to the X toolkit are sufficiently slow that they can act as the delay themselves. Additionally, when the user wishes to execute the program as quickly as possible we must reduce the number of checks to the X toolkit so as not to impact the execution rate. On the other hand, when the user does begin interacting with the environment we wish to check for additional interactions more frequently, until the program can determine that the user has discontinued interaction. This can be accomplished by applying appropriate locality of reference models to the delay code.
7.2.2 Instrumentation Level Steering

At the next level, the user can interactively control the instrumentation. By default, the instrumentation instructions force a display update whenever a variable upon which the display is dependent, changes. The default display updates can lead to awkward animations of the display as temporary results also force the display to update with the corresponding data. Updating the display during these temporary computations often is not desirable.

To reduce or even remove these artifacts, the user can remove display updates from their default locations and force display updates at others. The user can then remove display updates resulting from temporary results of a computation and force display updates at more appropriate locations during the computation. This extends most work to date that has either incorporated full instrumentation of the program, without providing the user with the ability to change the instrumentation, or forced the user to adjust the instrumentation, recompile, and re-execute the program. Having to change the instrumentation instructions manually can be a very time consuming approach.

As the modifications to the instrumentation are window dependent, a user can thus have multiple windows examining the same executing program. One window may display an optimal animation through the removal of display updates based on intermediate computation and the addition of updates at the end of an iteration or other appropriate locations. A second window can provide only intermediary computation, making it readily apparent when that intermediary computation is occurring. A third window can show the unmodified display. Using such a multiple window approach gives the user unparalleled ability to select views of the application that are pertinent to the current task. The instrumentation modifications are removable on a window by window basis.
The user may save or restore the state of the environment (i.e., window settings, parameters, execution state, etc.) at any time, permitting the appropriate instrumentation modification to be reloaded to test a new program version under the same characteristics.

### 7.2.3 Real-time Data Modification

The third level of steering we provide is direct modification of actual data values. Direct data modification is a more conventional steering technique except we are applying it to concurrent data and allow entire regions of data to change simultaneously. The user can directly manipulate the data values of selected elements. By clicking the middle mouse button the user enacts the data modification operation, which is independent of the data tagging mechanism, figure Env-11. All values within the selected region have their values increased or decreased an amount corresponding to the distance the mouse moves to the right or left.

To reduce the range of values the user must deal with, we provide the ability to select the minimum and maximum range for all data interaction. Boundary conditions constrain the values of the data elements. Setting multiple data elements to the same value is possible by forcing all the elements to the minimum or maximum value in the preset range and then selecting the desired value.

While it is more common to pause execution of the program under investigation before beginning to change values of data elements, it is not a necessity. The user may allow the program to continue executing while data modifications are being made. If data elements are being moved among processors or to different addresses then modifying data elements as the program executes could have the affect of continuously modifying different values as they pass through the selected region. In addition, it is possible to modify only a single data element by not having a region selected when enacting the data modification capability. The data element currently under the cursor will change.
Data manipulation is applicable not only to concurrent or array data. Any variable ultimately is modifiable. By modifying selected variables it is possible to direct the flow of execution to regions of code that need testing. For example, it is possible to change variables used in the test conditions of “if” statements to force execution down specific branches. By changing the values of control expressions the user can force the program to execute exception conditions which normally would not occur. This allows the user to verify that should something unusual or unexpected happen that the program will still perform as expected and ensure that exception code executes as needed. It is also possible to modify variables used in the control expressions of loops, functions calls, etc. We place no limitation on what data values may change.

It is important to note that changing data values does not always make sense. For example, consider a large application with many components. A common component of most applications is some form of sorting routine. It would make sense to modify data values before calling the sorting routine but doing so in the middle of the sorting routine would generate unusual results. However, if one wanted to explore the impact of a hardware fault on the sort routine, one could fix a data element to a specific value and watch its effect.

Providing for the modification of data elements makes it easier to test specific values. These values can be test conditions, boundary values, or any form of preconceived data to test the operation of the program. By combining data modification with data tagging it is possible to modify data elements to specific values and then highlight those elements so they are easily traceable.

7.2.4 Persistent Vs. Instant Interaction

The data level steering capability is applicable in one of two ways. The method used in the description of the previous section deals with the more conventional approach that we
term instantaneous interaction. We propose a modification to the conventional interaction technique that we term persistent interaction.

### 7.2.4.1 Instantaneous (Conventional) Interaction Techniques

Typically, when a user interacts with the environment the modifications take place immediately and when the user discontinues interaction the modifications terminate. In our environment, when the user is modifying the values of a region of data elements, the modification of data elements stops when the user no longer holds down the middle mouse button. There is a one to one correspondence between the user’s action and the modification to the data elements.

### 7.2.4.2 Persistent Interaction Techniques

There are occasions when the instantaneous technique is insufficient. For example, a user may wish to have a region of values forced to a specific value continuously to change the characteristics of an algorithm or to change the environment that the algorithm is attempting to simulate. This can change or add environmental characteristics; e.g., for the heat dissipation algorithm, the user can add new boundaries that do not take part in the algorithm. The user can also add new sources and sinks in a similar way. For such cases we use the term “persistent interaction.” When a user switches to persistent interaction mode the data modification process remains identical to previous descriptions. Internally however, instead of merely making the noted changes to the data elements, we maintain a list of all the persistent interactions the user has made. Whenever an instrumentation instruction executes that indicates a persistent modification-flagged variable has changed, the values of the elements are reset to the values initiated by the persistent modification. Note that only data elements that specifically change are reset; all other elements of the variable remain unchanged. When restarting an algorithm, the environment maintains the persistent interactions currently in
effect. Thus, after designing a set of desirable interactions, the user can restart the algorithm from the beginning to examine the execution of the algorithm with these new characteristics from start to finish. The list of persistent modifications is removable on a window by window basis so that the user need not remove all modifications simultaneously; though an option is available to remove all interactions simultaneously.

The method we use to incorporate persistent interaction into our environment is fundamental. We use a common metaphor often used to rotate 3D models continuously. If the cursor is moving when the user releases the middle mouse button then the user selects the persistent mode of interaction. Otherwise, the user selects the instantaneous mode. This interaction metaphor maintains the direct manipulation philosophy of the environment without losing the ability to choose between the instantaneous or persistent modes. To make the environment easier to use for all users we do support a toggle that will force the persistent interaction mode to take place.

7.2.5 Dynamic Instruction Modification

The instrumentation paradigm shown below allows the modification of any and all operations during the execution of an application. It may be performed both at the instruction level and at the equation level; i.e., modifying the assignment operation. In locations where user changes to the instructions occur, the environment must provide some form of visual feedback to indicate when those modified instructions are being executed. This can be as simple as a flash of the display or a led indicator in the status bar of the display window.

When the user wants to make changes, the user stops the executing program and brings up the instruction editor. The user types in the new operation or equation that then gets stored in a list in association with the original instruction. An interpreter and parser then
execute on the newly entered equation to evaluate it whenever necessary. It can also be pre-parsed to make execution quicker.

The user should be able to change any characteristic within the program and at any level. Namely, the user should be able to change not only a single operation within an equation but also the entire equation with a single replacement equation, i.e., the user should be able to modify the assignment operation rather than having to modify each operation originally in the equation individually. If the user modifies an instruction more than once then the last modification is the only one that actually executes, otherwise confusion will occur. This technique can be quite useful in some algorithms, such as selecting the partition element in quicksort. For a particular data set, the user may wish to try out different partition element determiners, since the partition element greatly affects the rate of execution of the algorithm. This technique allows modification to the routine dynamically without recompiling or even leaving the executing environment. The partition element is modifiable during execution to improve performance of a subset of the execution trial.

The following code segment shows how instrumentation instructions within a program can allow the modification of the algorithm. The equation to solve for a variable, res, is res=a*b+c+1. We add instrumentation instructions for each operation (*, +, +, and =). The user can modify a single operation; i.e., the user can change the multiplication to be division or the multiplication to be a whole equation in and of itself. Alternatively, the user can modify the assignment operation to change the entire equation as a whole in a single swoop. If you look at the sample code for the instrumentation instruction itself you can see that all the instrumentation instruction need do is check a database to see if the user has applied a modification to the instruction currently under investigation. If the user has applied a modification then the application uses the results of the modified instruction for the remainder of the computation rather than the original results. Otherwise, the application uses
the original results. The way in which the environment adds instrumentation instructions ensures that the remainder of the equation will use the value returned by the instrumentation instructions correctly. This allows the instrumentation instruction to control completely the value used by the remainder of the equation.

```c
#include <stdio.h>
#include <stdlib.h>

int internal, opres;

main()
{
    int a, b, c, res;
    a=5;
    b=10;
    c=15;
    res=(opres=a*b, opres=instr(ID, MULT, a, b, opres)+c,
        opres=instr(ID, ADD, internal, c, opres)+1,
        opres=instr(ID, ADD, internal, 1, opres)),
        opres=instr(ID, ASSIGN, internal, internal, opres);
    printf("%d\n", res);
}

int instr(int ID, int type, int a, int b, int opres)
{
    if (checkModified(ID))
        internal=modifiedResults(ID);
    else
        internal=opres;
    return internal;
}
```
8 THE ANALYSIS AND DEBUGGING PROCESS

This chapter discusses how the visualization and steering techniques investigated during the course of this research apply to the analysis and debugging process. Here we explain how these techniques are useful to this process. For this discussion we will use several applications as examples, namely a sorting algorithm and a heat dissipation algorithm. We first discuss these two algorithms. A discussion of errors or situations that are likely to occur and how to resolve them with the techniques in this thesis follows.

8.1 Example Algorithms

This section provides two examples to which we will apply many of the discussed techniques. The first example, an even-odd exchange sort, is simple but makes it easier to understand the techniques. The second example, a heat dissipation algorithm, is more realistic of what scientists and other users might encounter.

8.1.1 Even-Odd Exchange Sort

The even-odd exchange sort is a simple sort geared towards SIMD environments in which the algorithm sends data to adjacent processors for comparison and swapping. The algorithm works by comparing data elements stored by adjacent processors, first comparing the values on even processors with values adjacent to the right and then values on odd
processors with values also adjacent to the right. The algorithm exchanges values when necessary. During the swap routine, the visualization display updates continuously, leading to artifacts as intermediate points during the swap force these display updates. Consequently, when examining the execution of a concurrent sorting algorithm, it is desirable to examine the data after each full iteration, figure Proc-1a. Examining the intermediary results, figure Proc-1b, can lead to a confusing animation that distracts from the desired impact of the changing data more than it aids understanding.

![Fig. Proc-1: Example execution of even-odd exchange sort.](image)

Using the instrumentation level steering, we can remove the display updates from the swap routine, such that the environment does not display intermediary results, and add display updates at the end of each iteration or immediately after the swap routine. The modified display updates will generate the desired result of having a smoothly animated display, showing the execution of the sorting algorithm in which the exchange routine is an atomic element rather than being broken down by the instrumentation instructions.

### 8.1.2 Heat Dissipation

Our second example is a heat dissipation algorithm consisting of a source, a sink, and a slotted barrier between them, figure Proc-2a. The remaining values are instantiated to be the average of a source and sink value. The algorithm then simulates heat dissipation using a convolution class algorithm in which values do not exchange through the barrier except for where the slot occurs, figures Proc-2b and Proc-2c.
8.2 Application Comprehension

Situations often occur when a user or programmer must understand an application or an algorithm within an application with which they are currently unfamiliar. This can result from adding new programmers to a team, the learning process, receiving an algorithm from a published source, etc. Understanding an algorithm often requires more than merely examining the algorithm code. It requires knowing the behavior of the algorithm, the impact of various aspects of the code on the algorithm as a whole, how data changes over the course of execution, and how data moves between processors during the course of execution.

8.2.1 Understanding Algorithm Component Importance

When attempting to understand an algorithm, it is important to understand the importance and impact of individual components of the algorithm on the algorithm as a whole. In many cases, algorithms have components that are particularly important.

For example, most universities teach students of Computer Science the Quicksort algorithm. An important part of this algorithm is the partition element or more precisely the partition element determiner. The partition element can greatly impact the execution of the algorithm. For individuals learning the algorithm for the first time it is beneficial to their comprehension of the algorithm to examine different determiners and the corresponding affect on the algorithm. This would normally require modification of the source code,
recompilation, and then re-execution of the algorithm. The VisCon environment can speed this process by allowing users to change the partition element dynamically during execution. This prevents the need to actually change any code. This is achievable using the real-time data modification interface. Note that the user can use the instantaneous mode to change the partition element for a single iteration or the persistent mode to set it for the remainder of execution.

The ability to change the partition element dynamically provides added benefit in that the user can truly see how changing the partition element dynamically impacts execution. The previous work section discusses the importance of this type of real-time interaction.

The data modification techniques provide one method of modifying the partition element. A second technique is through the use of the instruction modification interface. In this way, the user can modify the determiner algorithm itself, allowing the partition element to change naturally with the course of execution according to the newly insinuated algorithm.

### 8.2.2 Understanding Data Movement Characteristics

The use, perhaps improperly, of the interconnection network of a concurrent program can often be the most important determinant of that program’s behavior and performance. Understanding this behavior can be difficult since just looking at text values or array indices to gain this understanding is very time consuming and confusing. Consequently, we provide several techniques that visually aid in understanding the inherent communication of the program.

The tagging technique in particular is useful for comprehension. By tagging a single element the user can observe and follow that element as it changes and moves to other processors. This is not possible with the standard approach that concentrates on the processor. The standard approach allows examination of the activity of a processor and the
data associated with that processor. It loses focus on data elements as soon as they transfer away from the processor under investigation.

The provision of both scaled intensity and solid highlights insures that any data element is traceable, no matter its values. This allows us to follow data elements throughout the entire execution of the program and to test all types of boundary conditions and special circumstances.

In many cases it may be beneficial to know where data values end up in relation to one another after execution. During execution we would like to see how the data elements move to their final destination, i.e., how do the processors pass the data to get the data to their destination. Ultimately, just watching the execution of the algorithm and the movement of the greyscale values can give a sense of the motion and behavior of the algorithm over the interconnection network. However, often we will wish to focus on specific elements or be able to follow the same elements from start to finish. Data tagging satisfies these requirements. Figure Tech-3 shows two initial configurations of highlighted elements in an even-odd exchange sort. Figure Tech-4 shows the positions of those highlighted elements after the even-odd exchange sort has nearly completed execution.

Merely viewing the greyscale values as they move between processors can sometimes generate artifacts. In the even-odd exchange sort, the continuous movement of data values generates a smooth animation. While viewing this animation it would appear that there are large numbers of high values, bright pixels, moving towards the bottom right of the display and that no values are moving to the upper left, which would be the lower values. This is actually a side effect of the eye being attracted to the bright pixels and the interpretation of the black pixels as being part of the background. If we apply a random colormap to this display we observe an even distribution of values moving in both directions, which is clearly not visible with just the greyscale view.
Tagging data elements in this way will usually result in one of two effects. First, the contagious approach to tagging can result in a wide distribution of the tag element, which is seen with the heat dissipation algorithm. Second, it can merely highlight a single element throughout computation as it moves between processors, which is seen in the even-odd exchange sort. It is important to note that without the contagious approach to tagging even a simple sort could not effectively be examined. This results from the fact that at best the swap routine’s use of temporary variables will remove the tag from the data element. At worst it will end up merely tagging the same processor or array position, since there would be no method for moving the tag elsewhere. Consequently, without contagious tagging we would revert to processor tagging which has been done previously and was not our goal.

8.3 Application Debugging

Application debugging differs from application comprehension in that the user is trying to identify if, where, and potentially why the application is generating incorrect values. Errors in the execution of a program are usually first noticed through the resulting output values. It is the result of the computation that ultimately shows whether the program is executing correctly or not. Please note that we feel very strongly that the use of domain experts is imperative to the correct development and debugging process. It is the domain expert that will be able to provide insight as to how to perform a given task and to the correctness of the results.

Even a very simple algorithm can have errors. If programmers do not know what the resulting data is supposed to look like, they will not know what to look for when ensuring the correctness and accuracy of the program. Domain experts, on the other hand, should be able to point out unexpected or undesirable results almost immediately.
Since it is through the results that errors will first be identified, we concentrate on this data. We provide techniques for viewing the data visually and for analyzing the details of this data and the behavior of the algorithm surrounding the generation of this data. By viewing the data in this way, the programmer can more quickly identify when the application generates incorrect data and, what is more important, why.

This section concentrates on techniques provided by the VisCon environment that aid in verifying the correctness of different aspects of the application, identify errors, and locate the origin, cause, and extent of errors. Ultimately, these features should greatly reduce the time required during the debugging process to complete the algorithm.

### 8.3.1 Verifying Border and Boundary Integrity

Since we are using a SIMD architecture organized into a linear array configuration, we must ensure that data is not accidentally being used across the borders of the calculated area. One method of verifying border integrity is through data tagging. Because of the contagious approach being used, tagging regions of data on the very edge of the calculated region will highlight any use of the data crossing the border. Figure Proc-3a shows the correct execution of the heat dissipation algorithm. Processing does not cross the boundary of the operated area. This mistake is easy to make when the architecture is being forced to simulate a configuration not directly representable. In this case, we are forcing a linear array of processors to simulate a rectangular array. It is very easy to mistakenly allow the computation to cross the artificial border. This will result in a display similar to that shown in figure Proc-3b. Notice that as the tagged elements cross the border they shift down one row, due to the organization of the linear array in the matrix, figure Proc-4. This artifact is easily observable using data tagging and our contagious approach to tagging. Without these techniques, this error would be extremely difficult to identify quickly.
An added complexity arises in our test architecture since it incorporates a toroidal component. This means that the end of the linear processor array actually loops back to the beginning of the processor array. In this situation, we must insure that data does not accidentally pass across the end of the array. Figure Proc-5a shows the initial configuration of a test run of the heat dissipation algorithm in which we have tagged the very last element on the processor array. The correct execution of the algorithm, figure Proc-5b, shows that data does not pass across the end of the processor array. An incorrect execution, in which data does pass across the end of the processor array, is shown in figure Proc-6c. Upon initial examination of the incorrect image you would expect the colored region in the upper left hand corner, generated from data crossing the end of the processor array, to be nearly the same size as that in the lower right hand corner. What this indicates is that there are likely several circumstances in which computation occurs at the end or beginning of the processor array and not all of them are incorrect. It is actually more likely that only one of the computational
routines is incorrect. This indicates the need to examine all portions of an algorithm before
determining the correctness of the algorithm as a whole.

Fig. Proc-5: Examples showing correct and incorrect operation at the end of the processor array.

To ensure the barrier and borders are not taking part in the computation of the heat
dissipation algorithm they must not count as computable elements. By using multiple
windows to display different variables, we can include the variable holding the count of the
number of valid elements to be incorporated into the result and ensure it is correct for all
regions of interest, figure Proc-5a. We know that there can be a maximum of five elements
contributing to the value of any element. This includes the element itself and the four
surrounding values; values along the diagonals do not take part in the computation. Visually,
the value of the elements is easily identifiable. Probing is useful to verify the values,
particularly in areas where the contrast insufficient. In order to provide visual displays for a
wide range of applications, each of which may have a different range of variable values, we
provide the ability for the user to set the minimum and maximum expected values of the
variables. The environment then maps the range selected by the user onto the full range of
greyscale values. In this way, we can change the contrast of the image in figure Proc-6a by
changing the maximum value from five to six, figure Proc-6b. If a value falls out of the range,
then an unexpected color will result. The user can set this to any color that is easily
observable, such as red, figure Proc-6c. The example here was generated by setting the
maximum expected value to four. The red area incorporates those values that are greater than four, five in this case. This technique makes unexpected results stand out like a sore thumb. Effectively, this allows the user to specify what should constitute an unexpected or undesirable value by setting the range of expected values in the interface and having all other values which fall out of this range be automatically highlighted by the environment. By ensuring this color is not used elsewhere in the display the user can quickly identify where further investigation is needed. Using this technique the user can quickly verify the values of each variable by setting the range appropriately for the given variable and examining the execution of the application.

Data tagging is also useful to verify which elements the application uses in a computation. In figure Proc-7a we have highlighted the barrier that should not be taking part in the computation. In a correctly executing algorithm this highlight should either just disappear, in the case where the barrier is consistently reset to zero, or should remain highlighted but not impact the surrounding elements, in the case where the barrier does not partake in computation but is not consistently reset to zero. Also notice that because the data values of the barrier are zero, the intensity highlight would not be observable. Since this is our focus in this investigation we must overcome this difficulty by using the solid highlight option.

In the case where the algorithm is incorrect, the barrier will in fact take part in the computation of the surrounding elements. This will most likely result in an image similar to that shown in figure Proc-7b. In this example, the barrier remains highlighted and the
surrounding elements are quite obviously taking part in the computation as the tag contagiously passes to them. Figure Proc-7c shows an example in which the barrier loses its tag. This indicates that the barrier’s value has been reset to zero, which was not the case of first example. In both cases the barrier continues to impact the computation of the surrounding elements.

![Fig. Proc-7: Examples showing data tagging used to verify the correct use of barrier elements in the computation.](image)

Tagging data elements in the middle of the operational area, figure Proc-8a, provides a final test of the integrity of the barrier and borders. In this way, we can verify that the computation does not incorporate or pass over the barrier, figure Proc-8b. If tagged elements should appear to jump over the barrier, figure Proc-8b, then the implementation of the barrier is incorrect. We can also verify that the computation does, however, pass through the slot in the barrier, figure Proc-8d. At the same time we can see if computation is crossing the border in any direction.

![Fig. Proc-8: examples showing the correct and incorrect use of the barrier.](image)
8.3.2 Verifying Algorithm Configuration

Our ability to change the initial configuration of the environment permits the examination of a variety of initial and boundary conditions. Will the algorithm work with different configurations of sources and sinks? What if a sink or source is adjacent to the barrier? What if the source and sink are adjacent? What if there are multiple barriers, sources, or sinks? Would the algorithm still operate correctly in these cases? These are the types of questions the developer would ask after the initial implementation of a heat dissipation algorithm. Normally, verification of the algorithm’s integrity under different configurations would require either modification of the algorithm itself to change the configuration or manual creation of environment description files, which would then require testing and debugging themselves. This can be extremely time consuming.

Since the persistent interaction technique effectively guarantees that the selected elements will always be equal to a particular value we can use persistent interaction to simulate sources, sinks, and barriers. Generating test scenarios using persistent interaction greatly reduces the amount of time required to test different types of configurations. It allows the user to generate, quickly and dynamically, new test scenarios without needing to repeatedly stop and restart the program. The user can make the modifications while the program is running such that the user can see the actual configuration they have created and make corrections instantly. Figure Proc-9a shows a new configuration in which a source has been added close to the sink. Figures Proc-9b and Proc-9c show the execution of the algorithm under this new configuration. Since the newly added source does not dissipate during the course of execution we know that we can correctly add new environment characteristics anywhere and have them operate correctly. By looking at the result display, we can verify that the newly added source is acting properly and that it does not disrupt the correct operation of any of the other features in the environment.
The ability to save and restore the environment characteristics within the VisCon environment insures that we can create a variety of useful test scenarios, some of which may show errors in the algorithm, and restore them exactly when we wish to re-test them on a new version of the algorithm. This not only provides a very effective testing and debugging facility but also provides an automatic environment configuration tool to any application incorporated into the VisCon environment without the need to write any additional code.

Fig. Proc-9: Examples showing persistent data modification to add a new source to the environment.

8.3.3 Verifying Data Values

The user can use discrete interaction to test the effect of changing the default values of the elements in the environment, not including any sources, sinks, or barriers, or the effect of having different portions of the environment initially set to different values. In figure Proc-10a the user has set a region of values to a different initial condition. This modification is not persistent so the data values can change over the course of execution. As can be seen in figures Proc-10b and Proc-10c, the modified values will dissipate during the course of execution. This indicates the algorithm executes correctly even without the application of a source or sink. The user can use this to verify the correct execution of the algorithm even in areas where it does not appear that computation is taking place. It also allows the user to verify the correct execution of the algorithm in situations where it would usually take an extremely long time to get to a particular setup. By changing data values dynamically, we can setup the environment to some future stage and allow the application to execute normally.
from that point on. Thus, we can debug that future stage without having to wait till execution
reaches that stage naturally.

![Figure Proc-10: Example of instantaneous data modification.](image)

Additionally, the capability mentioned previously which allows the user to set the range of
values expected for a given variable allows the user to increase the resolution of the data
display. By setting the value range to a range of particular interest the user can observe more
detailed changes in values. Effectively, this allows the user to zoom in on a selected range of
values, filtering out the surrounding values, and providing more detail on the values in the
selected region.

### 8.3.4 Debugging Loops and Conditionals

The principal control elements of an algorithm are the conditionals and conditional
control elements of loops. These conditionals identify the behavior of the algorithm. They can
also be the most complex and confusing portions of an algorithm. Verifying the correctness of
and debugging these conditional expressions is of great importance.

To an extent these conditionals consist of variable components that the user can
analyze and debug just like any other variable. This can be difficult, though, when mentally
trying to bring the components together to determine their meaning. For this reason, we
incorporate new techniques into the VisCon environment specifically to aid in this process,
described in sections 7.1.3.1 and 7.1.3.2. These techniques provide visual representations of
the meaning of the control expression. By viewing the value of the termination condition and
the current value of the control expression we can determine if the loop or conditional is
working correctly. Signs of a potential problem include:

- the intensity of the current value is getting farther away from the termination
  condition
- the loop or conditional terminates before the current value reaches the termination
  condition
- the intensity of the current value passes the termination condition
- the loop or conditional continues executing when the termination condition is
  reached
- the termination condition is changing in relation to the current value
- The current value does not appear to be changing

Note that with the techniques provided, the value of the termination condition and the
current value of the control expression are representable visually. Thus, the user can
determine visually whether the application achieves any of the indicators in the above list.
The visual representation also identifies the type of control expression, e.g., equality, less
than, greater than, etc. The indicator list above assumes equality is the desired termination
condition. All other relationships have similar attributes; the indicators merely need slight
modifications according to the corresponding relationship for termination.

A common problem is the use of an incorrect conditional within a control expression;
this can be something as simple as using “=” (assignment) instead of “==” (equality). This is
potentially observable from the control expression never evaluating to the expected result
clause. The value of the control expression can also indicate this through changes made to it
during evaluation of the control expression.
8.3.5 Debugging the Execution Stack

The execution stack of a program consists of all functions or routines that remain in effect for some time after their initial invocation. This includes: function calls, loops (for, while, do), and conditionals (if, switch, if else). While these routines can complete execution instantaneously, they very often do not. This is particularly true for recursive functions. This technique effectively can tell the user if a routine executes at all or not and if so when and for how long.

A typical recursive function will interweave function calls with conditionals. A definite pattern is observable and expected. A broken pattern or the lack of a pattern altogether indicates an error in the program. At the end of execution, the execution stack should be completely empty. The visual representation of the execution stack follows suit with the actual execution stack within the application. If the visual representation of the execution stack indicates that the execution stack was not empty at the end of execution then there is a problem with the program. Visually, the user can determine if the execution stack is empty if no square rectangular borders, section 7.1.3, remain on the screen. If any such borders do remain, then the execution stack is not empty.

The purpose here is to identify that a problem is occurring and providing a means for classifying the problem. Knowing that there is a problem with the way the execution stack is operating gives a direction for further analysis.

8.3.6 Verifying Algorithm Impact

When viewing the execution of an algorithm, it can often be difficult to verify that activity is occurring if no visual changes occur. The heat dissipation algorithm for example initially consists of large areas of equal values that are not changing. Is computation still being performed on these elements even though no change is being applied to the elements?
We can change data elements to force a difference in the visual display but that does not answer the question since we are in fact forcing computation to take place when we change the data values. By tagging a small region of the operational area in which we do not know if computation is actually occurring, figure Proc-11a, we can determine if computation does occur when the application is not actually changing the data elements, figure Proc-11b. As can be seen, even though the elements are not having new values computed, they are taking part in the computation and generating new values. This could be an area for performance improvement.

![Fig. Proc-11: Example of tagging showing that computation is progressing even though there is no visual indication of computation.](image)

A second issue related to the impact of data relates to the inability to identify visually extremely small changes in data values. In figure Proc-12a we have tagged the source in the heat dissipation algorithm. The tagged region in figure Proc-12b shows the full region the source has left an impact on. The region in which the intensity of the displayed elements has been changed is much smaller than the tagged region. This results because while the source impacts all the data elements in the tagged region, the amount of impact is extremely small, a mere fraction of the amount needed to result in a change in intensity. This results from the limitation of current display technology that limits us to about 256 intensity values in a color or intensity scale. In addition, the human visual system is logarithmic in nature which makes it difficult to accurately discern more values than display systems currently provide. By using the full range of hues and intensities we can greatly increase the number of available colors.
8.4 Identifying and Correcting Errors

The ultimate goal of any debugging environment is to aid users in identifying and correcting errors and to do so in a more efficient manner than is possible otherwise. The VisCon environment provides this ability through the use of the variable range selection in combination with the instruction modification facilities. Specifying the expected range of variables will cause values outside that range to be highlighted in red. This can be used as a first line of defense in the investigation of errors. Once an error is identified, the code segment responsible for generating the incorrect result can be modified dynamically until the code is correct. The debugging process can then proceed from that point.

This is much more efficient than the standard debugging process of identifying errors through print statements, exiting the program, modifying the code, recompiling the program, and then re-executing the program from the beginning. To make matters worse, many programs will require substantial execution time before reaching the location of the error.

Figure Proc-13 and Proc-14 show an example of how this process might take place. Figure Proc-13a shows the generation of the random numbers used in a quicksort algorithm.
We halt execution before all the random numbers have been generated and change the equation used to generate the numbers. The equation is changed to be equal to the “counter” variable, the loop index. As can be seen in Figure Proc-13b, the generated values are shown in red, indicating they are out of range. This occurs because the expected range of values is from 0 to 255 but the loop index runs from 0 to 4095; the loop generates 4096 values. So this equation is clearly not what we desired.

![Fig. Proc-13: Example showing how the equation generating random numbers for a quicksort algorithm can be changed incorrectly. The incorrect values are shown in red.](image)

The incorrect equation shown in Proc-13b can be modified to correct the error. The equation should actually equal the “counter” variable modulus 256. The result after changing to this new equation is shown in Figure Proc-14a. This generates the continuously looping gradient you would expect when using a modulus operator. Finally, we can reset the algorithm to its original equation for the remainder of the execution, Figure Proc-14b.

The principles shown here can be applied to real programs in which the incorrect or unexpected results really occur within the program and the programmer then modifies appropriate equations until the program executes as expected. Since the programmer does not need to restart the program after changing the code, the debugging process is made much more efficient. The user can see the results caused by the code modifications instantaneously.
8.5 Methodology

We provide enormous capability for aiding users in analyzing and debugging concurrent programs. To use these capabilities effectively the user should keep a list of rules in mind when applying the techniques.

When individuals first begin programming the instructor gives them tasks in which they are domain experts. A common task is to write sorting algorithms; we are all experts in basic numerics. When we are experts in the domain with which we are working we can very quickly determine whether our results are correct or not and whether they are significant or not. When we are not domain experts it can be extremely difficult to determine whether the algorithm is actually correct. Test data sets along with expected results that a domain expert provides can reduce this difficulty but this is actually just another form of input from the domain expert. The actual presence of a domain expert to assist in the process can greatly speed up the process and improve the quality of the resulting algorithm.

A second issue is the overuse of visualization and steering techniques. While we provide many techniques, it is important that the user not attempt to discover everything
simultaneously. If the user attempts to use all the techniques simultaneously right from the start, the amount of information the environment provides will likely overload the user with information and the user will find it difficult to learn anything. The techniques are more effective when applied methodically to learn about the environment one step at a time. Applying techniques to verify different aspects of the environment one at a time is much easier procedurally for most users than attempting to discern everything simultaneously.

Other than the two rules just given the user may apply the techniques in any fashion the user finds beneficial. Many users may find it beneficial to follow the following order and guidelines for analyzing their environment. The user’s experience with the application, difficulty with the application, and goals may impact their use of these rules.

1. Examine results of the entire computation and compare with expected results
2. Examine results during computation to determine if generation of incorrect results can be observed
3. Verify boundary, border, and barrier integrity
4. Verify execution stack
5. Verify correct execution of loops and conditionals
6. Verify correct movement of data

It is up to the user of a particular application to apply the techniques in an appropriate fashion for the application under investigation and the problems inherent in that application. We can only provide suggestions for some of the possible uses; the environment is only actually limited by the user’s imagination and creativity.
9 CONTRIBUTIONS

In summary, this research resulted in the following contributions to the comprehension and debugging technology available for concurrent systems:

- Application of steering techniques to concurrent systems
- An environment integrating the ability to both test and debug programs
- An environment with the ability to save and restore investigations
- Simultaneous execution of multiple classes of algorithms
- Multiple windows representative of different views of the algorithm
- Customization of display updates
- Customization of display update rates
- Tagging and instrumentation modification on a window by window basis
- Real-time dynamic modification of data values
- Implementation of dummy arrays (e.g., incremental arrays) for parameter mappings
- Contagious approach to tagging
- Constant color and color scales for representation of tagged data
- Visual and symbolic representations of operations and execution stacks
- Visual representation of operation parameters
• Visual representations of control expressions
• Visual display of correlated information
• Techniques for MIMD and distributed systems
• Techniques for long running programs
• Execution rate controls
• Locality of reference models to provide interaction under various execution rates
• Continuous execution of the program during data modification
• Instantaneous vs. persistent data steering
• Maintenance of interactions and modifications during program restarts
• Dynamic modification of instructions

In addition, examples were provided in how the techniques in this research can be used to aid in the debugging process. Examples were provides on how the following tasks can be assisted:

• Removing artifacts caused by the display of intermediary results
• Removing artifacts caused by the animation of an algorithm
• Understanding the importance and effect of components of an algorithm
• Understanding data movement characteristics
• Verifying the integrity of software boundaries and barriers
• Identifying the data elements used in a computation
• Verifying a variety of initial environment configurations
• Verifying the execution of the algorithm with selective data elements
• Verifying the algorithm at some future stage
• Verifying the correctness of control structures
• Verifying the execution stack
10 CONCLUSION

A wide range of tools are available to assist in the understanding and debugging of serial programs, while the tools available for concurrent systems are still limited in their capability. The need for high performance computing continues to grow in certain fields, such as weather and climate predictions (i.e., effects of El Nino), as indicated by the grand challenges in high performance computing [Commu93]. One of the goals of these grand challenges is to improve the development and dissemination of high performance computing technology. Development of tools to aid in understanding and debugging these types of architectures is imperative to their use and dissemination.

We have extended the technology available for concurrent systems and provided new capabilities in several areas, particularly in the application of interactive computational steering to the debugging of concurrent applications. Most of our techniques follow the concepts of immediacy put forward by Ungar et al. [Ungar97]. Data which are related through time, distance, or semantics are closely related visually and can be accessed or even steered in relation to one another.

The VisCon environment was developed as a test bed for the new techniques investigated during the course of this research. It allows users to analyze and debug applications using visualization and steering techniques. While many of the techniques
provided are unique in and of themselves, their true benefit lies in their application to concurrent architectures, which previously has been lacking debugging support. To incorporate an application for analysis into the VisCon environment the application must be fully instrumented to identify the instruction or operations being performed at any given time. The environment architecture is effective at efficiently providing instrumentation interpretation, visualization of the meaning of these instructions, and user directed steering, all within a single environment.

In order for the environment to provide many of the techniques developed during this research we needed to solve several key engineering issues:

- How to design and implement instrumentation instructions?
- What information these instructions pass and to whom?
- Where to insert these into the application?

The fundamental issue was what information to provide and how. The solutions provided include:

- Dynamic insertion and removal of instrumentation instructions.
- Visualization of operations, which provides information on the operation, the value of the operands, and the resulting value.
- Visualization of control structures, which provides information on the current value of the control, the current values of the operands, the minimum and maximum values if appropriate, and the type of relation in the control structure.
- Operation modification. Here the instrumentation instructions replace the results of the original operation with a user specified equation, value, or operation.
- User specifiable execution rate (which also required resolution of the application delay issues discussed).
• The contagious approach to tagging, a new technique singly responsible for allowing data to be tagged and not merely processors.
• User directable performance modification providing the ability to specify that display updates be performed only 1 in N iterations and whether the environment should refer to this per instruction or for all instructions collectively.

The data visualization techniques incorporated into the VisCon environment are primarily geared towards providing a platform to test steering and operation visualization techniques. Even so, the range selection capabilities provided in conjunction with the data visualization techniques allow the user to highlight unexpected or out of range values and their location in the processor array or data array very clearly. The ability to tag data elements, even on a concurrent architecture, provides the ability to trace erroneous values back to their source and determine how the application is generating the erroneous values, which cannot be done by tagging processors. The inclusion of both solid and intensity tagging provides the ability to trace any data element no matter its value or visual representation.

The operation visualization techniques, novel in their own right are revolutionary for aiding the debugging of concurrent architectures. They are particularly effective since the visualization routines merge them into the data displays. This provides visual correlation between data and operations. The operations visualization techniques are designed such that the user can easily differentiate between different types of operations and even between different instantiations of the same operations. Since each operation is not removed from the display until it has completed execution we have an inherent ability to represent the execution stack.

A specialized form of the operation visualization relates to control structures for loops and conditionals. We presented visualization techniques, including iconographic techniques, for representing control expressions in all types of systems, including massively parallel
systems. The visualization techniques provide additional information with these types of visualizations to better aid in their understanding. This includes visual representations of the component values in the operation as well as the type of operation and the relationships in the operation.

We have investigated and developed techniques for the steering of both the data elements and the operations within the application under investigation. While data steering has been done previously in other contexts it has never been applied to concurrent architectures. The operation steering is novel in and of itself. We have also investigated unique applications of steering techniques for the debugging of concurrent programs. Finally, we differentiate between two types of data steering, instantaneous and persistent. The instantaneous modification of data that is the standard approach and the persistent modification of data that ensures that once modified to a specified value the VisCon environment will not allow a data element to change, which is novel to this research. The persistent data modification allows the user to set data to a specific value for the remainder of the execution and for future executions, allowing new environment configurations to be simulated dynamically.

With the novel techniques for visualization and steering techniques we developed it is important to understand how the user can apply them to the debugging process on concurrent architectures. To this end, we explored the types of visual conditions and patterns programmers should look for which may indicate that an error or problem is occurring within the application. The sooner the user locates a potential error and identifies its cause, the quicker the user can complete the debugging process as a whole. In summary, we looked at methods for applying the visualization and steering techniques to aid in the analysis and debugging process. These include:

- Monitoring the data movement characteristics of the application.
• Monitoring the behavior of various components of an application.
• Verifying border and boundary integrity.
• Verifying various algorithm configurations.
• Verifying data values.
• Debugging loops and conditionals.
• Debugging the execution stack.
• Verifying the algorithm’s impact.
• Providing correlation between the data display, the instruction stack, the current operation, and the status of loops and conditionals.

While developing the interfaces for the steering and visualization capabilities we went to great length to ensure that the interaction metaphor was principally instrumented by a direct manipulation philosophy. We also attempted to incorporate techniques that would harness previous experience the user may have to reduce the need to learn new interaction metaphors. We also provide feedback as to what interaction is occurring or how data is being modified to keep the user abreast of the current state of the interactions and system.

We have developed interactive techniques, totally within the completely instrumented environment, that support the analysis and exploration of concurrent program execution. We developed techniques such as dynamic modification of the instrumentation instructions and continuous interaction. Dynamic modification of the instrumentation instructions allows users to denote their own concept of what “visually” should be an atomic instruction. Through display update controls the user can prevent the updating of the display for routines that have already been verified correct or which too greatly interfere with the current focus of the analysis.
Ultimately, our goal is to allow the user to quickly identify points of interest within a program at which errors, faults, or trends are being generated by showing generated values of interest immediately rather than at the end of the program run and providing the user the tools needed to verify that the values are of interest. These techniques aid the analysis and debugging process and not merely program learning. Our techniques move away from program animation techniques that required specialized animations be developed for each application or algorithm. Our approach is independent of any specific program and works without special modifications to either the instrumentation or to the program itself.
11 FUTURE WORK

While the environment we have developed provides enormous capability for comprehending and debugging concurrent applications, there are many areas that need further development. We list some areas of particular interest below.

1. Many of our techniques are dependent on the selected region of interest to which the user applies tagging and interaction operations. The region selection must currently be a rectangular area. The techniques are directly scaleable to arbitrary regions. There will be an increase in overhead to deal with more complex regions but this should be readily handled by today’s architectures.

2. The persistent interaction techniques currently only set fixed values in the selected region. We are exploring the capability of having the region continuously incremented or decremented, and later to be set to some general function. While incorporating modified persistent interaction in itself is not difficult, incorporating an interaction metaphor that does not interfere with the current instantaneous and persistent interaction metaphors will be more difficult.

3. The user needs to control the method by which the transfer of tagged elements occurs. The user should be able to select a less contagious approach to the transfer of the tag elements, perhaps even only allowing parallel data movement commands to exchange
the tag data. The user would be able to more narrowly focus on specific types of interactions and network communications.

4. We have incorporated some preliminary visual representations of the operations that are executing in an algorithm. How does one provide a user with contextual information as to where in the application the program currently is without having to rely on textual representations that are serial in nature and greatly reduce the efficiency of the user?

5. Providing the actual source code of the executing application in a separate window is not new. However, it will provide an effective means for allowing the user to identify breakpoints, the exact location where to modify instrumentation instructions, and the exact operations to modify. This interface will be very familiar to users and makes it extremely useful. It should be added to any complete environment and should be applicable to all types of architectures.

6. We only provide very rudimentary visualization techniques, choosing instead to improve the interaction with those visualization techniques. The exploration and addition of further visualization techniques need to be investigated.

7. The SIMD architecture used for the initial development did not support C++, which consequently was unusable for development. A port of the environment to C++ will make use of operator overloading. This could potentially eliminate the need to insert instrumentation instructions directly into the program. This will result in a cleaner environment, easier incorporation of applications into the VisCon environment, and the ability to add easily greater functionality.

8. With the popularity of the Microsoft Windows platforms, the environment should be ported to Windows NT. This will increase the potential audience and increase the number of users who could potentially use the environment. This type of environment
should make use of the SMP (Symmetric Multi-Processing) capabilities provided with Windows NT as well as distributed environments. Windows NT can also be used in front-end systems for massively parallel architectures.
12 REFERENCES


[Commi93] Committee on Physical, Mathematical, and Engineering Sciences, Federal Coordinating Council for Science, Engineering, and Technology, Grand


[Levko87] H. Levkowitz and G. T. Herman, “GLHS: a generalized color model and its use for the representation of multiparameter medical images,” In M.A. Viergever...


13 REFERENCES USED BUT NOT CITED


14 WORLD WIDE WEB SITES


Appendix A - Greyscale Images

This appendix shows additional examples of some of the visualization techniques that have been applied to the visualization of programs. Examples are given of visualization applications designed for both serial and concurrent systems. The examples also apply to a wide range of design goals (e.g., behavioral analysis, application debugging, and performance tuning). These example images are provided as examples of the current extent to which visualization has been applied to the debugging and performance tuning of concurrent applications and of the novelty of our techniques. This appendix presents gray scale images.

**Fig. G-1:** Process activation grid [Fried91]. Used more often in SIMD architectures, this grid shows the active and inactive processes for a given computation. The darker squares represent inactive processes which will not take part in the computation.

**Fig. G-2:** Process data representation [Fried91]. This example show the values of a particular data item on the processors. The processors are represented as a 2D mesh corresponding to the hardware architecture. The data values are represented by intensity and height fields.
Fig. G-4: Process data representation [Fried91]. A second example of the data values residing on a processor array.

Fig. G-6: Process data representation [Fried91]. This is a third example of the data values residing on the processor array. Unlike the previous two examples, this is the result that was expected from the algorithm.

Fig. G-3: A communication graph from ParaGraph, presented in [Rafie93]. The processor nodes are shown on the horizontal axes and the current length of the processor’s message queue is shown on the vertical axes.

Fig. G-5: A processor utilization display from ParaGraph, presented in [Rafie93]. Time is presented on the horizontal axes, while number of processors is shown on the vertical axes. The different shading indicates the number of processors that are IDLE, BUSY, or executing OVERHEAD.
Fig. G-7: An insertion sort algorithm animated using BALSA, presented in [Rafie93]. In this image, the current position of the elements is shown on the horizontal axes, while the value is represented using the vertical axes.

Fig. G-9: Another view of an insertion sort algorithm taken from BALSA, presented in [Rafie93]. Each row shows the current progress of insertion.

Fig. G-10: This example of ParaGraph displays, taken from [Glend92] shows what a user’s workstation could typically look like when attempting to debug a program using ParaGraph. It shows the number of displays that can realistically be used simultaneously.

Fig. G-11: TRI Procedure Histogram from MTOOL [Goldb91]. Shows characteristics for three separate procedures and the proportionality of these characteristics.
Fig. G-12: Source window display from MTOOL [Goldb91]. Shows the application code and highlights the code currently being analyzed.

Fig. G-13: This example shows a more complex use of fisheye views [Sarka94]. Each image shows the same road network but with a different application of the fisheye view techniques. The top-left image is unmodified.

Fig. G-14: An example of the use of fisheye views, taken from [Sarka94]. This example shows a regular network in which one node of particular interest is highlighted and scaled in size while the surrounding nodes gradually fall off in size and receive less focus.
Fig. G-15: An example of quicksort being visualized using a modified scatter plot technique, developed using Polka-3D [Stask92a]. When the algorithm begins, the small diagonal line on the right side of the image consists of randomly scattered dots. Horizontal position is determined by position in the array while vertical position is determined by the data value. As the array sorts, the diagonal line shown here is formed. The planes to the left of the diagonal line perform a history function. When two elements are exchanged, the two data values which are exchanged are used to define the two corners of the plane. The planes start right behind the scatter plot.

Fig. G-17: A bubblesort being analyzed with an augmented 2D bar chart, developed using Polka-3D [Stask92a]. The bar chart is calculated normally, with the height of each bar representing the value of the data value at the corresponding position. A constant depth is then added to simulate a 3D visualization.

Fig. G-16: An example visualization generated using Maritxu [Zabal92a]. The top and bottom sections represent different views. Each square (icon) represents an individual processor. The different features of each icon represent different resources. The intensity of the gray level of a feature represents the value of the corresponding resource.

Fig. G-18: An example of the second version of VIPS being used to analyze linked-list structures [Shimo91]. The display on the right side of the image is a representation of the entire linked list structure currently being examined. Notice the highlighted region. This highlighted region is zoomed in on in the upper left window. This view also has a selected region, which is shown in the bottom left window. This final view shows much more detail, including the values of the data structures in each of the nodes.

Fig. G-19: An example of a 3D particle simulation is visualized in this example, developed using Polka-3D [Stask92a]. Each object's shape and color represents characteristics about the particle.
Fig. G-21: A standard Kiviat diagram, designed using POLKA [Stask92b]. Different colors are used to represent the different regions. The darker shade represents the average of processor utilization while the lighter shade represents the current usage.

Fig. G-23: An example taken from the first version of VIPS [Isoda87]. The bottom right window shows the code fragment currently being executed. The upper left window shows the data structure being accessed. Modifications to the data structure are shown dynamically. The bottom left window shows a history of the executed commands.

Fig. G-20: A Kiviat diagram, taken from ParaGraph, is shown here [Tomas94]. While examples of Kiviat diagrams have already been given, this example shows very clearly the ability of patterns or textures to differentiate between regions of an image. Thus, while color is useful it is not always a necessity.

Fig. G-22: A visualization example of quicksort developed using BALSA [Brown91b]. The image on the right shows a scatter plot representation of the animation. The position of the data value in the array controls the position of the pixel on the horizontal while the data value itself controls placement on the vertical axes. The changes in color are representative of the different threads executing different portions of the sort. The left image represents the partitioning performed during the sort.

Fig. G-24: A Feynman diagram taken from ParaGraph, presented in [Kraem92]. This type of display shows communication patterns among processes over time. The horizontal axes represents time while the vertical axes represents the processor. Communication is represented by a line connecting the sender and receiver.
Fig. G-26: Utilization count display, taken from ParaGraph [Tomas94]. Shows the number of processors currently busy, the number of processors burdened with overhead processing, and the number of processors which are idle.

Fig. G-27: Concurrency profile, taken from ParaGraph [Tomas94]. Shows the percent of time that a given number of processors are active.

Fig. G-28: Communication traffic display, taken from ParaGraph [Tomas94]. Shows the number of bytes being transferred over the processor interconnection network during the execution of the program.

Fig. G-29: Utilization meter, taken from ParaGraph [Tomas94]. Shows the percent of processing power currently being used by the system or the average used during the course of execution of the program as a whole.

Fig. G-25: Utilization summary, taken from ParaGraph [Tomas94]. Shows the percentage of processing power used by each processor at any given point in time or averaged over the entire execution time.

Fig. G-30: Streak display, taken from ParaGraph [Tomas94]. Shows the current execution streak for each processor. Periods of busy activity are shown by a continuously growing bar above the horizontal axis. Periods of inactivity are shown by a bar growing down from the horizontal axis.
Fig. G-32: Space time diagram, taken from ParaGraph [Tomas94]. Shows the activity of each processor, active or idle, as well as inter-processor communication—lines between processors. Color may be used to indicate message characteristics.

Fig. G-34: Message queue display, taken from ParaGraph [Tomas94]. Shows the length of each processors message queue at any point in time or averaged over the entire execution period. The darker shade indicates the current size while the lighter shade indicates the maximum size previously achieved.

Fig. G-31: Communication matrix, taken from ParaGraph [Tomas94]. Indicates the sending and receiving nodes of a message by an entry in the matrix. Message characteristics can be represented by the color or other characteristics of the entry.

Fig. G-33: Task count display, taken from ParaGraph [Tomas94]. Shows the number of processors executing a given task at any point in time or the average over the course of execution.

Fig. G-35: Task Gantt chart, taken from ParaGraph [Tomas94]. Indicates the current task each processor was executing at any point in time. The processors are on the vertical axis and time is on the horizontal axis. The colored bar represents the task.
Fig. G-36: Task status display, taken from ParaGraph [Tomas94]. Shows each task in a matrix. When the task starts, it is lightly colored. As it executes its representative square is slowly darkened. At completion its square is black.

Fig. G-37: Task summary display, taken from ParaGraph [Tomas94]. Indicates percentage of execution time of a task in relation to execution time of the entire application.

Fig. G-38: Phase portrait display, taken from ParaGraph [Tomas94]. Shows the relationship between processor utilization and communication bandwidth used.

Fig. G-39: Example output from Intel's IPD debugger, given in [Panc92a]. This example shows some of the textual output generated by text based parallel debuggers.
Fig. G-41: Example output from IBM's Parallel FORTRAN Trace Facility, given in [Panca92a]. A second example of textual output generated by text based parallel debuggers.

Fig. G-42: An example display from PF-View, presented in [Panca92a]. Each processor is represented by a circle. The processor's status is represented by a symbol in the circle. The middle processor above is waiting for a lock on shared memory.

Fig. G-43: An example taken from SHMAP, presented in [Panca92a]. Accesses to a shared matrix are represented with the accessing process's representative color. Read accesses are shown on the left while write accesses are shown on the right. Each matrix element is represented by a grid square.

Fig. G-44: An example display from PF-View, presented in [Panca92a]. This tool uses different flow chart constructs to differentiate between serial and parallel code. The exact symbol used in the flow chart is indicative of the type of code being executed.

Fig. G-45: An example taken from Matvu, presented in [Panca92a]. This example shows the data values of the matrix using a logarithmically scaled color table.

Fig. G-46: An example taken from Belvedere, presented in [Panca92a]. Communication among the nodes is shown using highlighted arrows along the inter-processor connections.
Fig. G-47: Dynamic application state diagram [Glenn91]. An example of an application's flow. Instruments placed at important points in the application's flow present instructive information on the progress and efficiency of the application.

Fig. G-48: Dynamic network diagram [Glenn91]. Each node is represented by a grid square. Inter-node communications channels for each node are represented by circles within the node. The rate of conflicts on each channel is represented by the color of the circle.

Fig. G-49: Textual output from the C* Data Visualizer [Jourd90]. The dbx session can be seen in the background. The values of the selected variable are shown in the foreground.
Fig. G-50: Visual display of data values using the C* Data Visualizer [Jourd90]. These values are unfiltered. The intensity of the value represents its value.

Fig. G-53: Visual display of data values using the C* Data Visualizer [Jourd90]. The intensity of the value represents its value. The values have been filtered to select a much smaller range of values. This new range was then re-grayscaled to recompute the intensities of the pixels.

Fig. G-51: Zoomed visual display of data values using the C* Data Visualizer [Jourd90]. The display has been zoomed in or scaled to make the individual processor values more readily apparent.

Fig. G-52: Simple bar graph taken from [LeBla90]. This example shows the fraction of execution time that the target application achieved specified amounts of parallelism.

Fig. G-54: A trace example taken from [LeBla90]. This example shows the amount of parallelism achieved during the execution of the program, as a function of time.
Fig. G-55: An example taken from [LeBla90] which shows the averaged trace of the utilized parallelism shown in Figure G-55.

Fig. G-57: A simple scatter plot taken from [LeBl90]. This figure shows the ratio of communication vs. computation.

Fig. G-58: A second scatter plot taken from [LeBl90]. This example shows the amount of communication time in each iteration of a loop for one process.

Fig. G-59: Glyph example taken from [Zabal92a]. Processing is taking place on the nodes.

Fig. G-60: An elevation map of the communication data shown in figure G-60 [LeBl90]. All processes and all loop iterations are shown in this single display.
Communication is taking place across the east-west interconnection network links.

Fig. G-62: A process execution graph from Parasoft’s PM and provided in [Nicho90]. The process each node in a parallel system executed at every point in time is shown as well as the actual activity that was being performed by that process.

Fig. G-63: Glyph example taken from [Zabal92a]. Shows two 4x4 sub-networks. Each is running a different code segment and uses a different glyph.

Fig. G-64: A display from the University of Illinois’s Hyperview, provided in [Nicho90]. This example shows many of the statistical displays that are provided by Hyperview.
**Fig. G-65**: Simple pie charts taken from [Calza95] which show different aspects of communication issues. The top charts show the total communication bandwidth as allocated to different types of messages. The bottom charts show the percentage of the number of calls for each type of message.

**Fig. G-67**: Simple bar chart taken from [Calza95]. This chart shows the number of messages occurring with a specified length.

**Fig. G-68**: Example animation of a packet routing simulation taken from Zeus [Brown94]. The animation shows the physical network connections, the physical nodes, the location of actual packets (squares), and the volume of traffic over each connection.

**Fig. G-69**: Example animation of Bresenham’s line-drawing algorithm taken from Zeus [Brown94]. Three different levels of detail are shown. The actual line in the middle, a large scale view of the line in the bottom right, and a detailed view of the current pixel in the bottom left. The arrows in the bottom left represent local variables used to calculate the position of the next pixel.

**Fig. G-66**: Example animation of Breseham’s line-drawing algorithm taken from Zeus [Brown94]. Three different levels of detail are shown. The actual line in the middle, a large scale view of the line in the bottom right, and a detailed view of the current pixel in the bottom left. The arrows in the bottom left represent local variables used to calculate the position of the next pixel.

**Fig. G-69**: This example of views used by PARADE shows particular attention to message sends and received [Stask95]. A variety of different techniques are shown here.
Fig. G-70: This example taken from PARADE shows a visualization of high performance fortran programs [Stask95]. Arrays are illustrated here with color used to identify different processors.
Appendix B - Color Images

This appendix shows additional examples of some of the visualization techniques that have been applied to the visualization of programs. Examples are given for visualization tools designed for both serial and concurrent systems. The examples also apply to a wide range of design goals (e.g., behavioral analysis, application debugging, and performance tuning). These example images are provided as examples of the current extent to which visualization has been applied to the debugging and performance tuning of concurrent applications and of the novelty of our This appendix presents color images.

Fig. C-2: An example taken from the View Kernel (VK) portion of AIMS [WWW-4]. This example provides an overview of the I/O resources being used. Value is encoded in the height of the bar, duration in the length of the bar, and process identification in the color of the bar.

Fig. C-1: A second example taken from the View Kernel (VK) portion of AIMS [WWW-4]. This example also provides an overview of the I/O resources being used. Instead of using the height of the bar to indicate value, this example uses different faces as a representative factor. The two examples could actually be combined to provide several variables in a single display.
Fig. C-4: An example Xish session [WWW-4]. This application is geared towards performance analysis and provides information on the CPU requirements of individual procedures as well as the percentage of the total execution time that each processor provides. This can be used to identify where performance tuning will provide the greatest benefit.

Fig. C-7: A sample visualization of processor workload, developed using POLKA [WWW-1]. Processor number is on the horizontal axes while workload is on the vertical axes. Each processor is also identified by a unique color.

Fig. C-3: A Kiviat diagram showing processor utilization, developed using POLKA [WWW-1].

Fig. C-5: A Kiviat diagram, shown using color to differentiate between processor ID's and spoke length to indicate utilization at a particular point in time [Hacks95].

Fig. C-6: A 3D Kiviat tube, using the same representation as in the color Kiviat diagram [Hacks95]. This version has the advantage in that historical data is not lost. As time progresses, new utilization values for each processor are placed in increasing Z values.

Fig. C-8: A modification to the 3D Kiviat tube [Hacks95]. In the standard 3D Kiviat tube it is difficult to differentiate between the individual time slices or see the values of an entire slice simultaneously. This version is modified such that the tube is semitransparent except for a single slice that is being investigated.
Fig. C-10: Isosurface representation of local data accesses, as described in [Hacks95]. Five isosurfaces are used here to represent local data access patterns.

Fig. C-12: An animation of a quicksort algorithm in progress, developed using POLKA [WWW-1]. The array index lies along the horizontal axes while the associated data value is shown on the vertical axes. The color of the associated bar is representative of the thread sorting the particular chunk of data.

Fig. C-13: A visualization of a quicksort algorithm, developed using POLKA [WWW-1]. This view represents a history of the swaps that have taken place so far during the progress of the sort.

Fig. C-9: An example visualization of a Pthread program, implemented in the GThreads view library, is shown here [WWW-1]. The small inset window in front shows locks attempting to gain or already holding mutex locks. The smaller circles are the threads and the larger circle is the mutex. When the thread gains the lock it moves inside the mutex circle. The larger window in the center, behind the mutex lock window, shows the program call graph. The window far to the left shows the threads executing in the program. The state of the thread is indicated by the coloring of the icon. The window to the top right shows the execution history of each thread. The final window to the bottom right shows barrier synchronizations.

Fig. C-11: Isosurface representations of remote data accesses, as described in [Hacks95]. Two time steps are shown, each with four isosurfaces, of the frequency of accesses to remote elements of distributed data.
Fig. C-14: Another example of quicksort, being visualized using a modified scatter plot technique, developed using Polka-3D [WWW-1]. When the algorithm begins, the small diagonal line on the right side of the image consists of randomly scattered dots. Horizontal position is determined by position in the array while vertical position is determined by the data value. As the array sorts the diagonal line shown here is formed. The planes to the left of the diagonal line perform a history function. When two elements are exchanged, the two data values that are exchanged are used to define the two corners of the plane. The planes start right behind the scatter plot. This example differs from the previous one in that it uses color rather than gray scale and shows that the resulting display can be rotated to arbitrary display angles.

Fig. C-17: An example developed using Zeus [WWW-2]. This image shows a three dimensional view of a two dimensional tree.

Fig. C-15: A view of a quicksort animation developed with Zeus [WWW-2]. This view shows the blocks being computed by the individual threads. Each thread is designated by a unique color.

Fig. C-18: Another view of a quicksort algorithm, this one developed using Pavane [WWW-3]. In this view, the processor id is encoded on the horizontal axes and the height of the individual bars encodes the value at the associated processor. The color of the bar indicates the thread that is currently sorting the associated value.
**Fig. C-19:** A communication graph, developed using Polka [WWW-1]. The processor id’s are encoded in the vertical axes and time on the horizontal axes. A line is drawn between processors whenever a communication has been received. The line shows where the message originated (start point) and where it was received (end point).

**Fig. C-20:** A second example of inter-processor taken from [Saruk95]. In this example a large white column can be seen in which the system is poorly utilized.

**Fig. C-21:** The animation of a sort algorithm, taken from [WWW-1]. The bars are positioned along the horizontal according to their position in the array. The height of the bar indicates the size of the corresponding data value. The two elements being exchanged have been pulled out of alignment.

**Fig. C-22:** An inter-processor communication graph generated with Paragraph and taken from [Heath95b]. In this example the processor nodes are arranged in an arbitrary circle and communication between processors at a particular moment is indicated by lines between the sending and receiving processor. Node color indicates the state of the processor.

**Fig. C-23:** An example taken from [Saruk95]. The inter-processor communication pattern is shown by the lines interconnecting the different nodes. The starting and ending points of the lines show the when a message was sent and received respectively. The colored blocks represent CPU utilization.

**Fig. C-24:** A second inter-processor communication graph also generated with Paragraph and taken from [Heath95b]. In this example the nodes are arranged in their physical arrangement and the color of the communication line indicates whether or not the communication is along a physical communication link.
Fig. C-25: A third example of inter-processor communication generated with Paragraph and taken from [Heath95b]. This example shows the actual path of each communication. Messages start on the left and end on the right.

Fig. C-28: A representation of utilization statistics generated with AVS and taken from [Heath95b]. The entire processor array is shown in this image (16,384 processors). Color represents the processor utilization.

Fig. C-24: A representation of utilization statistics generated with Seeplex and taken from [Heath95b]. Sixty four processors are represented.

Fig. C-26: These examples were generated with Seeplex and taken from [Heath95b]. The top image is a 2D scatter plot of processor utilization. The bottom image is a stacked-bar histogram representing the coloring of the top image.

Fig. C-27: A representation of data array accesses taken from [Heath95b]. Each color represents a processor node. The four large rectangles are the processor nodes. The smaller black rectangles are the distributed variables. The colored squares are the portions of the distributed variable allocated on the individual processor nodes. A data access is represented by a black region on the receiving node and a white region on the sending node.
Fig. C-30: An example of data accesses generated with IBM Data Explorer and taken from [Heath95b]. This example shows the local accesses received in a distributed data array. The higher the point the more local accesses the node has received.

Fig. C-32: An example of kaleidoscope visualization of a program's call graph taken from [Heath95b]. The call graph for the program's entire execution is represented in this single display. Color represents time of invocation.

Fig. C-34: A second example of kaleidoscope visualization taken from [Heath95b]. This image has been modified from figure C-30 to include a 3D characteristics, time is mapped redundantly to elevation.

Fig. C-29: A third example of kaleidoscope visualization taken from [Heath95b]. This example is similar to figure C-32 except color is used to represent the processors.

Fig. C-31: An example visualization of task execution history generated with Paragraph and taken from [Heath95b]. This image shows a task Gantt chart of 64 processors.

Fig. C-33: An example visualization of task execution history generated with Paragraph and taken from [Heath95b]. This image shows a utilization Gantt chart.
Fig. C-35: An example visualization of task execution history generated with Paragraph and taken from [Heath95b]. A task Gantt chart of 8 processors.

Fig. C-37: An example of a Kiviat tube generated with IBM Data Explorer and taken from [Heath95b]. Kiviat tubes show the utilization of all processors during the entire execution history of a program.

Fig. C-36: An example of a Kiviat tube generated with IBM Data Explorer and taken from [Heath95b]. Kiviat tubes show the utilization of all processors during the entire execution history of a program.

Fig. C-38: A utilization Gantt chart generated from Paragraph and taken from [Heath95a].

Fig. C-39: A utilization summary histogram generated with Paragraph and taken from [Heath95a].
Fig. C-42: A utilization count display taken from [Heath95a]. The number of processors in each of the busy, overhead, and wait states are represented.

Fig. C-44: A space-time diagram generated with Paragraph and taken from [Heath95a]. This image shows the inter-processor communication patterns of the program execution.

Fig. C-41: A processor status display taken from [Heath95a]. This display consists of four sub-images showing the processor status, volume of messages being sent, volume of messages being received, and unused, in counterclockwise order from the upper-left image.

Fig. C-43: A scatter plot of transaction execution taken from [Jakie95]. Transaction start time is mapped to the y-axis while completion time is mapped to the x-axes. Color represents the processor node.

Fig. C-45: Example taken from Annai’s Distributed Data Visualizer [Cleme96]. This image shows the data map visualizer. The data is a matrix and the processors are distinguished by colors.
Fig. C-46: Example taken from Annai’s Distributed Data Visualizer [Cleme96]. This example shows a tabular representation of the actual data values.

Fig. C-47: Example taken from Annai’s Distributed Data Visualizer [Cleme96]. This image shows the data map visualizer. The data is a matrix and the processors are distinguished by colors. The height of the bars represents the value of the data element. Multiple iterations are shown.

Fig. C-48: Example taken from Annai’s Distributed Data Visualizer [Cleme96]. This image shows the data map visualizer. The data is a matrix and the processors are distinguished by colors. The height of the bars represents the value of the data element.

Fig. C-49: Example taken from Annai’s Execution Statistics Display [Cleme96]. Shows execution time of various routines and time spent performing other tasks for the routine.
Fig. C-51: Example taken from Annai’s Execution Statistics Display [Cleme96]. Shows execution time of various routines and time spent performing other tasks for the routine.

Fig. C-52: Example taken from Annai’s Execution Statistics Display [Cleme96]. Shows execution time of various routines and time spent performing other tasks for the routine.

Fig. C-53: Example taken from Annai’s Execution Statistics Display [Cleme96]. Shows execution time of various routines and time spent performing other tasks for the routine.

Fig. C-54: Example taken from Annai’s Execution Statistics Display [Cleme96]. Shows the amount of data being sent by various routines and the purpose of sending that data.

Fig. C-55: Example taken from Annai’s Processor Balance Display [Cleme96]. Shows the amount of execution time and communicated data for each processor.
**Fig. C-57**: Example taken from Annai’s Evolution Time-line Display [Cleme96]. Shows processor interaction summaries and memory utilization graphs. High level view.

**Fig. C-58**: Example taken from Annai’s Evolution Time-line Display [Cleme96]. Shows processor interaction summaries and memory utilization graphs. Low level view, showing individual message transfers.

**Fig. C-59**: Example taken from Annai’s Processor Balance Display [Cleme96]. Shows execution time and time spent on other process related activities/overhead. Examples for 64 and 128 processors are shown.
Fig. C-60: Example taken from Annai’s Program Structure Browser [Cleme96]. Shows the programs structure (not the actual source code). This provides a high level description of what is currently executing. Routines and loop blocks may be folded or unfolded to provide control over the detail seen.

Fig. C-62: A 3D surface example taken from Annai [Cleme96b]. Processor distribution is represented by color. The image shows a matrix with the values of each element represented by the height of the surface at that point.

Fig. C-61: This example from ParSee shows the parallelization of a program [Prest96]. Areas with lighter shades achieved better parallelization. The hue indicates potential reasons why only poor parallelism was achieved.

Fig. C-63: This example from ParSee shows the parallelization of a program [Prest96]. Areas with lighter shades achieved better parallelization. The hue indicates potential reasons why only poor parallelism was achieved. In addition, this example has been zoomed to a particular period of time to better highlight the details during that selected time period.
Dr. Robert Erbacher received the Doctor of Science degree in computer science from the University of Massachusetts at Lowell in 1998. He was a part of Umass-Lowell’s Institute for Visualization and Perception Research from 1990 to 1998 working on visualization and perception issues. From 1994 until 1997 he worked part time with the MITRE Corporation’s Virtual Reality group on visual simulation projects. He has been co-chair of the SPIE Conference on Visual Data Exploration and Analysis since 1995 and has been heavily involved with the IEEE Visualization conference since 1997. His interests include visualization, perception, computer graphics, and virtual reality.