The PARADE Environment for Visualizing Parallel Program Executions: A Progress Report

John T. Stasko

Graphics, Visualization, and Usability Center
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

E-mail: stasko@cc.gatech.edu

Technical Report GIT-GVU-95-03

Abstract

This report describes the current status of the PARADE visualization environment. PARADE supports the design and implementation of software visualizations of parallel and distributed programs. It contains primary components for monitoring a program's execution, building the software visualization, and mapping the execution to the visualization. In this report we provide brief descriptions of many of the projects that comprise the PARADE environment, and we provide references to more detailed information on the projects.

1 Introduction

Every year an increasing amount of software is being written for parallel and distributed computers. Unfortunately, parallel programs are more difficult to write, debug, evaluate, optimize, and understand than serial programs because of the concurrency they manifest. Programmers must coordinate and synchronize communication between processes, they must control access to shared resources, and they must carry these actions out as efficiently as possible.

One approach to facilitating the growth of parallel and distributed programming involves the development of new programming languages and new hardware. Recently, however, researchers have begun to focus on the importance of good software tools to assist developers of concurrent programs[Che93, PC94]. These tools include debuggers, performance monitors, execution analysis and replay tools, and other aids.

Our research also has focused on tools for program understanding and development, but we have a different emphasis: *software visualization* tools[SP92]. A key component of understanding a program execution is knowing what is occurring in the program, how individual processes are working, and how they are communicating. By visualizing the operations of a program, we help convey to the programmer what those operations are, and if they are behaving in the desired manner.

Software visualization taps into the highly developed visual systems of humans. People have a tremendous ability to track patterns, observe images, and detect anomalies in the things they see. A well constructed picture, diagram, or visualization can communicate much more information in a small space than a corresponding textual description[Tuf83, Tuf90].

Recently, increasing attention has focused on the use of software visualization to assist parallel programming (see [KS93] in particular and [CE93] for a collection of articles on this topic). This paper is a progress report on a project called PARADE (PARallel program Animation Development Environment) that was started in 1991. Its goal was to develop an environment that facilitates the creation and use of visualizations by programmers developing concurrent programs. A report roughly summarizing the original proposal for the project can be found in [SAK91]. Primary support for the project has been a three year National Science Foundation grant (CCR-9121607). Portions of the project also have been supported by Kendall Square Research and by an Intel Graduate Fellowship.

In the remainder of this report we describe the current status of the PARADE environment and we describe the components and sub-projects within it.

2 Overview

A number of objectives have guided our efforts in building the PARADE environment:

- PARADE should support visualizations of many different types of programs from different architectures, different programming models and languages, and different applications. It should support the creation of automatic, canonical program views as well as application-specific, algorithm animation style views. It should support both performance visualizations and correctness visualizations.

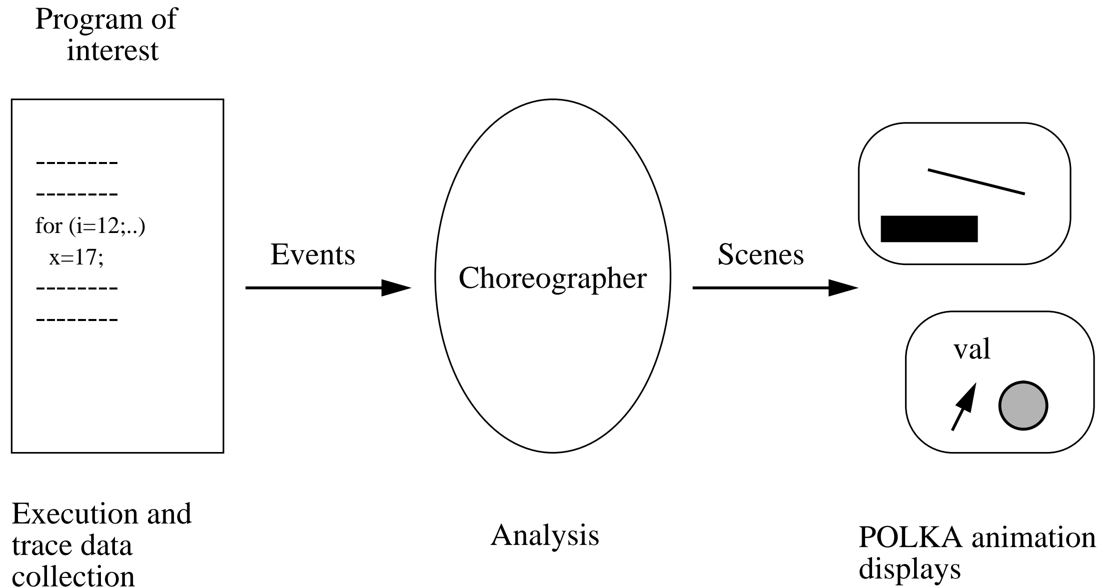

Figure 1: PARADE system overview, highlighting the three major components.

- The environment should be easy to use. A developer need not be a graphics expert to work with it.
- The visualizations developed in the environment should be relatively sophisticated and also aesthetically pleasing. They should support color, animation, and be able to depict concurrency in a program's execution.
- The environment should support visualizations that portray other feasible program executions. That is, a viewer may wish to examine a program execution as it occurred with respect to a global clock, as it occurred under some logical clock ordering[Lam78], or as it might have occurred under some other valid event ordering.

It is easy to see how these objectives have led us to develop a flexible environment with many different components and tools, as opposed to a monolithic system built to do only one thing. In fact, the PARADE environment can be conceptualized as having three primary components. Figure 1 presents a high-level overview of the organization of PARADE.

The first component is the program monitoring aspect of the environment. Basically, to drive a visualization, information about the program's execution is required. Many details about the execution must be known in order to build an appropriate visual presentation of it.

The third component of the environment, shown to the right side of Figure 1, is the support visualization/animation system. In PARADE the Polka animation system is used to build all the graphical views. Polka runs on top of the X Window System and it achieves the objectives mentioned earlier.

The second or middle component of the environment provides the mapping from program execution data to appropriate visualization actions. In PARADE, simple mappings are possible, but usually we utilize a system called the Animation Choreographer. The Choreographer's primary functionality is to control the temporal mapping of program oper-

ations to visualization actions. In particular, it provides the capability to view animations of the program execution under different logical orderings of program events or operations.

In the next section we expand on the descriptions of these three components and describe our progress to date on each.

3 PARADE Components

3.1 Program Monitoring

To build a visualization of a program execution, adequate descriptive information about the execution is required. For example, a programmer may want to know which processes are active, which functions are being invoked, what the values of variables are, and so on. Often, this program monitoring can be the most challenging problem in building a software visualization.

In order to learn about a program's execution, we must instrument it at some level to transmit tracing information. Hardware level instrumentation is sometimes available, but it is usually too low level except for things such as performance monitoring. Therefore, we rely on software-level instrumentation which can be utilized at many levels such as the operating system, the run-time system, system-supplied libraries, libraries used as alternatives to system-supplied libraries, or in the source code of the program under study. Typically, software level instrumentation is very machine and language specific, however, so building a general monitoring mechanism is unrealistic. In PARADE we utilize techniques that are specific to the machine and language of the intended application, but all these different techniques have some common, general principles.

Perturbation of the program under study is also a side effect of instrumentation. In PARADE we have not focused on the perturbation issue. We simply make an effort to minimize its influence whenever possible using established techniques.

A key issue in program monitoring is whether the software visualization will run on-line (display as the program runs with some relative time delay) or post-mortem (the program produces a trace which is post-processed at a later time). In PARADE our primary method of operation is to use post-mortem visualization with trace files. This is necessary to take full advantage of the Animation Choreographer. It also allows us to minimize perturbation, as we can utilize substantial buffering in our software-level instrumentation.

The techniques for performing on-line visualization in PARADE involve mechanisms to transmit program events to the animation component in a timely fashion. It is more complex than a simple transmit/receive action though. Such an approach breaks down due to transmission latency or lack of synchronization in timestamps across processes. For instance, it would not be uncommon for the animation to receive a *message receive* program event before the corresponding *message send* program event. Our approach uses filters that preserve the causal ordering of program events by applying simple ordering rules to the event transmissions[GEK+94].

To generate post-mortem visualizations with PARADE, the primary method used in the environment, we utilize three different software instrumentation techniques. Below we describe these in decreasing level of programmer involvement.

The most basic way to gather execution information is to have the programmer hand annotate his or her source code with output statements. Typically, a *print* statement is added that will produce a line of output containing the event name or type, a process-id, a timestamp if available and any other event specific parameters. The programmer can have all processes write to one file (contention is a clear problem here) or each process can write its information to a unique file. Because of the direct programmer involvement and amount of work required, this method can be time-consuming and error-prone. Nevertheless, it is the one method that is truly general, and it is the method that can produce the most detail about a program's execution. For example, if a visualization requires the value of a program variable at key points, hand annotation may be the only adequate instrumentation technique.

The second method of software instrumentation we have utilized is to override a standard parallel communication library with a set of replacement macros. For example, KSR machines provide a C library called pthreads that includes basic process control and communication calls such as `pthread_create`, `pthread_mutex_init`, `pthread_barrier_checkin`, and so on. We have developed a set of macros called gthreads that can be used to monitor KSR pthreads programs[ZS95]. In essence, we define a macro for each pthread call that first writes a trace event of that call, then calls the original pthread routine. Because this method can only trace actions that correspond to pthreads routines, we added two supplemental calls, `gthread_enter` and `gthread_back`, that programmers can add to their source. These macros are used to signal function entry and exit, tracing information not available from the basic pthreads calls. The monitoring information from all these macros serves as the input to a visualization package we have developed for KSR pthreads programs. It will be discussed more thoroughly in the next section. The pthreads monitoring macros are available via anonymous ftp from the machine `ftp.cc.gatech.edu` as the file `pub/people/stasko/gthread.KSRtracing.tar.Z`.

The third and least programmer-involved method we have used to gather post-mortem trace information is to actually modify the resident parallel communication library for a system. We utilize this approach with Conch, an experimental heterogeneous network computing system[BFK+94]. Conch contains communication primitives for send-receive communication, barriers, rendezvous, and so on. We have modified the native code of these routines to support run-time activation or deactivation of the trace production[TSS94]. At run-time the programmer simply specifies a command-line flag to turn on tracing. We also have provided a way to gather more "subtle," application-specific trace information from a Conch program, that is, information not available from the communication calls. We have added the routine `c_parade_log()` that a programmer can place anywhere in source code. This routine generates a trace event in a manner similar to a C `printf` statement, but it only works when tracing is turned on.

Our monitoring techniques in Conch also are unique in the addition of logical clock timestamping. Determining an ordering of events across processors is difficult in a distributed system. We have added a logical clock to the system to help alleviate this problem.

The monitoring in Conch occurs with minimal perturbation as well. Program execution times with tracing on are quite close to those without tracing.

All these techniques to extract program execution data have a common element. They produce trace records or events that capture important operations in a program. In PARADE we have developed a specification file format that captures and describes what this

monitoring information will be for a particular program. Below is an example of a simple event specification file.

```
KSR_C
1

INIT:id  INIT:_synch pid:d  ts:d
INPUT:id  pid:d  ts:d  index:d  value:d
READY:id  pid:d  ts:d  index:d
EXCHANGE:id  pid:d  ts:d  from:d  to:d
FORK:id  FORK:_synch  pid:d  ts:d  forkedpid:d
```

The first line always describes the environment (machine and language) of the program, and the second line tells the field in which the event type or name will appear in all the event records. Subsequent lines describe the different event types and detail the trailing arguments of each. Left-hand sides provide parameter names which can be special reserved symbols such as *pid* (process-id) or *ts* (timestamp), or they can be user defined values such as *index*. Right hand sides of parameters specify the type (printf argument style) of the parameter.

We also have created these type of specification format files for the other two primary components of the PARADE environment: the visualization and the event-to-visualization mapping. All three specification files are used as input to the Animation Choreographer when a user generates a software visualization of a program execution. Details of this process will be described later in the report.

3.2 Visualizations

Visualizations in PARADE are built using the Polka animation system[SK92, SK93]. Polka supports color, 2-D visualizations, and in particular, it provides high-level primitives for smooth animation effects. It also supports independent scheduling and execution of animation actions, thus permitting easy design of concurrent animation scenarios.

This last capability is important because of the number of different ways a concurrent program may execute. On one run, an essentially serial ordering of operations may occur, and the animation of the program should reflect this. On another run, a number of operations may occur concurrently (or be thought of as logically concurrent) and the program animation should illustrate this concurrency. Most importantly, the same visualization code should suffice to illustrate both scenarios; the animation designer should not be forced to write different code for each potential scenario. Polka meets these expectations.

Polka provides an object-oriented design model to developers. Animations can include any number of windows or *Views*. Within a View, a designer utilizes *Location*, *AnimObject*, and *Action* objects to implement the animation activities. The focus of the system has been to provide sophisticated graphics capabilities, yet keep the paradigm easy to learn and use. Very expressive, complex animations can be developed with relatively little code.

Polka is implemented in C++ on top of the X Window System and Motif. It is available via anonymous ftp as the file `pub/people/stasko/polka.tar.Z` on the machine `ftp.cc.gatech.edu`. Detailed documentation and example animations are provided with

the distribution. We also have developed a 3-D version of Polka written in GL on Silicon Graphics workstations[SW92, SW93]. We have used it to develop a number of interesting software visualizations of parallel programs as well.

As was done in the program monitoring component, we have created a specification format describing a Polka visualization. A sample visualization specification appears below.

```
View BlocksView qsort.H
Init name:s
Input pid:d ts:d pos:d value:f
Ready pid:d ts:d totalnum:d
Exchange pid:d ts:d pos1:d pos2:d

View Chart qsort.H
Init name:s
Bounds ts:d  pid:d  num:d
Swap ts:d  pid:d pos1:d pos2:d
```

The two sections here define the two different Views (windows) of the animation. This particular animation is the one discussed in the next paragraph. Below each View name (with the file in which the include information appears) are listed the individual *animation scenes* (C++ member functions) provided by the View. The first argument is the scene name and the trailing arguments are the parameters to the scene function.

Many different software visualizations and animations of concurrent programs have been built using Polka. At one level, it is possible to build an application-specific visualization of a particular program by writing the Polka code oneself. For instance, Figure 2 shows a two View animation of a parallel quicksort program. The left View shows the classic algorithm animation style blocks sorting view (we use color to indicate the process responsible for a comparison or exchange of an element) and the right View encodes time along its y-dimension to portray the history of exchanges in the program. This entire animation with smooth motion and potentially concurrent animation effects takes only 175 lines of Polka code.

At a second level, it is possible to use Polka to build a canonical view library for a particular machine or programming paradigm. Then, software developers simply use the library as an application—they write no Polka code themselves.

We have done this for a few different programming paradigms. Figure 3 shows the set of views built for the KSR pthreads package mentioned earlier in the Program Monitoring section[ZS95]. These views highlight the state of threads, barriers, and mutexes; They show where each thread is within the program call graph, and they show a history of the threads over time.

Figure 4 shows the set of views developed for the Conch distributed system also mentioned earlier[TSS94]. Here, views show message communication between processes, the status of different processing elements, and the history of the computation. We currently also are developing a visualization library for the PVM[Sun90] distributed system. It will soon be available via anonymous ftp at the site mentioned at the end of this report.

Figure 5 shows our preliminary work in building a view library for High Performance FORTRAN programs. Individual views here show the processor grid, data distribution,

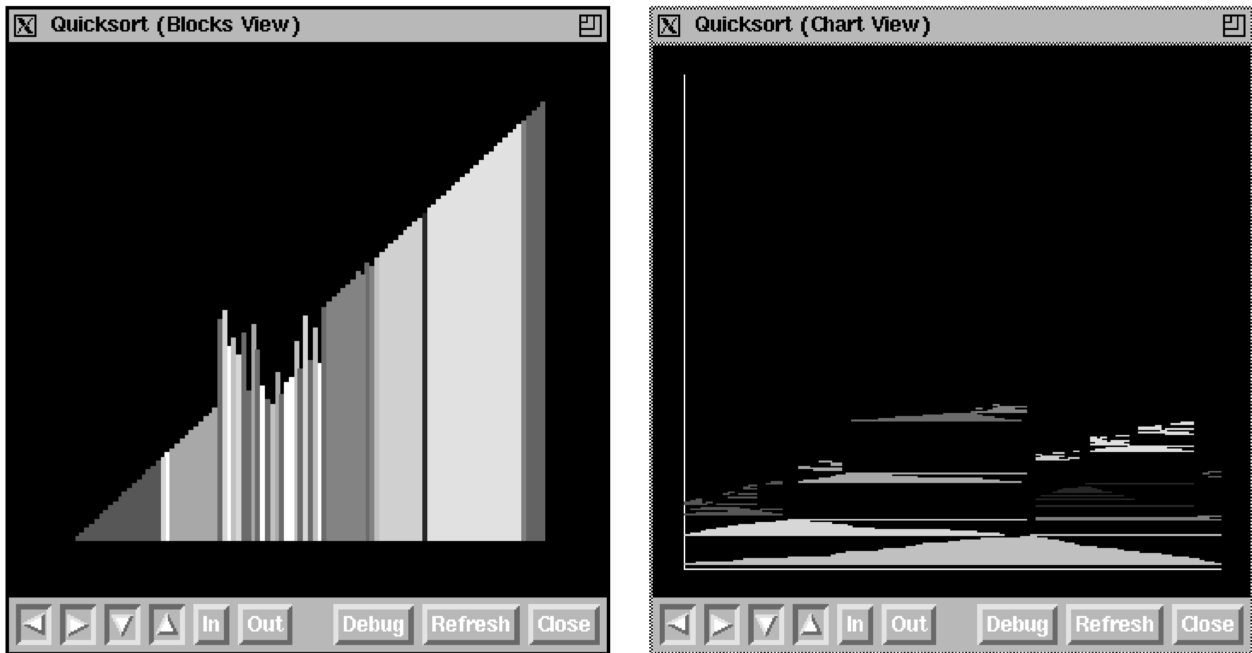

Figure 2: Two Views from an application-specific animation of a parallel quicksort program.

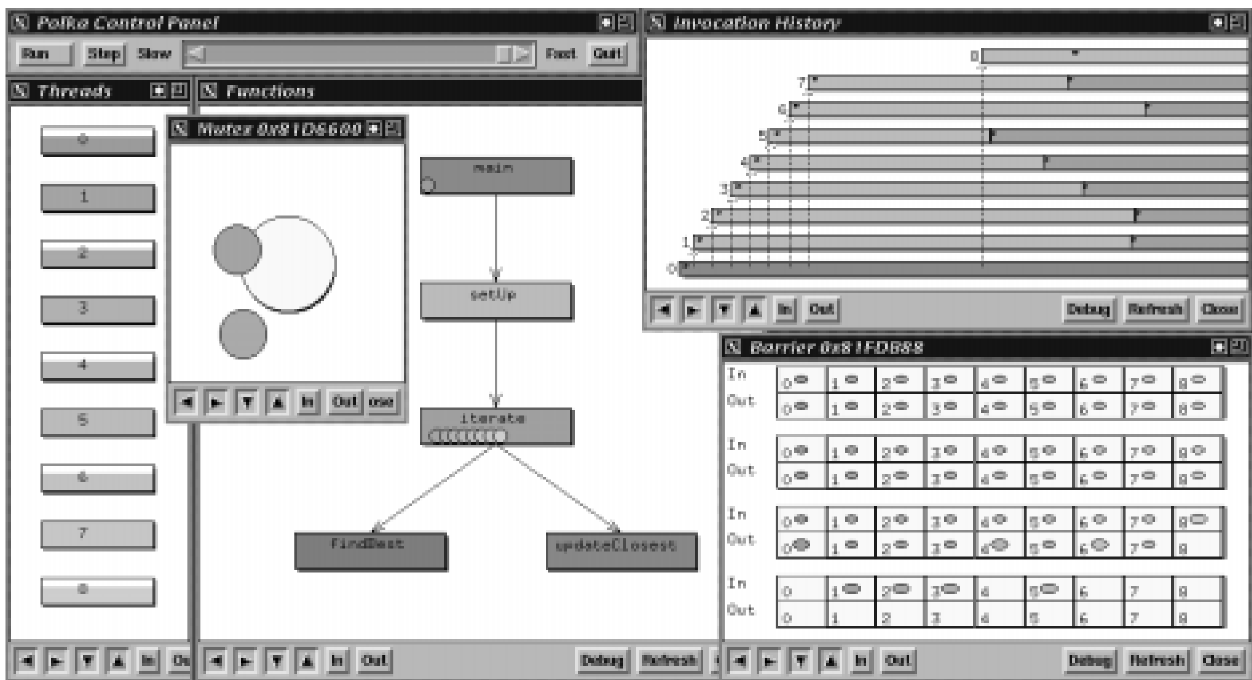

Figure 3: Library of views ued to illustrate KSR pthreads programs.

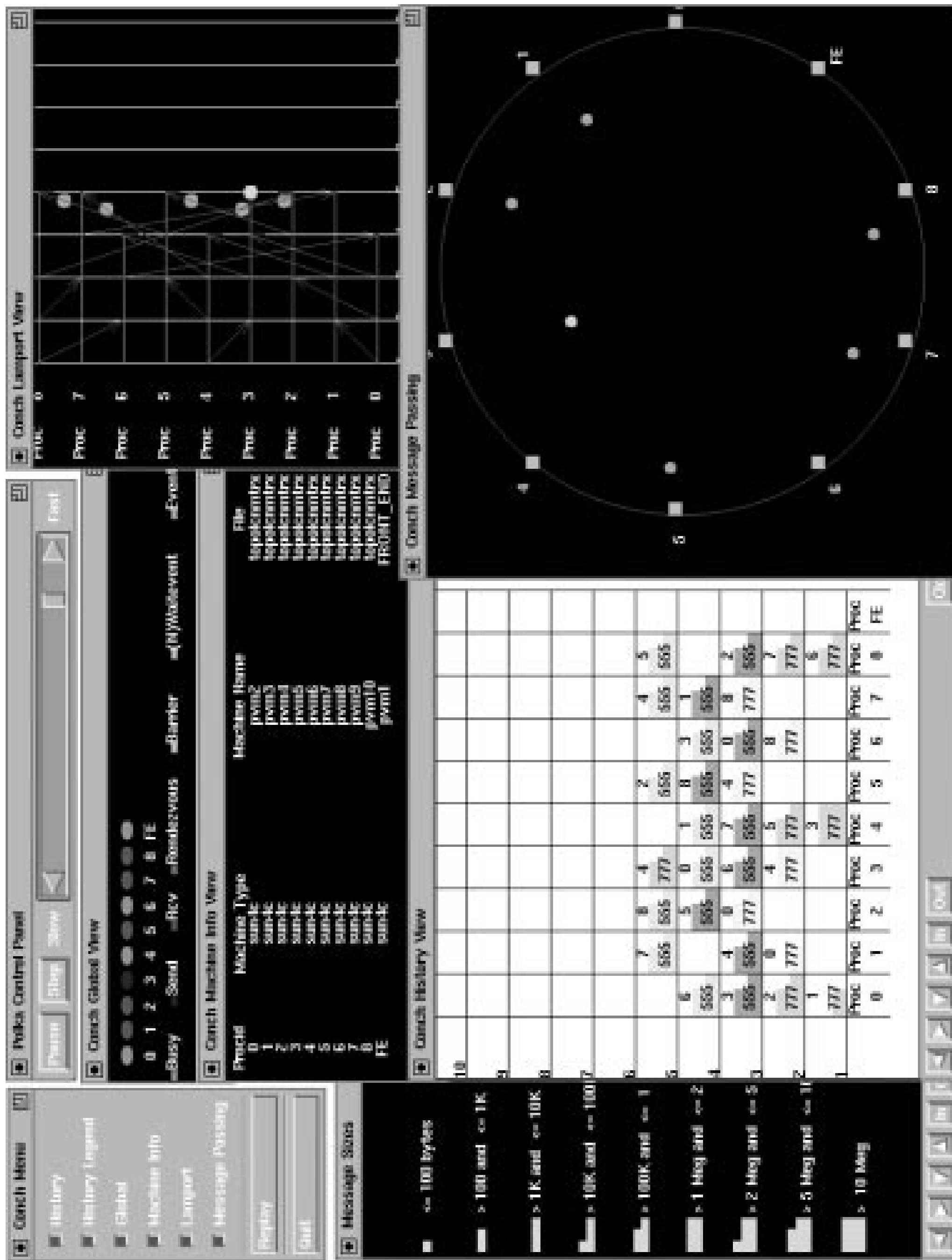

Figure 4: Library of views used to illustrate Conch programs. Particular importance is paid to message sends and receives.

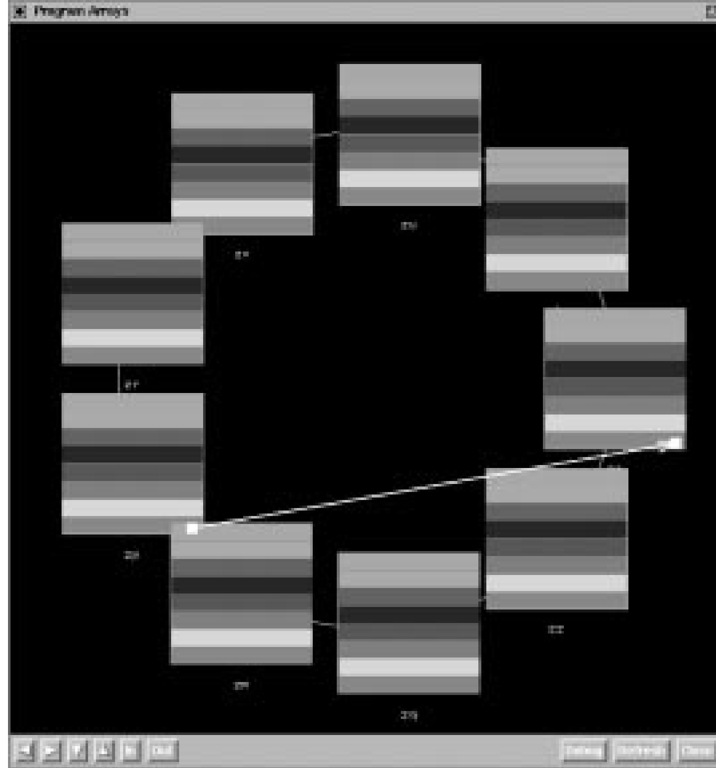

Figure 5: Library of views used to illustrate High Performance FORTRAN programs. This view illustrates important arrays in the program. Color is used to indicate the different processors.

and the important arrays manipulated in the program. Color, image flashing, and arrows are used to indicate data access, movement, and work between the processors.

Finally, views of the state of cthreads programs have been developed using Polka as well[GEK+94].

Moving to the actual design of animations themselves, one key challenge is how to "scale up" the views. That is, how do we visualize very large programs or execution of programs on very large data sets? We utilize a concept called *semantic zooming* that is useful in such situations[MS95]. With semantic zooming, we can present the entire program and its data set within one view by using abstraction and clustering. From there viewers can interactively select graphical objects to zoom in on areas of interest. But rather than zoom in by a straightforward magnification, the view adjusts to the next semantic level and the presentation may change dramatically.

Figure 6 shows the data in a parallel sort of 10,000 elements. Each rectangle represents a contiguous 10% of the values being sorted, and geometric properties of each rectangle depict the maximum, minimum, average, and sortedness of the region. When a rectangle is selected, a new view is shown, focusing only on the selected portion of the array. As less and less elements are shown in the view, the presentation adjusts to provide more detail.

We also have used the Polka-3D system to build visualizations of large programs. In particular, we used natural 3-D perspective and zooming to depict executions of program running on a MasPar machine[WS93].

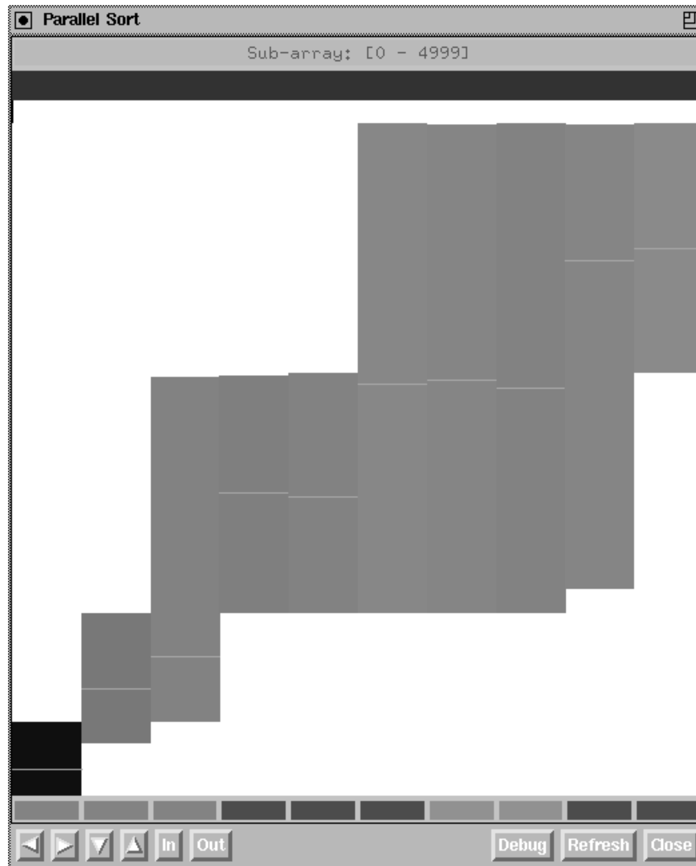

Figure 6: Visualization of sort of 10,000 elements using a semantic zooming technique. This animation allows a viewer to examine characteristics of the entire data set and to zoom in on particular regions of the array.

To further help simplify the development of Polka visualizations, we are currently working on a tool that will allow users to build visualizations without any textual graphics coding. The tool will provide a number of view templates for things such as scalar variable views, grids, graphs, charts, etc. Individual attributes of these views such as size, color, level, and value will be modifiable through point-and-click direct manipulation means. Each attribute will be able to be "attached" to particular values from program events. The designer will be able to interactively specify the mapping between the two.

3.3 Program-to-Animation Mapping

The third component of the PARADE environment is the mapping from program executions to their animations. At a first glance this may seem unnecessary—Whenever an event or a state change occurs in a program, we could simply display the corresponding animation. This solution might be sufficient for serial programs, but we believe it is inadequate for concurrent programs.

One basic problem with this approach is how to decide when to display concurrent animations (program execution events) in a view window. If the timestamp between two events is very small, are they concurrent? Similarly, if two events can logically be thought of as simultaneous (or potentially simultaneous), are they concurrent?

In the PARADE framework, we believe these questions must be answered by the programmer or viewer. That is, we define a number of different temporal perspectives under which an animation of a program execution can be viewed. We provide a system and interface, the Animation Choreographer, that allows a viewer to interactively choose one of these perspectives and to further adjust the perspective interactively[KS94a, KS94b].

The Animation Choreographer of PARADE must know the operations of the program being visualized, the set of available animation views and scenes, and the mappings between these two sets. The first two are described by the event and visualization specification files described in the prior two sections. The third is described by a mapping specification file, an example of which is shown below.

```
INIT -> BlocksView.Init 1 ti 3
INIT -> Chart.Init ti 1
1 -> BlocksView.NewVal 1 ti 4
READY -> Chart.Axes ti 1 @0.0 @10.0
```

Each line defines a program event to animation scene mapping. The first line states that an INIT event in the program should be presented by calling the Init scene of the BlocksView view. The trailing parameters describe which arguments of the program event should be passed to the scene, whose parameters are specified in the linear order in which they appear. Values preceded by the '@' character mean to always pass a literal value to the scene for that parameter. The 'ti' characters mean that the Choreographer should pass the time (when to schedule the animation actions) in as that parameter.

Once the Choreographer has these three specification files, it knows how to proceed with generating animations for this program. We use a Choreographer generator program that reads all three specification files and outputs the source code for a Choreographer appropriate to this particular animation. This source code is compiled together with generic

Choreographer code and the Polka animation code to generate the final binary. At run-time the Animation Choreographer reads the set of post-mortem trace files from a program, then it is ready for interaction. A summary of this framework in PARADE is shown in Figure 7.

The Animation Choreographer portrays a program execution as a directed acyclic graph whose nodes are events. This depiction is presented in Figure 8. Each column corresponds to a process or thread, and time starts from the top of the display and proceeds downward. Dependences between events, such as a send-receive pair, are indicated by an edge between the events.

The *Ordering* menu at the top contains our default temporal orderings of program events. Currently, it includes

- **Timestamp** – Portray the events at times consistent and relative to how they occurred with respect to a global clock.
- **Serial** – Portray a serialization of all the events using their causal order.
- **Minimal distortion** – Portray the events relative to how they occurred in global time, but resolve problems in the logical or causal order.
- **Maximum concurrency** – Portray the events as they would occur to generate maximum concurrency under their causal ordering.

When a viewer selects one of these options, the presented event graph adjusts itself to reflect the selected ordering. Choosing the *Run* option then starts the animation of the program execution under that temporal ordering. When the maximum concurrency ordering is chosen, for instance, the animation displays have many concurrent animation actions occurring at once. When timestamp order is chosen, one often encounters large bursts of animation followed by idle periods, thus mimicking (of course at a slower rate) the actual program execution.

To achieve this functionality we needed to analyze the semantics of the different communication and synchronization operations of the program being examined. Obviously, these primitives vary between machines, architectures, and languages. Currently, the Choreographer "understands" shared memory type primitives as exemplified by cthreads or pthreads (barrier, mutex, condition, etc.) and message passing primitives common on distributed systems such as PVM and Conch. We continue to add more semantics.

Additionally in PARADE we have built a few animations not using the Animation Choreographer to do mappings. Essentially, these animations use a particular hard-wired choice of one of the Choreographer mappings. When a particular perspective is sought and a low-tech solution is sufficient, this approach is reasonable.

4 Using PARADE

In this section we briefly summarize how programmers use the PARADE environment to visualize their programs and code. Let us begin with the case where the program trace events are generated "automatically" (tracing of parallel primitives has been activated through a macro or run-time library approach) and the visualization is predefined. This is the situation with the KSR pthreads and the Conch visualizations discussed in prior sections. In this

Figure 7: Overview of how the Choreographer fits within the PARADE environment.

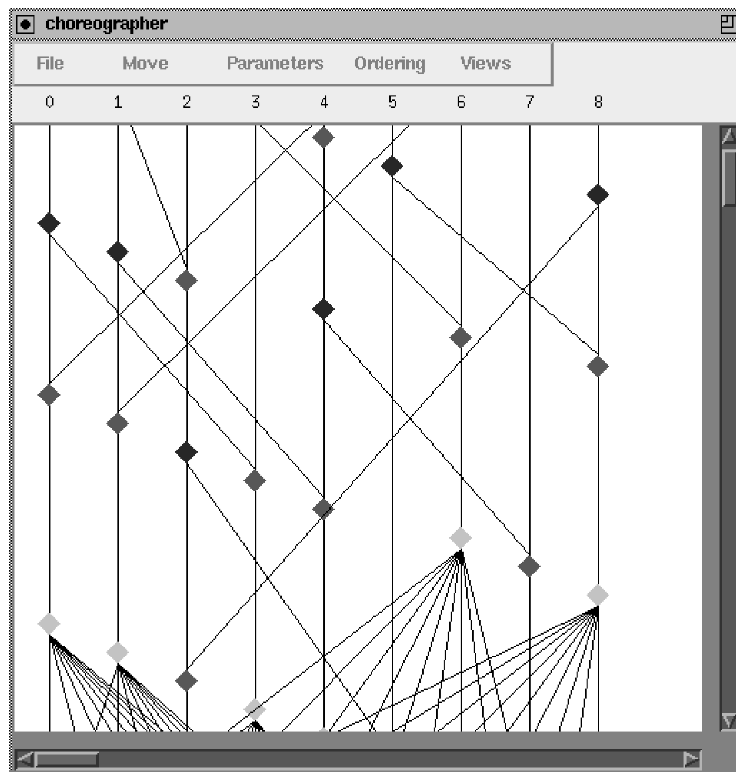

Figure 8: User interface for the Animation Choreographer that presents the ordering and constraints between program execution events.

case, Animation Choreographers for each of these programming environments can be pre-developed since all three specification files are known a priori. Consequently, a programmer simply runs his or her program, gathers the trace files, and then invokes the appropriate Choreographer with the trace files' name as an argument. The Choreographer starts up, displays the graph interface, and allows the viewer to interact with it in order to view animations of program executions under different temporal orderings.

Now consider a scenario in which a programmer is building an application-specific, algorithm animation style visualization of a program. Here, the programmer must generate semantic events beyond the simple parallel primitives, and the programmer hand codes the visualization with Polka. In this situation, the programmer carries out the steps below to generate a PARADE visualization.[1] Figure 7 can be used as a helpful summary of this process also.

1a. Design and implement the Polka animation views and scenes for the visualization. Compile this code to make an object file.

1b. Annotate the concurrent program with output statements so that it generates the desired trace events. Run the program and gather the trace files.

2. Create the program event, visualization, and mapping specification text files.

3. Run the Animation Choreographer generator program with the three specification files as input. It will generate source code for the application-specific Choreographer used in the next step.

4. Compile the Choreographer source code and link it with the Polka scenes code from step 1a and the generic Animation Choreographer object code in order to create the Choreographer binary.

5. Run the Animation Choreographer, giving it the trace files from step 1b as input. Now the viewer interacts with the interface and watches animations.

Clearly, this last scenario is involved enough so that it is impractical for day-to-day debugging chores. Rather, it is useful when a particular program requires detailed study or a person wants to prepare a visualization as an explanatory instructional aid. The first scenario in which a programmer simply runs their program and then invokes a pre-built Choreographer is appropriate for program testing, debugging, and optimization chores, we believe.

5 Conclusion

In this article we have described the current status of the PARADE environment for visualizing concurrent program executions. PARADE consists of three primary components: Program monitoring and tracing, a visualization system, and the mapping from program actions to their visualizations. Currently, a number of different projects are underway in each of these components. This report has provided a brief summary of those efforts and it also serves as a reference on where to acquire more detailed information about them.

[1]Lettered steps within one number signify that they can be done in any order. The numerical steps must be carried out in the specified order.

6 Acknowledgments

PARADE has been and is the cumulative effort by a group of people. Bill Appelbe, Charles Hardnett, Eileen Kraemer, Song Liang, Jeyakumar Muthukumarasamy, John Stasko, Brad Topol, Joe Wehrli, and Alex Zhao all have contributed to the project. PARADE has been supported by the National Science Foundation under grant CCR-9121607, Kendall Square Research and an Intel Graduate Fellowship. For up-to-date information on the status of PARADE and publications relating to it, please examine the World Wide Web page `http://www.cc.gatech.edu/gvu/softviz/parviz/parviz.html`. All ftp-able software mentioned in this article can be found on the machine `ftp.cc.gatech.edu` in the directory `pub/people/stasko`. All GVU technical reports referenced here can be acquired via ftp also, on the same machine in the directory `pub/gvu/tech-reports`.

References

[BFK+94] Doug Bowman, Adam Ferrari, Melisa Kelley, Brian Schmidt, Brad Topol, and Vaidy Sunderam. The Conch network concurrent programming system. Technical report, Emory University, Atlanta, GA, January 1994.

[CE93] Thomas L. Casavant (Editor). Special issue on tools and methods for visualization of parallel systems and computation. *Journal of Parallel and Distributed Computing*, 18(2), June 1993.

[Che93] Dorren Y. Cheng. A survey of parallel programming languages and tools. Technical Report RND-93-005, NASA Ames Research Center, Moffett Field, CA, March 1993.

[GEK+94] Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan, John Stasko, Jeffrey Vetter, and Nirupama Mallavarupu. Falcon: on-line monitoring and steering of large-scale parallel programs. Technical Report GIT-CC-94-21, College of Computing, Georgia Institute of Technology, Atlanta, GA, April 1994.

[KS93] Eileen Kraemer and John T. Stasko. The visualization of parallel systems: An overview. *Journal of Parallel and Distributed Computing*, 18(2):105–117, June 1993.

[KS94a] Eileen Kraemer and John T. Stasko. Toward flexible control of the temporal mapping from concurrent program events to animations. In *Proceedings of the 8th International Parallel Processing Symposium (IPPS '94)*, pages 902–908, Cancun, Mexico, April 1994.

[KS94b] Eileen Kraemer and John T. Stasko. Toward flexible control of the temporal mapping from concurrent program events to animations. Technical Report GIT-GVU-94/10, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, March 1994.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[MS95] Jeyakumar Muthukumarasamy and John T. Stasko. Visualizing program executions on large data sets using semantic zooming. Technical Report GIT-GVU-95/02, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, January 1995.

[PC94] Cherri M. Pancake and Cutis Cook. What users need in parallel tool support: Survey results and analysis. In *Proceedings of the Scalable High Performance Computing Conference (SHPCC '94)*, pages 40–47, Knoxville, TN, May 1994.

[SAK91] John T. Stasko, William F. Appelbe, and Eileen Kraemer. Utilizing program visualization techniques to aid parallel and distributed program development. Technical Report GIT-GVU-91/08, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, June 1991.

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

[SK93] John T. Stasko and Eileen Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, June 1993.

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

[Sun90] V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2(4):315–339, December 1990.

[SW92] John T. Stasko and Joseph F. Wehrli. Three-dimensional computation visualization. Technical Report GIT-GVU-92-20, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, September 1992.

[SW93] John T. Stasko and Joseph F. Wehrli. Three-dimensional computation visualization. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, pages 100–107, Bergen, Norway, August 1993.

[TSS94] Brad Topol, John T. Stasko, and Vaidy S. Sunderam. Integrating visualization support into distributed computing systems. Technical Report GIT-GVU-94/38, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, October 1994.

[Tuf83] Edward R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, Cheshire, CT, 1983.

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

[WS93] Joseph Wehrli and John Stasko. Interactive three-dimensional visual debugging in massively parallel computation (extended abstract). In *Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 235–237, San Diego, CA, May 1993.

[ZS95] Qiang A. Zhao and John T. Stasko. Visualizing the execution of threads-based parallel programs. Technical Report GIT-GVU-95/01, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, January 1995.