Scalable Visualization of Parallel Systems*

Jorge García
Richard Hughey

UCSC-CRL-94-29
31 August 1994

Baskin Center for
Computer Engineering & Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064 USA

ABSTRACT

As parallel systems become larger and more complex, they become harder to understand. The vast amount of information produced by these systems can quickly overwhelm a person. In many cases, researchers have turned to the use of visualization techniques as an aid to understanding the inner workings of parallel systems. By converting the information to images, a visualization system can take advantage of the human ability to recognize patterns that would not seem obvious otherwise.

There are several existing systems for visualizing parallel programs. But most of them present fixed views of data, and will not scale to the thousands of processors available on massively parallel computers. In this paper we present a proposal for research in developing visualizations that scale with the number of processors. The methods used to create these views will make visualization systems useful in understanding computers with any number of processors.

Keywords: Parallel Computers, Visualization, Parallel Debuggers, Performance Evaluation, Animation, Scalability

---

*This work was supported in part by a NASA Graduate Student Research Program Fellowship.

Contents

1 Introduction 2
2 Overview 2
2.1 Requirements for visualization of parallel systems 3
3 Previous work 5
3.1 Performance evaluation tools 6
3.2 Debuggers 9
3.3 Program visualization tools 13
3.4 Towards scalable visualizations 17
4 Need for scalable visualization 18
4.1 Tool philosophy 19
4.2 Better analysis methods 20
4.3 Ties to program 20
4.4 Other issues 22
5 Comments and conclusion 23
References 23

1 Introduction

Parallel systems were created to shorten the time needed to solve some problems. The idea was that if one processor could solve a problem in time t, then two processors should be able to solve it in time $t/2$, and p processors should solve it in time t/p. But, in reality, getting the speedups was not that simple. Many problems have sequential restrictions, where the addition of more processors does not help. But even for problems where the workload can be shared, getting good speedup depends on using the available machine resources efficiently, keeping track of large amounts of data and having a good understanding of the parallel algorithm. Programmers have to master all of these areas in order to create good parallel programs.

In many cases, programmers have turned to visualization methods to help them understand the inner workings of parallel systems. Using program visualization and algorithm animation tools, they can get a better understanding of algorithms suitable for parallel computers. They can use parallel debuggers to produce programs that work correctly, and they can use performance evaluation tools to make the programs execute more efficiently.

Many of the problems associated with parallel programming are exacerbated by the increase in the number of processors in modern massively parallel systems. While the original parallel computers consisted of only 4 to 16 processors, newer computers can have thousands of processors. A programmer of massively parallel systems must orchestrate the execution of all the processors and keep track of the even larger amounts of data while making sure that all the available resources are used. Good visualizations are needed more than ever!

Unfortunately, most of the visualizations produced by the available systems are not scalable: as the systems increase in size, the visualizations become obsolete. There is the problem of screen area, which is limited, and number of things to view, which keeps increasing. Most systems have an upper limit on the number of processes they can view, and they are rapidly approaching this limit.

To be able to use visualization systems with massively parallel computers, we must develop views that scale with the number of processors. These views must present global information about the system, but should also present more detailed local information when necessary. To produce scalable visualizations, we need better ways of doing data collection, analysis, storage and display. In this paper we concentrate on techniques for data analysis and display.

In the next section, we will give an overview of the issues involved in parallel system visualization as related to the goals of the visualization. In section 3, we will review many available tools, indicating their goal and how they handled some of the issues discussed. In section 4, we will look towards the future and the areas where more research is needed. Section 5 will present some final comments and conclusions.

2 Overview

The amount of data produced by parallel systems can be overwhelming. Sifting through the data is tedious and likely to lead to erroneous conclusions. This makes parallel systems a perfect candidate for the use of visualization. Visualization is a way of taking advantage of the advanced cognitive capabilities of the user to pin-point irregularities in the data.

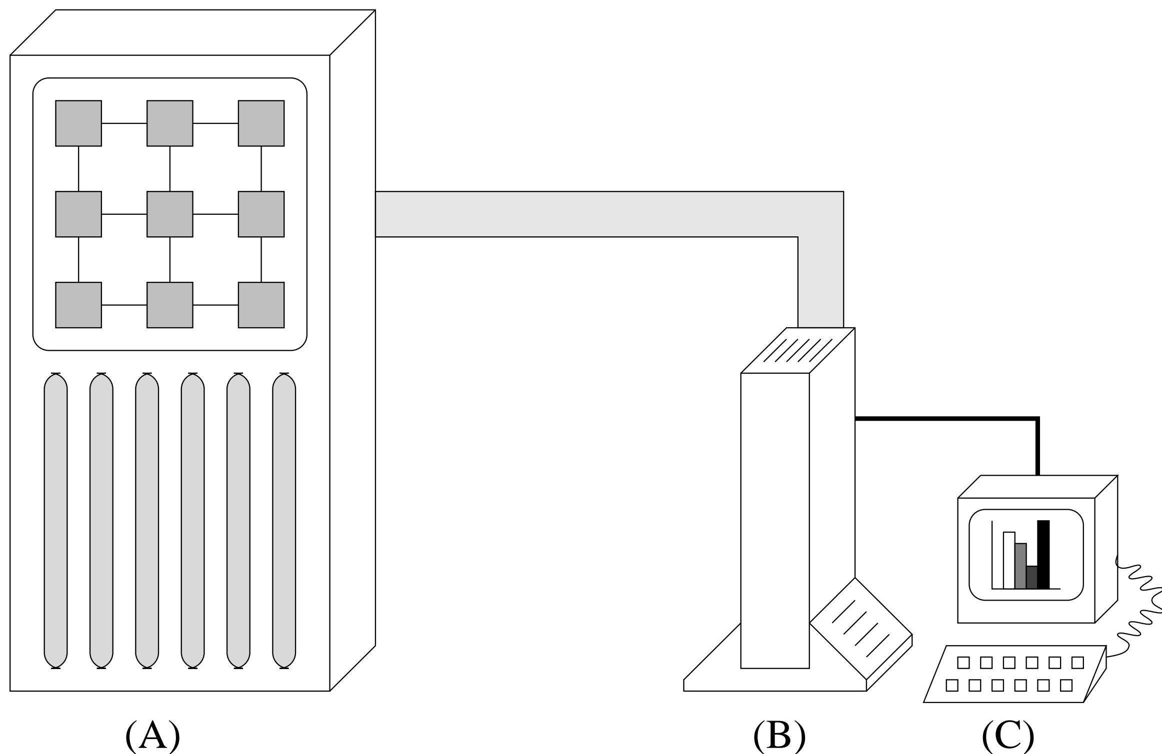

Figure 2.1: A typical setup for visualization of parallel systems. The parallel computer (A) will run the program we want to study and produce the data that we want to visualize. This data is transmitted to the workstation (B), where it is stored as a file. When the visualization system is run on the workstation, it analyzes the trace file and displays the selected information on the screen (C).

The purpose of the visualization is the most important criteria for deciding which data to gather from the parallel system. For example, figuring out if most processors are being used efficiently is significantly different than figuring out if the current program really implements a correct sort algorithm. Each type of visualization requires different kinds of data and displays the data in different ways.

In this section we will discuss some of the issues that need to be considered for the creation of visualizations of parallel systems. We will present a description of the tasks required to produce a visualization, then we will create a list of issues that can be associated with each task. In the following section, we will relate the importance of each of these issues to the goal of the visualization system, and see how some of the available systems have dealt with these issues.

2.1 Requirements for visualization of parallel systems

The creation of views involves several tasks that can be implemented as separate modules of a visualization system [KS92]. These tasks can be classified as collection, storage, analysis and display of the data generated by the parallel system. Figure 2.1 shows the typical setup for visualization of parallel systems.

There are different issues involved in each of these tasks:

Data collection Data collection is done by recording information every time some "interesting" event happens during the execution of the parallel program. Some of the important questions that should be asked about data collection are:

1. Which data should be collected?

   The data to be collected should be directly related to the purpose of the visualization. The user of the visualization should have some idea of how the data is supposed to look and use the visualizations to contrast the actual data with the expected behavior.

2. Is the data collection done by software or hardware?

   Collection by hardware has the advantage that it causes little perturbance in the program execution. Collection by software is slower and causes more perturbance, but it is more flexible and portable.

3. How much perturbation is created by the data collection?

   If the data is collected by software instrumentation, the program may be slowed down considerably. It may cause the program to execute differently when it is being monitored.

Data storage Data storage may become a problem because of the large amounts of data that can be generated from a traced execution. The main questions are:

1. Should the data be stored in a trace file or piped to the visualization system?

   The amount of data generated by the system can be extremely large, so it may be necessary to have a lot of available disk space if it is going to be stored into a file.

2. Should the trace be analyzed before storing the data?

   Storing analyzed data can reduce our disk space requirements, but eliminates the ability to examine the raw data that was generated by the system.

3. What amount of data will need to be stored? Should the data be compressed?

   The size of the trace files can grow very rapidly, and disk space can become a concern for long traces.

Data analysis Data analysis is any manipulation done to the raw data generated by the parallel system. Analysis can be done before storing the data or when reading the data into the visualization system. Some of the issues are:

1. What kind of information are we looking for in the data?

   We need to determine what we are interested in visualizing, and look for the data that is related to this information.

2. How do we separate useful information from the rest of the trace?

   When looking through the large amount of data, we must have some sort of filtering mechanism to extract only the needed information without wasting time processing useless information.

3. How much analysis should be done by the program and how much by the user?

   The advantage of visualization is that it uses the cognitive abilities of the programmer to detect patterns. If the visualization system does too much data analysis, it may obscure important details that the programmer would have noticed.

Data display Data display is the actual heart of the visualization. It involves presenting the data in a form that makes it more intuitive to the user of the visualization system. The important questions about displaying data are:

1. What is the most intuitive way to display the data?

   Most scientific visualization systems have the advantage of modeling a physical structure or occurrence, so the users already have a mental model of what the data should look like. In visualizing parallel systems, there are no obvious physical models, so the most intuitive way to display the data depends on the data being displayed, and on who is looking at the data. Different people will have different preferences on ways to display data, so we should offer many alternatives.

2. How do we highlight the important information?

   There is often so much happening at the same time that the visualization system must highlight some information that it deems important.

3. How much abstraction should we use to display the data?

   Sometimes abstractions help the user get a better understanding of what is going on, but sometimes they just obscure the information. Abstractions are also highly dependent on the user and the context, and an abstraction that makes something very clear for someone may just completely miss the point for someone else.

4. How much will the user be able to interact with the display?

   Some visualization systems only display the data, or give the user very little ability to decide on different ways to look at the data. Allowing the user to interact with the data can enhance the flexibility of the system, but also increases its complexity.

5. How does the display scale with the number of processors? Will a display give the same information if the system is using 200 processors instead of 4?

   Some displays lose information or become useless if too many processors are being visualized. Because of the limited screen size, visualization systems need to decide how to merge data for display without losing important details.

6. How much work is involved in creating a view?

   Some views are created automatically by the visualization system, and some views are completely user generated, which means more work for the user.

In the next section, we will look at some of the available tools for visualization of parallel systems, and we will see how they handled the issues just presented.

3 Previous work

There has been a lot of work done in the field of visualization of parallel programs. Most of the proposed tools can be classified by the view they present of the system being visualized:

- *Debuggers* present the information from the system's point of view. Visualization use in debuggers is mostly for looking at data movement, making sure events happen at the right time. To display the data movement, the views must match the hardware model being used (ie. SIMD, MIMD, hypercube, mesh, etc.). The other use of visualizations by debuggers is to present data that is distributed among the processors.
- *Program visualizations* present the information from the programmer's point of view with the purpose of matching the actual execution to the programmer's mental model of how the execution should proceed.

- *Performance monitoring systems* connect the hardware model to the programmer's implementation, indicating how well they match. They display information related to speed of program, use of available resources and efficiency.

These classifications are not mutually exclusive. In fact, many of the tools currently available cross over into more than one of these areas. But the goal of the visualization is always distinguished from the extra features available with each tool.

To help us look at the existing visualization tools, we try to classify each tool into one of the three categories described above. We will present the distinguishing features of each category, including how their goal affects the perceived importance of the issues. Then we will list some of the important systems from each of the categories. At the end of this section we will take a look at some early attempts to create scalable visualizations, and the systems that presented them.

3.1 Performance evaluation tools

Parallel computers were designed to increase the speed of executing programs, so it is no surprise that performance evaluation tools are important. They can assist programmers in finding bottlenecks or areas of the program where the performance can be improved. For an in-depth discussion of the issues involved in the creation of performance evaluation tools for parallel systems, the reader is directed to the two conference proceedings on the subject [SKB89, SK90].

Performance evaluation tools do not need to know the functionality of the program being executed. They just give information about the usage of the architecture or the balancing of loads. This means that the same displays can be reused with the execution of different programs. Most performance views are based on standard graphical displays, such as charts, X-Y plots, gauges and diagrams.

Tools in this category have also been used for performance debugging. This is useful when a program is producing the right result but is not executing as quickly as expected.

Data collection is one of the critical issues for a performance evaluation tool. Hardware monitors would be ideal for gathering performance information, since we know which data we need and we want to get it without disturbing the execution. Unfortunately, hardware monitors are not generally available. Software monitors are more flexible, but they are slow, and may cause performance degradation. In general, the more detailed information we want from a program, the less reliable the performance data will be because getting the information will affect the execution of the program.

There are two alternatives to data storage: direct recording in trace files or on-the-fly analysis and compression. The collected data is usually stored in trace files immediately because analyzing it on-the-fly could lead to serious performance degradation. Since the amount of data generated is large, efficient storage becomes an important issue. Storing all the generated data allows the user to visualize different aspects of the execution.

The analysis consists of extracting the data needed for the desired visualization and finding information such as averages and peaks. We also look at how long events take to complete, and the time processors spend doing useful work, doing overhead work or waiting for something to happen. Note that the time needed for performance evaluation is the physical time related to timestamps, so clock synchronization becomes very important.

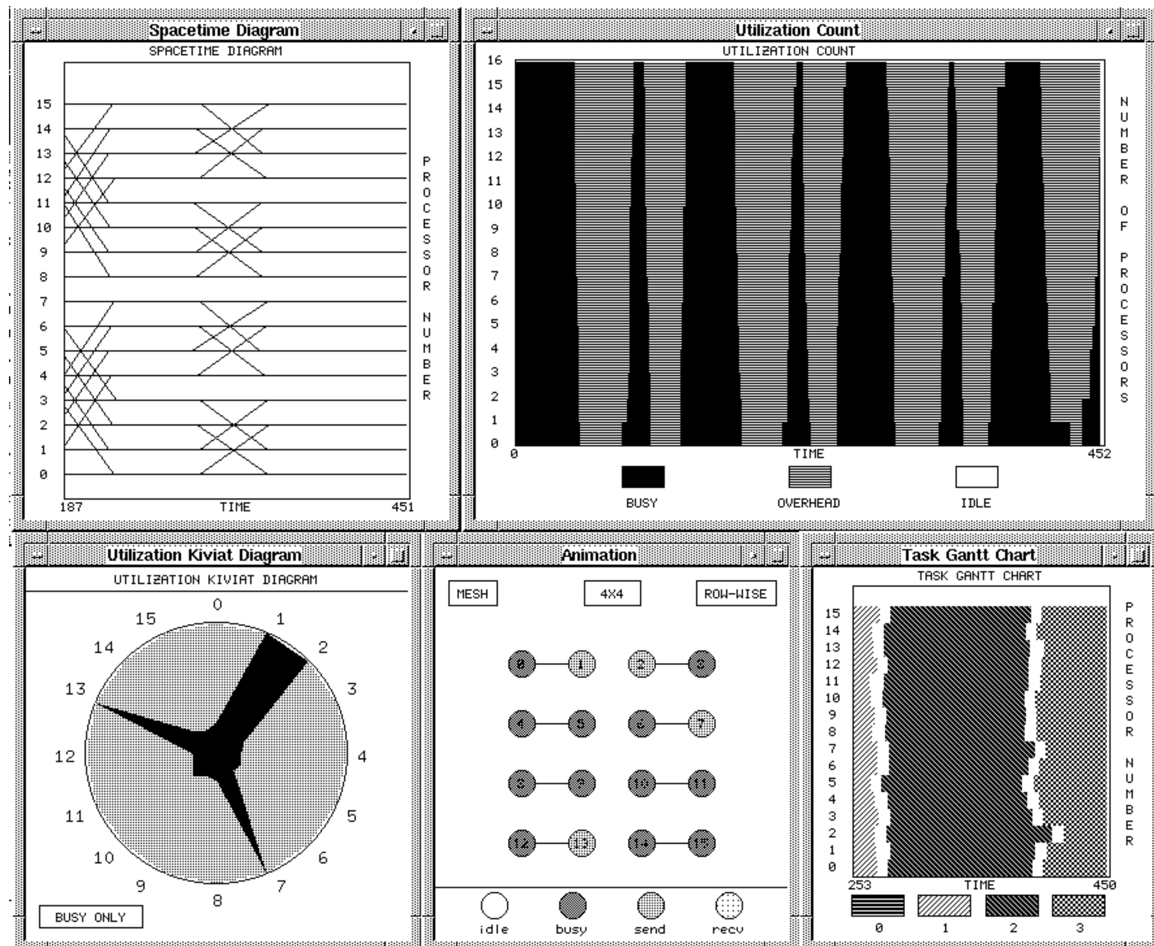

Figure 3.1: Some of the displays available from the ParaGraph system.

Performance data is quantifyable, so graphs, charts and similar diagrams work well for displaying it. These displays are supplied with most systems, so generating a new view of the data is very simple.

ParaGraph

The ParaGraph system [HE91] has a large number of predefined views for performance visualization. The goal of the system is to provide several ways of easily visualizing performance data. The system relies on the Portable Instrumented Communication Library (PICL) for data collection [GHPW90]. PICL is available in several vendors' parallel message-passing architectures (Thinking Machines, nCUBE, Cogent, Intel, Meiko, Symult). This is one of the advantages of the ParaGraph system: it can handle data produced by several different architectures.

ParaGraph uses trace files generated by PICL to show the performance of the architecture when executing that program. All the current visualizations are done "post-mortem". There is no way to control the execution of the parallel program from the ParaGraph system, so attempting visualizations on-the-fly would only result in a fast blur.

The emphasis of ParaGraph is on ease of understanding, ease of use and portability. It uses a large number of predefined visual perspectives which allow the user to choose the one that presents the best insights on the performance of the program. Some of these views are shown in figure 3.1. ParaGraph is also very easy to use, with barely any instruction needed for a novice user.

The information needed for the visualizations is automatically created by the calls to PICL routines. The program is not dependent on PICL, and could be used to visualize trace data generated by other programs, as long as the information is in the same format as the data generated by PICL. ParaGraph itself is written using C, and all the displays are created using X window calls, making ParaGraph portable to other Unix systems.

By their own admission, ParaGraph was not meant for application-specific visualizations. But the system has extensions to allow users to add displays that are specific to their applications. The creation of these displays is not trivial, but once created, these displays can be viewed along with the other generic displays.

One of the drawbacks of the system is that it simply creates the visualizations as fast as it can, so the length and timing of events is not intuitively obvious.

ParaGraph is a mature system, and a good one for what it was intended. It is easy to use, and the displays are informative, well designed and fast. We liked the fact that the learning curve is low and the amount of expected user intervention is minimal.

Pablo

The Pablo performance analysis environment is a system for the collection, analysis and presentation of performance data from parallel computer systems [ROA+91, RAM+92]. The environment includes a collection mechanism that generates self-documenting trace files where no data types or sizes are assumed. The analysis is done by connecting trace flows graphically to statistical data transformation modules. The resulting data can be presented using traditional visualization techniques as well as sonification techniques.

The emphasis of the system is on portability, scalability and extensibility. Portability means that, although the mechanisms for collecting data are system dependent, once the data is collected, the data transformations and visualizations should be system independent. Scalability refers to the increase in the number of processors in the parallel systems. The tool should be able to scale with the number of processors, not only gathering the data from larger systems, but also being able to present the data in an intuitive manner. Extensibility addresses the fact that different users have different expectations from the system. A novice programmer will not be willing to spend lots of time to get any usefulness from the system, whereas a more advanced programmer will take the time to learn and use different techniques if they will present better insights on the program.

These goals have directed the development into the design of two primary components of the system: portable software instrumentation, which generates trace data, and portable performance data analysis, which interprets and presents the trace data. The program instrumentation can be done interactively and three kinds of events are supported: trace, count, and time interval.

Trace events indicate the occurrence of an event, and can be accompanied by an arbitrary amount of associated data. Count events are associated with an event, but keep track of only the number of times the event occurs. Time interval events are associated with a pair of points in the program, and indicate the time elapsed while executing the enclosed section.

For analysis of the trace data, the user can create data analysis modules or use one of several standard modules provided with the system.

Pablo should integrate the performance data with the source code that produced it. This is a very important feature that is not currently implemented. Also, Pablo seems tailored to small MIMD systems, and not to data-parallel programs. Most of the displays provided by the system do not scale well with an increase in the number of processors, and they comment on the need for new techniques to analyze and present data from massively parallel systems.

Other systems

The Automated Instrumentation and Monitoring System (AIMS) provides facilities for the automatic insertion of instrumentation code [YHL+93]. The system has a monitoring facility that will save performance data, and two tools that process and display the data. The execution can be animated using the View Kernel tool, or statistics about the execution can be observed using the Tally tool.

The RP3 (Research Parallel Processor Prototype) system from IBM is important because they added a performance monitoring chip (PMC) to each of the processors in the system [WCC90]. As part of the RP3 system, a prototype visualization system [Kim90] was developed to display some of the data collected by the monitoring system. The goal of the visualization system was to give programmers a set of tools that they could use to find problems in their programs. They could run the program until some interesting event occurred, then use the tool as an aid to pinpoint the cause of the event.

This visualization system was created specifically for the RP3 system, and the instrumentation to produce the data for the visualizations had to be inserted into the code by the user. The graphics for each visualization were also created specifically for each application. Even the authors admit that this approach takes too much time and by the time the visualizations are in place, all the bugs have been fixed, making the visualizations useful only as instructional tools.

The Parallel Programming Instrumentation Environment (PIE) was created to aid in performance tuning of programs, which they call performance debugging. Their goal is not to make programs work, but to make working programs work faster.

PIE is tailored for shared memory parallel computers, and is implemented on top of MACH, an operating system developed at Carnegie Mellon University. It is meant to be portable to other similar systems. The displays available are limited, and they do not seem to scale well with an increase in the number of processors. One of the features of PIE is the ability to correlate the monitored events with a view of the source code.

3.2 Debuggers

Parallel program debuggers, like performance evaluation tools, can be used on a wide variety of applications with little customization. Debuggers can provide general information about the workings of a given parallel system, but the information is not very intuitive, and it is at a very low level of abstraction where what you see is the same data that the machine sees. The amount of data can be overwhelming for a person to comprehend. A good survey on debuggers for parallel programs was presented by McDowell and Helmbold [MH89]. Pancake and Utter give an in-depth look at visualization models for parallel debuggers [PU89].

The data collection for debugging systems has to include interprocess communication information as well as information about the behavior of each process. Some tools force the user to instrument the program manually (similar to adding print statements to interesting sections of code), and others just automatically add instrumentation to certain parts of the code. Perturbation created by instrumentation can cause different execution in asynchronous programs, which may hide existing bugs.

The speed of execution in debuggers is not critical. In fact, debugging usually proceeds slowly, with the user needing time to understand what is happening in the execution. This means that the data could easily be analyzed and sent directly from the parallel system to the visualization system. But, in most available systems, this is not the case, and data is stored in trace files for later viewing and reviewing.

One of the important issues in debugging is the timing of events. The problem is caused by the asynchronous execution on MIMD machines. Several systems reorder the trace data to present a clearer view of events, which are presented in a logical fashion instead of the way they actually occurred. In doing this sort of analysis, one must be sure not to produce erroneous sequencing of events.

In debugging, most of the data is just presented and the burden of analysis falls on the user. This means that most of the displays are not very abstract or complicated. Usually, there are some displays of the machine model being used, along with some trace event animation. Visualization use in debugging is mostly limited to displaying communication patterns between processors and to visualizing data that is distributed among the processors. It serves to locate some bugs, but not all.

Instant Replay

One difference between debugging parallel programs and debugging sequential programs is the possibility of non-determinism, making it possible for parallel programs to behave differently in successive executions. Instant Replay is a way of gathering data from a program execution to ensure that future runs will produce the same behavior [LMC87].

The main observation is that process communication and interaction is the cause of non-reproducible behavior, because a different sequence of interactions may cause a different result. Instant Replay models all process interactions as operations on shared data. The writes to the object will give the proper sequence of state transitions in the program, and the reads will indicate the sequence of events which should have occurred up to that point.

Instead of keeping a trace for all the data used in the interactions, the system only keeps track of the changes to the data by updating the version number of the data. By using the trace produced in the previous run as a guide, the system can ensure that the same version of the data is received at the same time in the next execution of the program. One advantage of keeping version numbers for the data instead of the actual value of the data is that the resulting traces are smaller and can be produced faster.

The key feature of Instant Replay is the ability to produce reasonably-sized traces that guarantee reproducible execution of a parallel program. It works on both shared-memory and message-passing machines, but relies on the fact that interaction between processes is a rare occurrence.

To ensure that a trace will be available in case of a failure, the tracing mechanism has to be enabled all the time. This means that the performance penalty for using the system

has to be minimized. They claim that the overhead for producing the trace will increase the program execution time by less than 5 percent.

Instant Replay does not use any visualization tools, but the techniques used to minimize the recording of trace data and the emphasis on reproducible behavior of programs are commendable goals for any parallel program visualization system.

Belvedere and Perspective Views

The major idea in the Belvedere system is that parallel programs are best understood by examining the patterns of interprocess communication [HC89]. Belvedere can be used to visualize information from an execution trace file and uses animation to display user defined communication events. The user can then detect bugs in the program by observing deviations from the expected pattern of communication.

One of the problems of animating communication events as they occur is that, because communication does not happen synchronously, the patterns may not match the user's mental model of how the communication should take place. To help identify patterns, Belvedere allows the user to define *abstract events*, or a series of events that should be grouped together. These abstract events are defined using a special event definition language.

Once the patterns are described, it is easy to find bugs in parallel programs if the actual communication pattern does not match the expected behavior. After using Belvedere on several different programs, the authors observed that animating the communication patterns is helpful, but that restricting the animation to different perspectives can make the information more meaningful.

This led the authors to study a technique they call Perspective Views [HC90]. They found that sometimes the information displayed by animated views can be enhanced by reordering events in ways that closely match the user's mental model of the program.

The major drawback of these systems is that abstract events have to be defined using an event definition language, which can be cumbersome. This also means that the user has to define all possible events before they are displayed. A deviation from the expected pattern can be caused by a mistake in the event definition.

Belvedere is a step in the right direction. It helps to visualize the execution of parallel programs and reveals incorrect actions by comparing them to the model of expected behavior. In the process, it gives the user a good, intuitive idea of the inner workings of the program. Visualizations in Belvedere are limited to small size programs. The views shown in the paper would not scale with an increase in the number of processes.

To allow the creation of abstract behaviors for large programs, the authors have introduced a new system, called Ariadne [CFH+93]. This system was designed for "scalable application of event-based abstractions." This new system does not include graphical displays, probably due to the difficulty of displaying scalable data.

MPPE - MasPar Programming Environment

The MPPE system is a tool for debugging and optimizing data-parallel programs for the MasPar MP-1 computer [Mas91]. This system uses a graphical user interface to allow the user to debug programs written for the MasPar. It also allows the user to animate the updating of variable values during execution and create some visualizations of selected data using a *data visualizer window* (see figure 3.2). This window will take a comparison

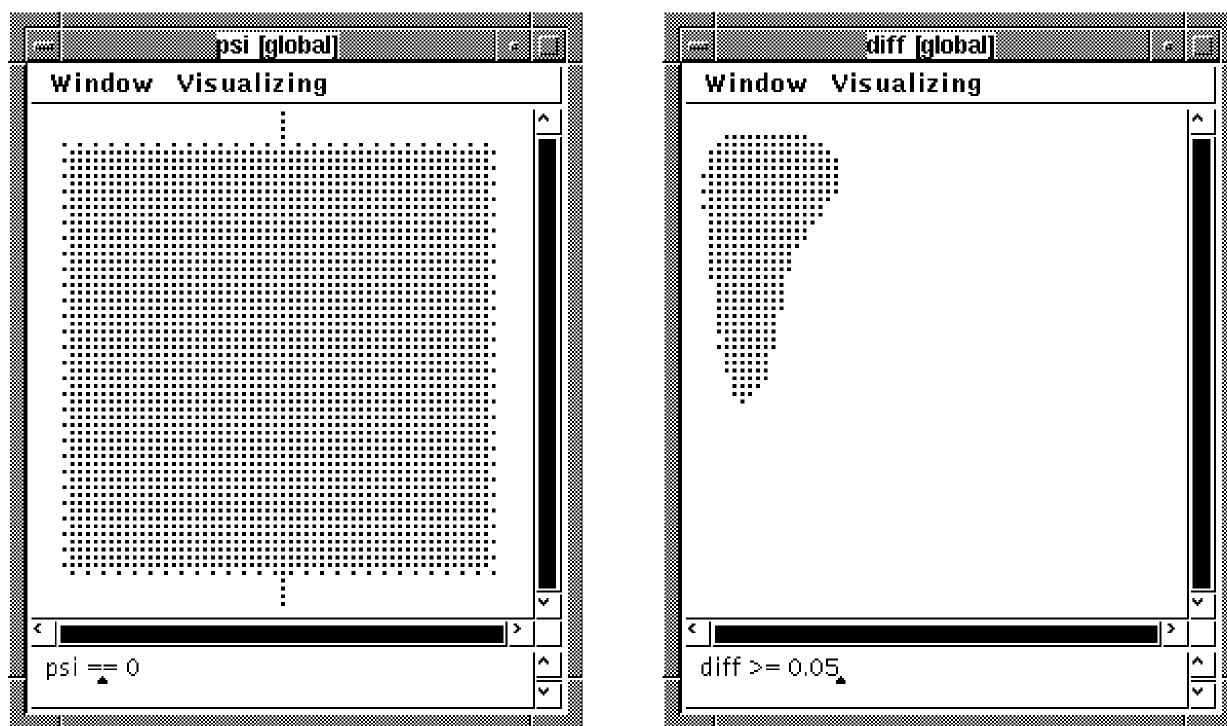

Figure 3.2: Data visualizer window from MPPE

expression (such as B == 0) and display one pixel per processor, where the pixel will be on if the criteria is satisfied in the corresponding processor. Further inspection can be done by using the mouse to click on individual pixels, where the actual data value at the corresponding processor will be displayed.

This system presents many great visualization ideas. The user interface is also very easy to use. The main problem is that it is only available for the MasPar, so it is not portable. The ideas are tailored for SIMD machines, where synchronous execution simplifies many of the animation issues.

Other systems

Bugnet is a system for debugging distributed programs running on a Unix network [Wit89]. It uses a monitoring facility to record interprocess communication and input/output for all the executing processes. All the information is traced and timestamped, allowing the user to back up the execution in case of an error and replay the events leading to the error in "almost" the exact sequence as they occurred before.

The system uses a checkpoint algorithm that saves a globally consistent system state at periodic intervals during the execution of the program. During these checkpoints, all processes stop executing and the state of each process is captured. Replaying the program consists of resetting the state to the last checkpoint and executing the trace entries since that point.

The PF-View system is a graphical debugger created to take advantage of the massive amounts of information produced by the IBM's Parallel Fortran Trace Facility (PF-Trace) [UHP91]. The PF-View system reads and processes program traces, using the resulting data to present a multi-level view of the execution of the program, which can be used for

debugging or for performance tuning. But perhaps the most important contribution of PF-View is the ability to correlate the behavior of the program with the source code that caused it.

The system starts displaying the highest level of the program, displaying it as a series of icons where sections of code are classified as serial, parallel loops or parallel cases. If more detail is desired, the user can click on the corresponding icon to expand the hierarchy and get further information on the section of the program. The hierarchy hides irrelevant details from the programmer, allowing him or her to concentrate on the important sections of code Animation is used to simulate the execution of the program as it happened in the trace. Changes in icon color will indicate the section of the code being executed.

The visualization of parallel machines is so important that there was a system that proposed the use of a parallel machine exclusively for monitoring and debugging a similar machine [RRZ89]. The proposed machine, called Makbilan, was a MIMD shared-memory machine. The goal of the Monitoring, Animating and Debugging (MAD) system was to do the monitoring in a non-intrusive, interactive and user-friendly way. The MAD machine would eliminate the need for large trace files which are not needed for interactive debugging.

3.3 Program visualization tools

Most program visualization tools available give intuitive displays of the parallel system at the expense of flexibility. These tools require programmers to create an abstract view of the program that is related to their mental model of the computation. The views created are designed for a particular program or algorithm, and different views are required for different algorithms. Stasko and Patterson present a discussion on some of the issues related to software visualization systems [SP92].

Algorithm animation can be considered part of program visualization, since the purpose is to give very intuitive displays of data. The views created for algorithm animation are extremely specific and are usually used for teaching purposes. Some of the algorithm animations that have been created are specific to certain kinds of algorithms (sorting, searching, etc.). One example of this is the Robust Animator of Fault Tolerant algorithms (RAFT) [Apg92]. This system allows the user to interact with the executing animation. The only drawback of the system is that you can only animate the selected algorithms, and creating new animations involves lots of work.

The data collection for program visualization is done by manually instrumenting the sections of code that supply the important information. There is no problem with instrumentation perturbation, since the goal is to enhance the understanding of the program, and not to see how fast it is executing. The collection is done by software, since the data collected varies from program to program.

The data is normally stored in trace files to allow multiple replays of the execution. To run the visualization on-the-fly, some way of slowing down the parallel program must be used. The trace data could be analyzed before being stored because we know which data we will need to keep to produce our displays.

Analysis of the data consists of determining the combination of trace events that constitute one of the interesting abstract events in the execution model. This forces the user to figure out how high-level events will be reported in the trace. These events can then be sent to the visualization system for display.

The display of data for program visualization requires lots of thought before creating a view. The view designer must find out the best way to convey events in an intuitive manner that will enhance program understanding. Views are hard to create, and may be a "one-shot" deal (usable only with this program). Much of the research in this area involves the simplification of creating the views, as well as the reusability of the building blocks used to create the views.

Voyeur

The Voyeur system was created to improve the trace facility of the Poker programming environment [SBN89]. The emphasis of Voyeur was to facilitate the creation of application-specific views of parallel programs. It was used to create views that were close to the programmer's mental model of the problem. The main simplification was to construct the views hierarchically, which makes it possible to reuse some of the parts already created.

The visualization system is capable of using trace data as input, as well as data generated directly from a parallel architecture or a simulator. The user starts the system by connecting the input to a trace file or to a pipe used for interactive viewing. After initialization (setting the number of processors, etc.), the execution can proceed continuously or by single-stepping through the program, and it can be interrupted as needed.

User input is required for the generation of each new view, and this is one of the drawbacks of the system. Once a view for a program is built, it can be reused by similar programs, since the visualization system is separate from the data generating architecture. They have even used the system to animate sequential programs.

The views presented by the Voyeur system range from textual display of the contents of variables to complicated abstractions, where information is displayed as sharks and fish. The system is written in C and uses X windows, which makes it portable to other architectures.

Zeus

The Zeus algorithm animation system [BH92] is the new system proposed by Marc Brown, the creator of Balsa-II [Bro88b] and one of the original proponents of algorithm animation [Bro88a]. The system expands on the available algorithm animation techniques by adding color and sound as new dimensions for conveying information.

Some of the techniques for algorithm animation discussed in the Zeus paper include:

- Multiple views — It is easier to understand an algorithm if you have different ways of looking at the execution. Also, emphasizing one feature in each view will ensure that one view will not be too overloaded with data, making it incomprehensible.
- State cues — Changes in the state of a data structure should be represented by changes in the graphical representation on the screen.
- Static history — A trace of where the algorithm has been is sometimes very helpful when one is trying to figure out how it got to the current state. By looking at a history of major changes, the path to the present state can usually be deduced.
- Continuous versus discrete transitions — For small data set, continuous "animated" transitions are better in depicting the changes in the algorithm (two items slowly exchanging place look better than if they magically switch place). But with large enough data sets, discrete transitions will look smooth, because the "in-between" details are lost.

- Contrasting algorithms — This is particularly helpful when there are two or more algorithms that do the same job. By showing the algorithms side by side, the user will gain an understanding of the advantages of each of the algorithms.
- Input data selection — Small amounts of data are best to introduce an algorithm, whereas large amounts of data can be used to understand the algorithm's behavior better. For pedagogical purposes, using a fixed data file is preferable, since it will highlight the important aspects of the algorithm. But it is also desirable to allow the users to input their own data, so they can customize the animation.

Color can be used to give cues about the state of data structures, to highlight specific activities, to unite related information in different views, to emphasize any obvious patterns, and to give a sense of history. Sound is more difficult to use, but it can be effective in reinforcing existing visualizations, conveying patterns of activity, replacing non-critical visual aid and signaling exceptional conditions.

Zeus presents many effective ways of conveying information about algorithms to a user. The system is intended as an instructional tool — the programmer must figure out the best way to display the information in order produce effective and informative animations. Zeus can be used to animate sequential as well as parallel algorithms.

Even though the techniques discussed in the Zeus paper seem very valuable, their effectiveness has not been formally evaluated. It would be interesting to see how much each of these techniques helps in clarifying the behavior of algorithms.

Polka

Polka [SK92] is the successor of Tango [Sta90], and it is a system for creating animations for specific algorithms. The system emphasizes the simplification of creating views by giving several primitives that, though not extremely easy to learn, are much simpler than the equivalent X windows programming primitives.

The Polka system separates the code to be animated from the actual animation "scenes". It is the job of the animator to determine the interesting events that give the information to be animated, and to add the corresponding calls to the Polka animation. Then the animator has to implement the views that animate the program and correspond to the calls made by the inserted functions.

The one improvement of Polka over Tango and other similar systems is the ability to animate parallel programs easily. The system allows the design of overlapping animations easily because it programs each individual object's actions independently from other objects.

Figure 3.3 shows two frames from an animation of a parallel matrix multiplication algorithm done using Polka. Creating the animation involved lots of thinking about how to design the visualization. This visualization will probably only be useful for this particular algorithm. One of the main drawbacks of Polka is the amount of intervention that is required from the animator in order to create the application-specific views.

Other systems

The Integrated Visualization Environment (IVE) system [FLK+91] is a system for visualizing parallel programs executing in SIMD machines. The system allows for 3 kinds of visualizations: Program visualization (eg. calling diagrams and dependency graphs), process visualization (eg. description of the state of the program) and application visualization (eg. specific abstractions for applications).

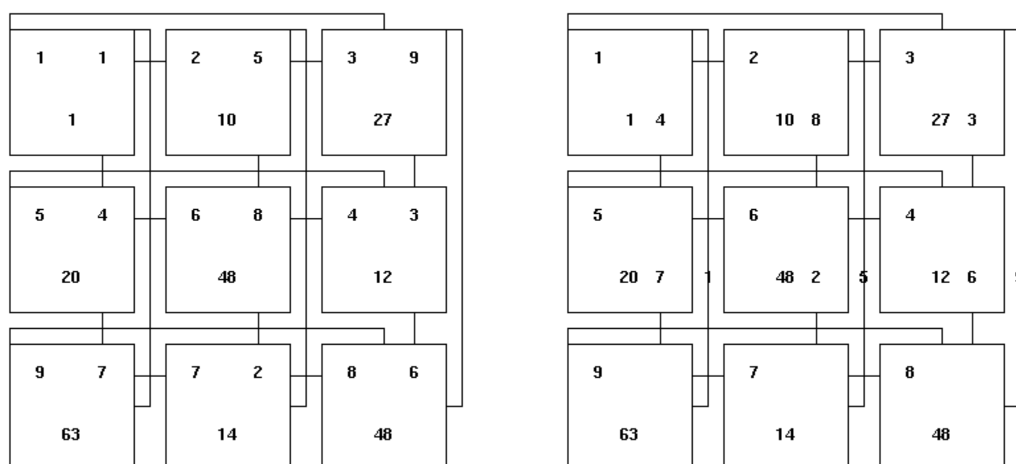

Figure 3.3: Two frames from a visualization of a parallel matrix multiplication algorithm for a SIMD machine. The left frame shows the values for the first matrix in the upper left, the second matrix in the upper right and the current total in the bottom of each processor. The right frame shows the columns of the second matrix rotating clockwise. This animation uses color and motion to highlight important information.

The main abstraction used by the IVE system is the grid. It uses a grid to represent the processors in the machine, since the program visualizes SIMD machines. Many of the views generated by the system are based on this grid.

One of the goals of the IVE system is to reduce the amount of effort required to create visualizations, since visualizations that require too much effort to produce will not be considered worthwhile and will not be used. The idea is to use the computer as an aid in the design of the visualizations. While the relative automation of the views is a step in the right direction, the user still has to be significantly involved, so it is not automatic.

The approach taken by the Parallel Animated Debugging and Simulation Environment (PARADISE) [KC91] is to create a tool for aiding the user to develop visualizations of the given trace data. They assume that the trace data will be collected using other methods, and it will be available in the form they require. In a sense, they only deal with the visualization part of the system, using their tool to process a predefined stream of trace events and display them in a user specified manner.

The goal of the system is to ease the creation of custom visualizations for analyzing parallel systems, but the construction of new visualizations seemed to require quite a bit of effort. The visualizations created using PARADISE will not easily scale to larger systems. Since the system is used for visualizing whatever trace file is given, there should be no requirement for the system to be parallel.

The goal of the Pavane system [CR91] is to develop a methodology for selecting the proper visualization of a concurrent computation. The method they use focus on the abstract formal properties of the computation. A visualization in Pavane is a mapping from a computational state to an image on the screen. The system assumes that the underlying computation can be characterized by a state that goes through atomic changes as the computation progresses.

The rules for creating a visualization are very complex and place a big burden on the programmers. The advantage is that the visualizations are tied to program verification.

The Parallel Architecture Research and Evaluation Tool (PARET) is a package that allows the visual study of a simulated distributed memory MIMD computer. The goal is to study multicomputers as systems, not as separate components. PARET was intended as a laboratory tool and consists of an interactive graphical front end, a simulator back end and a library of input files. It seems to be a good aid for gaining better comprehension of parallel environments, using the models available from the input library. Aside from these models, there is no obvious way of creating new models or for studying the behavior of a particular program.

The system presented by Williams and Lamont [WL91] integrates two systems developed at the Air Force Institute of Technology — AAARF and PRASE. The major emphasis of this system is the animation of parallel algorithms for improving the understanding of the algorithms. It also includes some limited capabilities for performance data display. Instrumenting the program to get performance data is done almost automatically by the PRASE preprocessor. A little more effort is required to get algorithm data.

An area where the views are fairly easy to generate is in systolic algorithm visualization. The reasons systolic algorithms are easier to visualize are:

1. They execute synchronously.
2. A grid is a good abstraction for their execution.
3. The interprocess communication is regular.
4. All processors execute the same program.

There are several systems that deal with the coding, execution and visualization of systolic algorithms. Both SDEF [EC88] and Hearts [Sny87] provide different environments for the definition of the algorithm, and for visualizing the execution of the implementation.

NSL [Hug92] is a language for the implementation of systolic algorithms. As part of the compilation, hooks are created to a visualization system (implemented using Xtango [Sta90]) that will animate the execution of the algorithm while highlighting the line of code being executed. One drawback of NSL is its current limit to one-dimensional systolic arrays.

3.4 Towards scalable visualizations

A common problem with most of the visualizations in the current systems is the lack of scalability with increases in the problem size. This problem is critical with the arrival of systems with thousands of processors. There are some systems that have presented ideas about the creation of scalable views.

A good description of the problem of scalability in visualization systems is presented by Couch [Cou93a, Cou93b]. He defines a scalable view as "one whose format, clarity, meaning, and size are independent of the number of processing elements involved in computation." The author presents some guidelines that are needed to ensure that the views created are scalable. These guidelines are incorporated into Couch's Seeplex system.

The Seeplex system provides two kinds of views: scalable and scrollable. Scalable views show global data and scrollable views show local data. The system allows selection of data for display in the local views, as well as zooming in and out of the global views. The visualization is always done by showing a graph plotting the values of the given data. The main visualization shows the distribution of the data values. In figure 3.4 we can see how Seeplex handles some of the scalable visualizations.

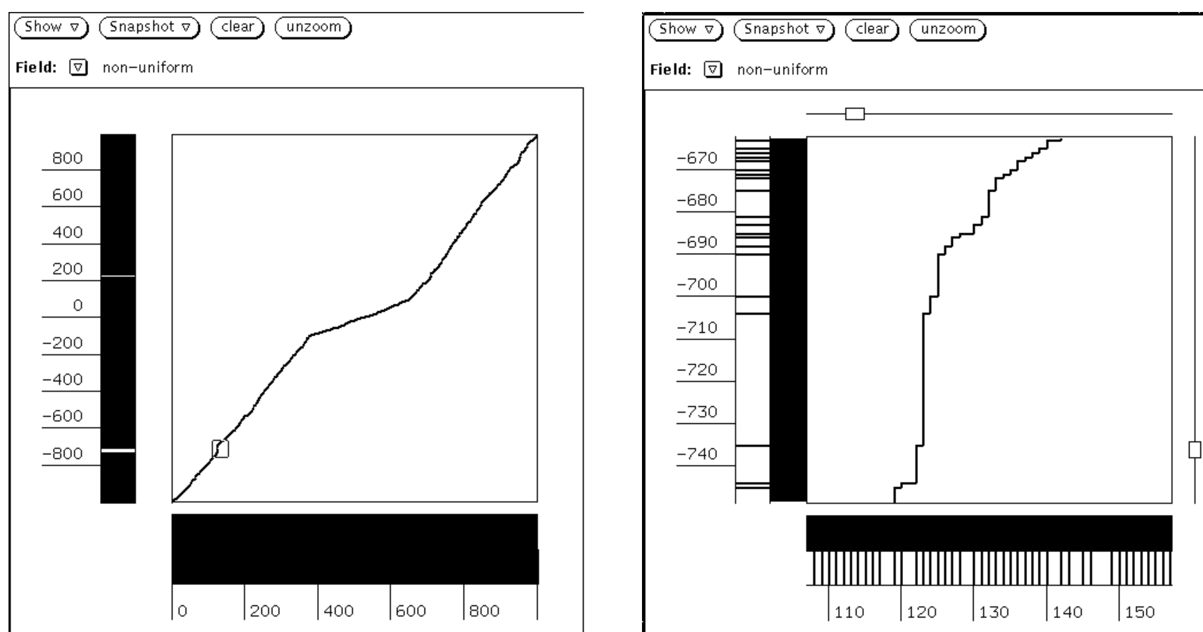

Figure 3.4: Scalable visualizations from Seeplex. The left figure shows all the available data sorted by value. The range of values is shown in the Y axis. The rectangle on the graph indicates a section of data selected for further inspection. This data is shown in the figure on the right.

Waheed and Rover [WR93] use techniques from the area of image processing to visualize performance data from massively parallel systems. These techniques are scalable to thousands of processors. They also use general purpose visualization tools like AVS and Matlab to analyze the data and create new visualizations.

The main contribution of these papers is an early attempt to produce scalable visualizations of data. More studies are needed to get better methodologies for the creation of scalable visualizations.

4 Need for scalable visualization

The latest trend in parallel processing is toward scalable programs — programs that are able to divide the workload among the number of processors available at execution time. The advantage of scalable programs is that there is no need to modify them when more resources become available, and they are able to effectively use many processors to reduce the time required to solve a problem.

But increasing the number of processors does not guarantee a corresponding reduction in execution time. The problem is that along with the increase in processing power comes an increase in complexity, with more inter-processor communication and more things happening in parallel.

Visualization could be used to aid in understanding the inner workings of scalable parallel programs. But most of the available visualization systems have limits to the number of processors they can visualize, and some of the views become useless for large number of processors. For instance, in figure 4.1 we can see how a view from ParaGraph that is useful for 16 processors is not useful for 128 processors.

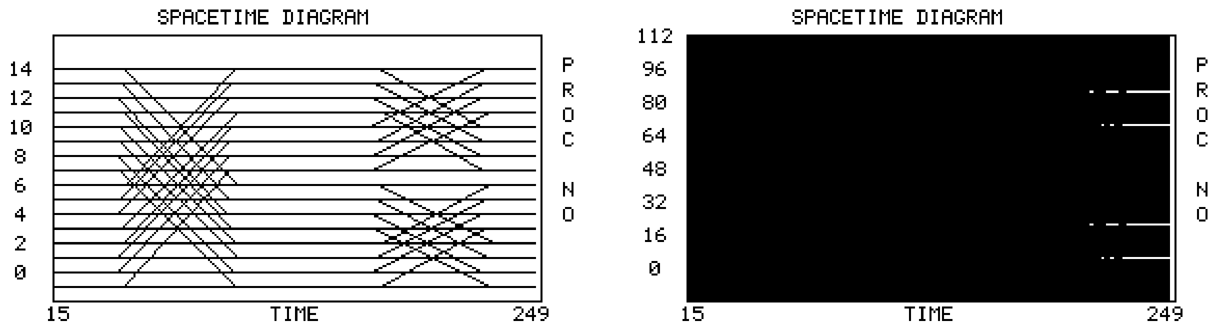

Figure 4.1: Here is one of the problems with non-scalable visualizations, taken from the ParaGraph system. The view on the left shows the messages passed between processors in solving a fast Fourier transform with 16 processors. On the right we see the same view when the program is executed on 128 processors.

The role of the visualization system is to present large amounts of information in an intuitive way to the user. In our example, we are more interested in viewing patterns of communication than in seeing every processor to processor communication. Therefore, creating 16 groups of 8 adjacent processors and displaying the group to group communication would generate a view similar to the first view that would give us much more information than the view generated by ParaGraph.

Future research should address the need for better ways of analyzing and visualizing the information produced by a large number of processors. In the rest of this section, we will detail some of the areas that need to be considered in developing tools for scalable visualization of parallel systems.

4.1 Tool philosophy

Experience has shown that a visualization system will not be used if it is difficult to create visualizations. To attract users, a visualization system should have ways of generating intuitive default views automatically. The main attraction of the ParaGraph tool is that the views are created with no extra effort from the programmer. But ParaGraph offers no help in selecting an appropriate view for the program. The user must choose from a confusing menu of available views to try to find a visualization suitable for the executing program.

Future systems should include several default views that can be used immediately to visualize programs. The system should select an appropriate view to use as the default for a given program, and give the user the ability to customize the views and to override the default selections.

Ultimately, it is the users of the visualization systems who know exactly what they want to investigate. Therefore, future systems should allow flexibility in the selection criteria for visualizations. The users should be able to select processors for detailed inspection, or select global views to see general patterns in the execution. Systems should also provide the ability to zoom in for more details at any time, using the global views as a road map to interesting areas.

One important aspect that needs to be investigated is the usefulness of the generated visualizations. It seems that everyone claims that their system creates helpful visualizations, but nobody has done empirical studies to find which visualizations are more informative

or to see how much visualizations actually help. Of course, the ultimate indicator of the usefulness of a system is if people continue to use it, even when they are not asked to, and most of the currently available systems fail this criteria.

4.2 Better analysis methods

Current visualization systems present all the data generated by the program. This is completely unnecessary, since most of the data does not help in understanding the behavior of the program. The idea is to develop ways of determining which sections are more interesting, and have the capability of giving more details when these sections are encountered. The system could use statistical analysis to select areas for detailed inspection and use scalable methods that give an intuitive global presentation of the program execution.

Statistical analysis

Visualization systems are used to understand complex events. It is wasteful to create visualizations for things that are obvious. The system should determine the areas that will benefit from the use of visualizations, and generate insightful views for those areas. The visualization system should use the information provided by the executing program to determine where to put its emphasis.

Statistics for important values should be kept during the execution. The visualization system can examine these statistics and present more details about the execution when they meet a pre-determined criteria. For instance, this can be used to examine the execution in more detail when the number of active messages reaches a given limit, or when some percentage of processes are idle.

The statistical information may also be used by the visualization system to decide which processors should be viewed individually. The user may want to focus on the processors that are working harder, or focus on a group where processors have big differences in usage statistics. Using this information, the system can steer the user into potential problem areas, where the visualization will be most useful.

Scalable methods

One way of creating scalable visualizations of parallel program execution is to group processors into clusters. The system can then show the interactions between the groups of processors as a high level view of the execution. To make such a view intuitive, the processors must be grouped in clusters that resemble the mental picture used by the programmer.

The simplest way to join processors together in scalable views is to use physical proximity. It would be more desirable to have automatic ways to group together processors according to what they are doing. Some work in this area has been done by Kunz [Kun93], but there is still much to be done.

4.3 Ties to program

Future visualization systems should have strong ties to the actual program being visualized, not just to the execution trace generated by the program. With this goal in mind, future systems should correlate the visualizations to the source code that produced them

and allow visualizations of arbitrary program data. Ultimately, the visualization systems should control the program execution, generating visualizations on-the-fly.

Ties to source code

One of the biggest complaints of users of visualization systems is the lack of information available to determine which sections of code produced the data being visualized [Fer93]. Ties to source code could help the users trace some abnormality to the sections of code that may have caused it. This can initially be done by adding a line to the trace indicating the file and line of code being executed. This information can be created automatically by aliasing the communication routines to include the file and line when the call is made.

Ties to program data

A visualization system should help visualize any kind of data supplied to it. The emphasis on parallel systems has been to visualize performance information, since it is the information that is most readily available. We should be able to visualize other information related to the execution of the programs, such as data that is distributed among the processors, or data being passed between the processors.

One way of getting more information about the messages being passed by the system is by annotating the messages when sent and received. The extra information can contain the format of the data, the number of the iteration in a loop, or other information that can be helpful when visualizing the data. Advanced compiler technology could automatically generate the tags for the messages. One drawback of annotating messages is that it rapidly increases the amount of data generated by the execution.

Ties to program execution

Current practice for generating visualizations is to run the instrumented program to generate a trace file of the execution, and then use this trace file to produce the visualizations. This has the advantage of avoiding too much perturbance from the actual tracing, but it generates very large trace files, which can become a problem for large number of processes or for long executions.

Generating the visualizations on-the-fly eliminates the need for storing intermediate trace files. It also allows direct debugging of programs, where the user can look at the data as it is produced, instead of looking at trace files. This will greatly increase debugging speed, but could cause other problems. Visualization systems should be able to control the execution of the parallel program, to effectively slow it down while the information is displayed graphically. Slowing down the program will also increase the "probe effect," which may cause different execution sequences by slowing down some processors and not others. When generating visualizations on-the-fly, systems must slow down all processors equally in order to reduce the probe effect. Careful analysis of the perturbation caused by the instrumentation may help to minimize the differences between a regular execution and an execution for visualization purposes.

4.4 Other issues

There are other techniques to improve the information conveyed by visualizations. Future systems should use color and sound to enhance the visualizations they produce, but the must avoid cluttering the data with excessive bells and whistles.

Use of color

Computers have made it so easy to add colors to views that in many cases the colors have been overused. The result is computer visualizations that look very attractive, with many bright colors, but are not very informative. Usually, the visualizations end up looking like video games.

In his book *Envisioning Information*, Tufte gives some guidelines on using color in displays of information [Tuf90]. His first principle is: "Above all, do no harm." He proceeds to list some categories where color may be used, such as for:

- Labels — to distinguish different structures or data.
- Measures — to give information about data values.
- Representation — to imitate the real color of the data, ie. using blue if viewing water data.
- Decoration — to make the data more attractive, without overwhelming the data contents.

Many computer visualization systems do not use color effectively to transmit information. For visualization of parallel systems, color can be used in several areas. Color can be used to distinguish sources of data, as well as for highlighting important areas, without forcing the user to continuously refer to a chart that indicates what the colors mean. Different intensities can be used to typify data values. And finally, it can be used sparingly as decoration for generated visualizations.

Use of sound

Sound can be used as an alternative to visualizations to portray the behavior of parallel programs. Francioni et al. present three experiments where execution behavior is mapped to sound [FJA91]. The first study tracks processor loads, and can be used to determine load balance. The second study is related to the flow-of-control of the program, and produces sounds when selected sections of code are executed on each processor. The last experiment maps communication events to sounds, and is used to portray the communication patterns between processors.

Sounds are different from visual aids because we can detect the occurrence of sounds without having to listen for them. Also, we can detect some relationships better with our ears than with our eyes. Sounds are really good to indicate an exceptional occurrence, since it will not detract from the normal behavior of the program. In general, sounds should not be used by themselves, but can be used to enhance the information displayed visually.

5 Comments and conclusion

In the area of visualization of parallel systems, there is still a need for easily-generated, intuitive visualizations. What makes a good visualization for parallel systems? When we asked users of parallel systems, the unanimous response was "I don't know." There is no consensus on which features are essential for parallel program visualization. The one thing everyone agrees is that the insights gained by using the system have to justify the effort needed to use the system. Current systems in general require too much effort to create visualizations that end up not being too helpful, so users give up on them. The attitude of most people we talked to was "I don't know what makes a good system for visualizing parallel programs, but if I see one I will know."

One thing that has become obvious is that visualization systems will not magically make everything clear. The users of a visualization system must know what they are looking for, must have a clear idea of what the program does and where it may run into trouble. Showing a visualization of a program to someone without knowledge of the context will, at best, create a pretty picture. If a visualization is being used for instructional purposes, then the student should have a good idea of what is supposed to happen, which will be corroborated by the visualization. If instead, it is being used to track down some possible problem, then the user must have a clear picture of how the execution should proceed, and use the visualizations to note deviations from the expected model.

Visualization shows a lot of promise as a way to enhance the understanding of the inner workings of parallel systems. It is the perfect vehicle for filtering the vast amounts of data generated by the systems and presenting it in intuitive ways. Better models of visualization for parallel systems will only increase our insights and will lead to the design of better, more efficient programs.

References

[Apg92] Scott W. Apgar. Interactive animation of fault tolerant parallel algorithms. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 11–17, 1992.

[BH92] Marc H. Brown and John Hershberger. Color and sound in algorithm animation. *Computer*, 25(12):52–63, December 1992.

[Bro88a] Marc H. Brown. *Algorithm Animation.* The MIT Press, Cambridge, MA, 1988.

[Bro88b] Marc H. Brown. Exploring algorithms using Balsa-II. *Computer*, 21(5):14–36, May 1988.

[CFH+93] Janice Cuny, George Forman, Alfred Hough, Joydip Kundu, Calvin Lin, Lawrence Snyder, and David Stemple. The Ariadne debugger: scalable application of event-based abstraction. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–95, May 1993.

[Cou93a] Alva L. Couch. Categories and context in scalable execution visualization. *Journal of Parallel and Distributed Computing*, 18(2):195–204, May 1993.

[Cou93b] Alva L. Couch. Locating performance problems in massively parallel executions. *Proceedings of the IEEE*, 81(8):1116–1125, August 1993.

[CR91] Kenneth C. Cox and Gruia-Catalin Roman. Visualizing concurrent computations. In *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pages 18–24, 1991.

[EC88] Bradley R. Engstrom and Peter R. Capello. The SDEF systolic programming language. In *Concurrent Computations*, pages 263–301, 1988.

[Fer93] Robert Ferraro. Personal Communication, 1993.

[FJA91] Joan M. Francioni, Jay Alan Jackson, and Larry Albright. The sounds of parallel programs. In *Proceedings of the Sixth Distributed Memory Computing Conference*, pages 570–577. IEEE Computer Society, 1991.

[FLK+91] Mark Friedell, Mark LaPolla, Sandeep Kochhar, Steve Sistare, and Janusz Juda. Visualizing the behavior of massively parallel programs. In *Supercomputing '91*, pages 472–480, 1991.

[GHPW90] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A user's guide to PICL: A portable instrumented communications library. Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, 1990.

[HC89] Alfred A. Hough and Janice E. Cuny. Initial experiences with a pattern-oriented parallel debugger. *SIGPLAN Notices*, 24(1):195–205, January 1989.

[HC90] Alfred A. Hough and Janice E. Cuny. Perspective Views: A technique for enhancing parallel program visualization. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume II, pages 124–132, 1990.

[HE91] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

[Hug92] Richard Hughey. Programming systolic arrays. In Edward Lee and Teresa Meng, editors, *Proceedings of the International Conference on Application Specific Array Processors*, pages 604–618. IEEE Computer Society, August 1992.

[KC91] James Arthur Kohl and Thomas L. Casavant. Use of PARADISE: A meta-tool for visualizing parallel systems. In *Proceedings of the 5th International Parallel Processing Symposium*, pages 561–567, 1991.

[Kim90] Doug Kimelman. Environments for visualization of program execution. In Simmons and Koskela [SK90], pages 135–146.

[KS92] Eileen Kraemer and John T. Stasko. The visualization of parallel systems: An overview. Technical report, Georgia Institute of Technology, 1992.

[Kun93] Thomas Kunz. Process clustering for distributed debugging. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 75–84, May 1993.

[LMC87] Thomas J LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[Mas91] MasPar Computer Corporation. *MasPar Programming Environment (MPPE) User Guide*, 1991.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[PU89] Cherri M. Pancake and Sue Utter. Models for visualization in parallel debuggers. In *Supercomputing '89*, pages 627–636, November 1989.

[RAM+92] Daniel A. Reed, Ruth A. Aydt, Tara M. Madhyastha, Roger J. Noe, Keith A. Shields, and Bradley W. Schwartz. *An Overview of the Pablo Performance Analysis Environment*. Department of Computer Science, University of Illinois, Urbana, Illinois 61801, November 7 1992.

[ROA+91] Daniel A. Reed, Robert D. Olson, Ruth A. Aydt, Tara M. Madhyastha, Thomas Birkett, David W. Jensen, Bobby A. A. Nazief, and Brian K. Totty. Scalable performance environments for parallel systems. In *Proceedings of the Sixth Distributed Memory Computing Conference*, pages 562–569. IEEE Computer Society, 1991.

[RRZ89] Robert V. Rubin, Larry Rudolph, and Dror Zernik. Debugging parallel programs in parallel. *SIGPLAN Notices*, 24(1):216–225, January 1989.

[SBN89] David Socha, Mary L. Bailey, and David Notkin. Voyeur: Graphical views of parallel programs. *SIGPLAN Notices*, 24(1):206–215, January 1989.

[SK90] Margaret Simmons and Rebecca Koskela, editors. *Performance Instrumentation and Visualization.* ACM Press, 1990.

[SK92] John T. Stasko and Eileen Kraemer. A methodology for building application-specific visualizations of parallel programs. Technical Report GIT-GVU-92-10, Georgia Institute of Technology, June 1992.

[SKB89] Margaret Simmons, Rebecca Koskela, and Ingrid Bucher, editors. *Instrumentation for Future Parallel Computing Systems.* ACM Press, 1989.

[Sny87] Lawrence Snyder. Hearts: A dialect of the Poker programming environment specialised to systolic computation. In W. Moore, A. McCabe, and R. Urquhart, editors, *Systolic Arrays*, pages 71–80. Adam Hilger, Boston, MA, 1987.

[SP92] John T. Stasko and Charles Patterson. Understanding and characterizing software visualization systems. In *Proceedings of the IEEE 1992 Workshop on Visual Languages*, pages 3–10, 1992.

[Sta90] John T. Stasko. TANGO: A framework and system for algorithm animation. *Computer*, 23(9):27–39, September 1990.

[Tuf90] Edward R. Tufte. *Envisioning Information.* Graphics Press, Cheshire, CT, 1990.

[UHP91] Sue Utter-Honig and Cherri M. Pancake. Graphical animation of parallel Fortran programs. In *Supercomputing '91*, pages 491–498, 1991.

[WCC90] Brantley William C and Henry Y. Chang. Support environment for RP3 performance monitor. In Simmons and Koskela [SK90], pages 117–134.

[Wit89] Larry D. Wittie. Debugging distributed C programs by real time replay. *SIGPLAN Notices*, 24(1):57–67, January 1989.

[WL91] Edward M. Williams and Gary B. Lamont. A real-time parallel algorithm animation system. In *Proceedings of the Sixth Distributed Memory Computing Conference*, pages 551–561. IEEE Computer Society, 1991.

[WR93] Abdul Waheed and Diane T. Rover. Performance visualization of parallel programs. In *Visualization '93*, pages 174–181, 1993.

[YHL+93] Jerry Yan, Philip Hontalas, Sherry Listgarten, Charles Fineman, Melisa Schmidt, Pankaj Mehra, Sekhar Sarukkai, and Cathy Schulbach. The automated instrumentation and monitoring system (AIMS) reference manual. Technical Report TM-108795, NASA, November 1993.