Integrating Visualization Support Into Distributed Computing Systems

Brad Topol, John T. Stasko
Graphics, Visualization, and Usability Center
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

Vaidy Sunderam
Department of Math
and Computer Science
Emory University
Atlanta, GA 30322

Abstract

Visualization and animation tools may become extremely important aids in the understanding, verification, and performance tuning of parallel computations. Presently, however, the use of visualization has had only a limited use for enhancing parallel computation. We hypothesize that one of the primary reasons for the limited use of visualization tools in parallel program development is the difficulty of acquiring the information necessary to drive the visual display. Our approach to this impediment focuses on integrating visualization support directly into a distributed computing system. Central to this integration is the addition of a logical clock that prevents the timestamps of events from violating causality. The implementation requires the "piggybacking" of a negligible amount of extra header information on system messages and the impact on performance is minimal. This results in a system that produces useful visualizations with no extra effort required by the applications programmer. Also integrated into the distributed system is support which simplifies the creation of programmer-defined, application-specific visualizations, unique to each new parallel program developed.

1 Introduction

In recent years, parallel and distributed processing have provided effective solutions to many challenging computational problems. Unfortunately, understanding and analyzing the execution of concurrent programs is a difficult task for most computer scientists. A number of ways to address this problem have been proposed, one being the use of visualization and animation tools. By illustrating and helping to explain program behavior, visualization tools have the potential to make an impact in not only the understanding of a parallel computation, but also in verification and performance tuning. Until now, however, the use of visualization has had only a limited impact on concurrent program understanding, verification, and performance tuning. The systems that have been created are mostly research prototypes and few have achieved wide usage[6].

Visualization's limited impact is an enigma. It is clear that an appropriate picture or image can convey the same amount of information as hundreds or thousands of lines of text[15]. Certainly, computer graphics hardware and software environments have matured and are now widely available, so they are not culpable. We hypothesize that one of the primary difficulties limiting the impact of visualization on parallel computation is the difficulty of acquiring the information necessary to drive the visualization display. Visualizers are ready with informative views and animations, but finding out *what* to present and *when* to present it remains a challenge to them.

Visualization tools rely on trace events, also referred to as annotations, that record the interesting aspects of a distributed system. These records are interpreted by the visualization tool to produce the graphical presentation. Because concurrent systems are built without a focus on subsequent graphical presentations, software visualizers are given very little support with regards to trace event profiling. This is a major impediment to the production of meaningful visualizations, but it need not be the case. Support for visualization, namely event tracing support, need not be an afterthought. Instead, it should be a vital design issue considered when developing a distributed or parallel system.

This article discusses our approach of integrating visualization support into an experimental heterogeneous parallel computing system. This approach results in a system that produces useful visualizations with no extra effort required on behalf of the applications programmer. Further, system support is provided that simplifies the creation of programmer-defined, application-specific visualizations which are unique to each new program developed.

2 Design Goals

The primary goal of this project is to integrate as much direct system support into a distributed computing system as possible in order to facilitate visualization of parallel computation. Typically, programmers must hand annotate their code with print statements to produce an event log for visualization. This activity is error prone and time-consuming; it requires programmer effort that would better be directed at evaluating program performance. Hand annotation also may not be able to produce an event trace of sufficient detail, thus further compounding the issue.

Another problem with this approach involves trace events that are timestamped by local clocks that are not accurately synchronized, thus leading to misleading visualizations. For example, we might discover a message receipt with a timestamp whose clock value is earlier than the timestamp clock value of the message send. Visualizations that use trace data filled with causality violations like this are clearly misleading. Creative techniques used during postprocessing such as those in [2] can produce makeshift solutions to some impediments that visualizers face. Nonetheless, this is another example of visualizers expending excessive effort to solve problems that are avoided with proper system support. To address this problem, we have developed the following requirements for facilitating production of visualizations of concurrent systems:

- **Minimal Effort**—Currently, too much extra effort is required to integrate visualization support into a parallel system. There are two primary causes of this. First, for a parallel application to be visualized, it must be annotated to provide information to "drive" the visualization. One method of supplying annotation is to directly modify a parallel application. For large applications, the modifications can be extensive. Environments such as PABLO[10] provide a graphical interface for interactively specifying annotations, but extra effort is still required.

  Our approach minimizes the need for annotation by having the distributed system automatically generate the annotation whenever possible. This is performed by integrating annotation support directly into the distributed system primitives. For example, the *Send* primitive is cognizant of what type of message is being sent, to whom the message is being sent, and so on. Therefore, the routine produces the trace event information necessary for visualization automatically, instead of the parallel application being annotated by hand. A similar approach is used by the ParaGraph[5] system, as its information is derived from PICL[4], a portable instrumentation communication interface. Our approach goes one step further, however. By truly integrating into the system itself, all system primitives can provide automatic annotation generation. This is in contrast to PICL, which can only provide annotations for the generic communication primitives that it supports. For example, some systems may have system specific routines such as a receive that utilizes timeouts or special synchronization constructs that simply aren't supported by a generic portable communications library. These type of events would require some other method of annotation such as annotating by hand. Hand annotation in our system is now only necessary for the visualization of application-specific information, and new system contructs are provided to make this as easy and straightforward as possible. This aspect will be more thoroughly discussed later.

  The second main reason why extra effort is required is that the visualization itself often needs to be created by the programmer. If the amount of time and effort required to produce a visualization is too extensive, developers simply will not use visualizations. It is therefore necessary to minimize this effort as much as possible. We address this problem by including many default, general purpose views with the system. Further, the effort to create special application-specific visualizations is minimized via the support of an object-oriented parallel visualization design toolkit.

- **Flexibility in Visualization Support**—A system should only provide support for visualization on demand. Further, the application programmer should be able to decide at execution time whether or not visualization support will be activated. This aspect, a result of a system integrated approach, is more desirable than the alternative of having to relink a program to a special library whenever visualization is desired.

  This is the approach we adopt for visualization support. Visualization options are available directly from the command line prompt of the distributed system. Therefore, the choice of visualization support can be deferred until application execution.

- **Minimal Perturbation**—Adding visualization support invariably will impact the performance of the underlying system being examined. It is unrealistic to expect to produce visualizations without incurring some performance degradation of the underlying program, but efforts should be made to minimize this perturbation as much as possible. By integrating the visualization support directly into the system, we remove extra overhead that would have occurred had support been implemented at the user level.

- **Timing Support**—It is advantageous for the program monitor to provide temporal information about the execution. Ideally, each program event would be timestamped from a perfectly synchronized real-time global clock. This is clearly unrealistic. What is desirable, however, is a logical clock as described in [8] that provides causality ordered timestamps needed to produce accurate visualizations. Also desirable is the addition of a real-time local clock timestamp to provide information for performance visualization. Our approach provides both these alternatives.

These requirements have driven our program visualization support within a recently developed distributed system. The result is a system that automatically produces several graphical views, and is easily augmentable with custom views created by an applications programmer with little or no computer graphics background. In the next section, we describe our framework in more detail.

3 System Foundations

We chose to provide visualization support and graphical views of programs written in Conch, an experimental system for heterogeneous parallel computing. Our visualization framework used POLKA, an object-oriented software visualization methodology and library.

3.1 Distributed Computing System—Conch

The Conch system[3] is an experimental heterogeneous network computing system. Like its predecessors, Conch is a framework for parallel distributed computing, allowing a network of Unix workstations to function as a single parallel computer. Conch provides features such as custom configurable machine topologies, fault tolerance, and a shared system-user context that allows the system to execute with less system overhead. The Conch system is based on two popular and established high performance concurrent computing systems. The first of these, PVM[13], is well regarded as a leading system for writing distributed and parallel applications. PVM is in widespread use and is considered by many to be the de facto standard for network computing. The second influence on the development of Conch is EcliPSe[14], a system used mainly for high performance concurrent simulation. The following is a brief introduction to the facilities provided by Conch.

3.1.1 Virtual Machine Configuration

One of the first choices a Conch application programmer must make is the type of virtual machine to be configured. This is accomplished by means of a host file that supplies the Conch system with the necessary information to produce the virtual machine. Figure 1 shows an example host file. In it, the programmer has specified a tree topology whose left child is a mesh topology and whose right child is a ring topology. The machines are given numeric identification numbers ranging from 0 to $n-1$ where n is the total number of machines listed. Communication such as message passing is implemented by sending a message to a specified identification number, (hereafter referred to as the node's *c_procid*). On labss1, the program `/home/topol/compute` will be instantiated. On labss2 through labss5 `/home/schmidt/mult` executes. Further, the machines that utilize the ring subtopology utilize the binary `/vss/particle`. Finally, a front end is connected to labss1 as it is the first machine listed in the host file. The user can also choose what program should be run on the front end. The front end is given a *c_procid* of n. Figure 2 is a graphical illustration of such a host file.

```
#TREE
 labss1 file=/home/topol/compute  /*root node*/
 -#MESH
  labss2 file=/home/schmidt/mult
  labss3 file=/home/schmidt/mult
  labss4 file=/home/schmidt/mult
  labss5 file=/home/schmidt/mult
 #END
 -#RING
  nirvana file=/home/vss/particle nprocs=4
  ibms1.fsu.scri.edu file=/vss/particle
  ibms2.fsu.scri.edu file=/vss/particle
  ibms3.fsu.scri.edu file=/vss/particle
  ibms4.fsu.scri.edu file=/vss/particle
 #END
#END
```

Figure 1: Example of a typical Conch host file.

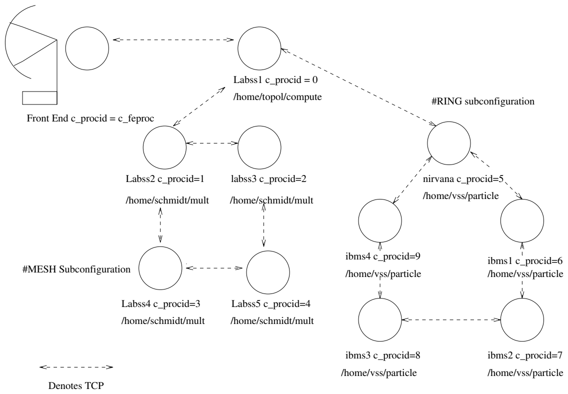

Figure 2: Graphical illustration of a typical host file.

3.1.2 Application Skeleton

Figure 3 illustrates code for a straightforward Conch program. The first Conch library routine in any Conch program is `c_setup()`. Upon calling `c_setup`, the host file is read, and the system instantiates processes of the specified programs on the specified machines. After all nodes finish their initial startup responsibilities, each node is free to begin executing its code as determined by its *c_procid*. Typically, as shown in Figure 3, the front end, denoted by *c_feproc*, sends data to the other nodes. The subordinate nodes receive this data and perform their necessary computations, usually determined by their *c_procid*. After performing such computations, the subordinate nodes send their results back to the front end. The front end typically combines and prints these results. At this point, all nodes perform a `c_exit()` and the Conch system cleanly shuts down. Because the Conch system is in final development,

```
main()
{
   c_setup();        /*front end starts up*/
   ...               /*all nodes start up*/

   if (c_procid == c_feproc) {
      c_send();
      ...            /*wait for results*/
      c_recv();      /*combine and print results*/
   else {
      c_recv();      /*get problem data*/
      .......................................
      :                                     :
      :  User Computation  /*do computation*/ :
      :.....................................:
      c_send();      /*send back results*/
   }
   c_exit();         /*exit Conch system*/
}
```

Figure 3: Skeleton code for a simple Conch program.

it is still possible for visualization to be considered a design issue. The result is a system with substantial internal support for visualization; support that simply cannot be duplicated by systems in which visualization is merely an afterthought.

3.2 Visualization System—POLKA

The visualizations described in this article were built using POLKA[12], a library providing color, 2D animation support on top of the X Window System. POLKA's operations and primitives have been designed especially to help illustrate the concurrency of distributed and parallel programs. POLKA is the presentation component of a comprehensive framework called PARADE[11] we are building for visualizing concurrent systems. For Conch, we have developed a library of default views; they are described later in section 5.

4 Monitoring Framework

Monitoring support required for visualization is integrated into Conch via direct modification of the Conch system routines. This section discusses the modifications we have made to allow for straightforward visualization of the system. It should be emphasized that the focus of our modifications was to support visualizations used for understanding and verification of parallel program execution. This includes modifications to produce automatic visualizations for general purpose use and modifications to facilitate support for custom, application-specific views. It does not include support for performance visualizations, however. This aspect will be addressed in future work.

In order to provide automatic support for visualization, the Conch system needed to generate suitable trace event logging. We used an approach in which each processing element generates its own trace of events. We modified the Conch communication and synchronization primitives so that they automatically log their events to the proper trace event file.

When a Conch program is run, the system prompts the user to specify which system options are to be utilized. From this prompt, the application programmer has the option of initiating the trace event logging (we call it PARADE logging). This involves setting the PARADE loglevel, supplying a clock level, and supplying a filename. Presently, the system implements logging for all communication and synchronization primitives, but several levels of logging will be added to support different granularities of trace event information. Two clock levels are currently supported. These are a logical clock and a real time system clock. Other hybrid clocking methods will be added as necessary to support future work such as performance visualization. The filename supplied by the user is used to create the group of log files. Log files are differentiated by appending the processing element's process identifier to the user provided filename. It is also possible to designate all these command-line parameters by using the *CONCHENV* environment variable, a Conch feature more thoroughly discussed in [3].

Once a Conch application has terminated execution, the programmer can use a tool, *viztrace*, we have created that reads the trace files and produces the visualization. The *viztrace* tool provides options that allow a user to define new application event types in addition to the standard Conch primitive event types. Further, *viztrace* can be augmented with application-specific POLKA animations that are created by the application programmer.

The following are more detailed descriptions of some of the more interesting and challenging Conch system modifications that were necessary for meeting the requirements posed earlier.

4.1 Logical Clock

One fundamental system enhancement made to Conch was the addition of a Lamport logical clock capability as described in [8]. This is a well known method for providing timestamps that preserve causality in a distributed system, and we used it to add logical clock timestamps to the Conch event traces. As alluded to earlier and discussed thoroughly in [2], if various system clocks are used for timestamping, lack of synchronization in the clocks will result in obvious causality violations. By augmenting the system with a logical clock and timestamping trace events with it, the logging can now be used to support visualizations that more accurately depict the execution of the parallel computation. The logical timestamps guarantee that the visualization is representative of a plausible ordering of events, but other plausible orderings may exist also. In development, as described in [7], is a tool that "choreographs" the visualization of many of the different possible *plausible* orderings.

The logical clock is a prime example of the benefits of our system integrated approach; implementing a logical clock in the system was relatively straightforward. This differs substantially from the approach of having an external monitoring tool attempt to implement the logical clock, which can be problematic and adds computing overhead better directed to other tasks. For example, Xab [2], a monitoring tool for PVM programs, implements a similar logical clock in both its online monitoring and postprocessing modes. In the online monitoring mode, Xab logical timestamping support for the PVM barrier synchronization primitive could not be done efficiently without having the PVM developers modify the system directly. For offline post-processing, Xab was able to provide logical timestamping of all PVM primitives by creating a directed acyclic graph of the event trace and traversing this graph in topological order. The complexity of the algorithm used is $O(E * P)$ in the worst case, where E is the number of events and P is the number of processors. With the current trend in parallel processing toward using many processors that produce hundreds of thousands of trace events, having the monitoring tool solely responsible for providing causality preserving (i.e. logical) timestamps requires extensive overhead. When visualization depends on such extensive overhead, its benefits may not outweigh the cost required to provide it. By considering visualization as a system design issue, this extensive overhead is removed, and along with it, a serious impediment to providing visualization.

4.2 Communication Modifications

The primary modification we made to the message passing primitives was the "piggybacking" of the Lamport clock value onto system messages. One possible approach to this problem was to place all responsibility on the user and require that the application must not only correctly implement the clock, but augment messages with a clock value also. If the application had already been written, this would have necessitated a large number of modifications, and may even have required debugging. Clearly, if such extensive modifications are necessary to produce visualizations, they will not happen.

This does not have to be the case, however. By modifying the Conch system *Send* and *Receive* routines and the internal message headers, we were able to quickly add automatic support for our needed logical clock. Our methodology uses the algorithm described in [8]. Briefly, upon receipt of a message, the local logical clock is updated to a value greater than the maximum of the "piggybacked" timestamp and the current local clock value. This necessitated increasing the internal message headers by only four bytes, and the system performance penalty for this extra feature is negligible. We also added the ability to turn this logical clocking on/off at the standard Conch system prompt. When the user does not require the logical clock, he or she simply disables it and no extra performance overhead is incurred. That is, the logical clock value is not used nor is it placed in the internal message headers. This is a recurring benefit of all the modifications used in our system integrated approach: a user decides at execution time whether or not the visualization support is active. This is a clear advantage over relinking to special libraries or adding extra header files as necessary in systems such as Xab[1]. Every effort is made to insure that support needed for visualization does not hinder or delay the application programmer. It simply should be there on demand.

4.3 Synchronization Modifications

The Conch system provides common synchronization primitives such as barriers, Ada style rendezvous, and the wait-event construct. Each of these system primitives are implemented using Conch's internal message passing primitives. Again, to produce a correctly ordered visualization, each process must log when it enters and when it leaves a synchronization construct using timestamps that do not violate causality. This can be done by utilizing Lamport's timestamping techniques. All that is required is the ability to compare the various local clocks of processes involved in the synchronization construct, and to set them all with a clock value that is slightly greater than the largest local clock value. In executing the synchronization primitives in the Conch system, empty messages are used to provide the necessary synchronization. The comparison of clock values is performed by "piggybacking" clock values onto these empty messages. Again, the penalty for providing the needed information is an extra four bytes per message for the processes clock value. Like our other event logging methods, this penalty is only incurred should the user choose the visualization feature from the system prompt.

4.4 Custom Annotation Support

To construct application-specific animations of programs (take for example one that shows the particles as they bounce around in a particle chamber simulation) we need more monitoring information than is available via standard Conch system primitives. In the particle chamber example, for instance, we need particle-to-wall and particle-to-particle collision events. Consequently, we added support for the program annotation needed for custom application-specific visualizations. The addition of the *c_parade_log()* routine to Conch allows the user to augment the trace files with application-specific data, in a format similar to standard `printf()`. This data may take the form of integers, doubles or strings. The data is logged and prepended with a logical timestamp and the process identifier. The user may now add new POLKA routines that support these new trace events and the data associated with them. This unfortunately involves the necessary step of having to augment the application code with these annotations. The annotations do not have to be *repeatedly* added and removed, however, because no action is executed unless visualization logging support is requested from the Conch system prompt just as for the other Conch routines.

4.5 Trace Logging Granularity Selection

Another feature that has been implemented, though not yet fully exploited, is the ability to select the granularity of the type of visualization trace events that are logged. Currently, the user is allowed to enter a visualization *loglevel*. Different levels may refer to whether only communication primitives are logged or whether only user annotations are logged or some combination thereof. We expect to implement the different types of trace file filtering once we have more experience with the way current tracing is utilized.

5 Conch Graphical Views

Figure 4 illustrates our general purpose views for the Conch system.[1] These views are our initial effort; we are currently in the process of constructing more views to fully visualize every aspect of the Conch system. Also, feedback gathered from users of the views will be utilized to modify and improve their content.

[1] Color pictures of the views are available at http://www.cc.gatech.edu/gvu/softviz/parviz/conch/conchviz.html.

At the top of Figure 4 is the standard POLKA control panel. This window allows the user to adjust the speed of the visualization, pause it, or step through it. The upper left view is the Conch machine information view. The first column of this view contains system process identifiers. The second column contains the machine type upon which each process is executing. The host name of each machine is located in the third column. The fourth column displays the name of the binary that the process is executing.

The middle left view is the message passing view. All the processes are laid out around the outside circle. Messages are represented as circles that are proportional to message size which smoothly move into the center of the ring of processes when sent. When a process receives a message, the message moves from its position in the center of the ring to the process. Messages that are never delivered conspicuously remain in the center. Also, the smooth animation of message traffic helps portray actions such as message broadcasts.

The view in the middle right is the Conch history view. This scrolling view maintains a history of messages that have been sent or received. The Y-axis of this view is labeled with Lamport clock values with time proceeding from bottom to top. The X-axis is labeled with process identifiers. Squares that are associated with message sends are filled in with dark colors. Lighter colors of the same hue fill squares that are associated with the corresponding message receives. Message size is encoded by filling squares with various amounts of color. This can be deciphered by using the message size view, which is shown to the left of the history view. It depicts the current scale of color fill for various message sizes. Back in the history view, the message type is placed in the lower center of each square. The number in the top center of each square can represent two different items of information. If the square represents a message send, then this number denotes the process identifier to whom the message is sent. In squares associated with message receives, this number depicts the process identifier from which the message was received.

The bottom view of Figure 4 is the Conch global view. In it, each process has an associated oval that is color-coded to indicate the status of the process during execution.

The upper right view of Figure 4 is a snapshot of our Lamport view. In this view, the Y-axis is labeled with process identifiers and the X-axis is labeled with Lamport clock values. When a message is sent, a circle appears at the appropriate logical time coordinate. Varying circle radii are used to denote message size and the color of the circle is the same as that used in the history view. When a message is delivered, an arrow "grows" from the coordinate of where the message was sent to the correct Lamport delivery time on the receiver's timeline. Simultaneously,

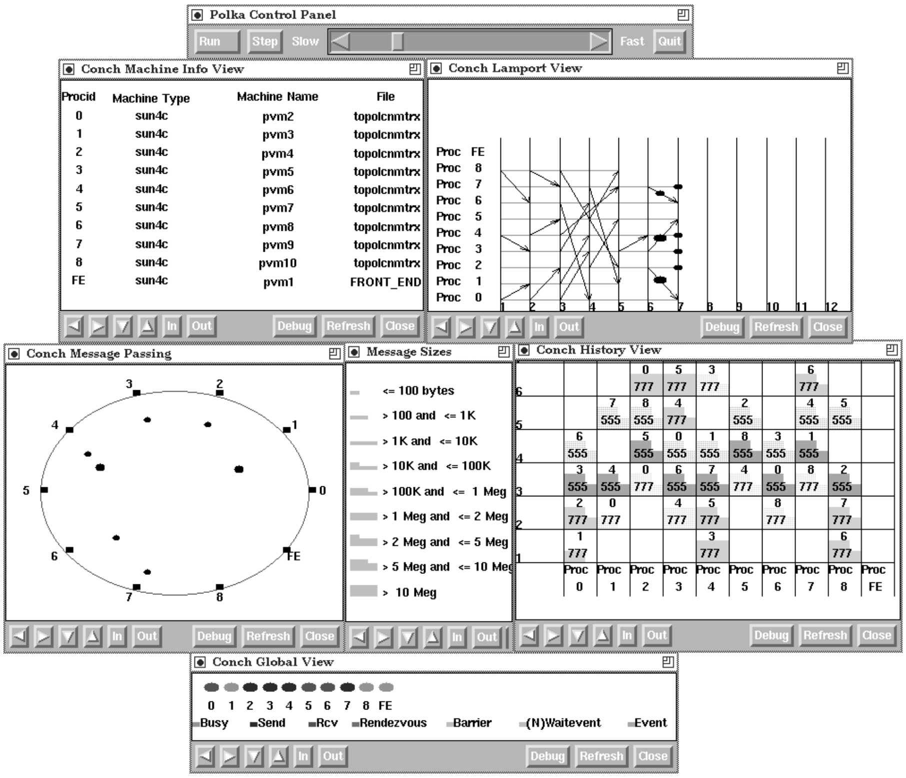

Figure 4: Set of default Conch graphical views.

the circle representing the message moves along this path and then disappears. This view is very useful when trying to distinguish communication patterns.

It is worth noting that the snapshots provided do not do justice to the animations provided by the views. Our views provide concurrent animation, i.e. events that happen concurrently are visualized concurrently. We feel that views which only portray a serialization of a concurrent program execution fail to convey critical information to the viewer.

6 Related Work

A fairly substantial amount of work has been done in the visualization of parallel programs[6]. The ParaGraph[5] system, perhaps the best know application of visualization to parallel programming, provides general purpose visualization support for performance analysis and tuning. ParaGraph utilizes the PICL[4] trace file format. In order to use ParaGraph, a parallel program must use the PICL communication facilities, or emulate their trace format. This implies that a user may only visualize information that exists in the PICL trace files. We believe our system to be more extensible, as the user may add application specific trace events. The user may not only produce new visualizations, but may also visualize new information that did not exist in standard trace events. Further, with our system integrated approach, we are in a strategic position to develop system-specific visualizations. As systems become more complex, these can be crucial to discerning what is happening in the distributed system.

The Pablo[10] environment also provides visualization support for performance analysis and tuning. Pablo primarily derives its event tracing by instrumenting source code. A graphical interface is provided for the specification of instrumentation points. Pablo provides a data analysis environment for reducing, analyzing, and presenting performance data, and is portable to various systems.

Xab[2] provides support for program understanding and debugging. Xab monitors PVM applications by providing macros for the standard PVM library routines. The macros generate trace information, and then call the normal PVM functions. Our approach differs from Xab's as we have integrated all our necessary support directly into the distributed system. This allows us to cleanly provide novel features such as using logical event timestamping and the choice of monitoring support options at execution time.

Another system that has an approach similar to ours is pC++[9]. The pC++ system is a language extension to C++ coupled with a runtime system that provides a framework for data-parallel computing. pC++ provides substantial integrated support for event tracing, and pC++ system specific analysis and visualization tools are under development.

7 Conclusion

Visualization of parallel programs has the potential to make an impact in the understanding, verification, and performance tuning of parallel programs. Unfortunately, impediments exist to producing useful visualizations, but many can be removed with direct visualization support from a distributed system. We have augmented a recently developed distributed system, Conch, to produce tracing information for driving general purpose, automatic visualizations and support to facilitate development of custom application visualizations. Our approach has shown the viability of integrating visualization support into a distributed system with a minimal impact on performance. Furthermore, it would be straightforward for the developers of other distributed computing systems to incorporate our techniques.

Currently, we are focused on adding further general purpose visualizations and animations, and complete integration into all aspects into the Conch system. Also, we are working on system integrated support for performance visualization and system support for on-line visualizations.

Acknowledgments

This research was supported in part by the U. S. Department of Energy grant DE-FG05-91ER25105, the National Science Foundation grants NSF ASC-9214149 and CCR-9121607, and the Office of Naval Research grant N00014-93-1-0278.

References

[1] Adam Beguelin, Jack Dongarra, Al Geist, and Vaidy Sunderam. Visualization and debugging in a heterogeneous environment. *Computer*, 26(6):88–95, June 1993.

[2] Adam Beguelin and Erik Seligman. Causality-preserving timestamps in distributed programs. Technical Report CMU-CS-93-167, Carnegie Mellon University, Pittsburgh, PA, June 1993.

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

[4] G.A. Geist et al. PICL:A Portable Instrumented Communication Library, C reference manual. Technical Report ORNL/TM-11130, Oak Ridge National Lab., Oak Ridge, Tenn., 1990.

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

[6] 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.

[7] 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.

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

[9] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, and Bodin F. Performance analysis of pC++: A portable data-parallel programming system for scalable parallel computers. In *Proceedings of the 8th International Parallel Processing Symposium (IPPS '94)*, pages 75–84, Cancun, Mexico, April 1994.

[10] Daniel A. Reed et al. An overview of the Pablo performance analysis environment. Technical report, University of Illinois, Urbana, Illinois 61801, November 1992.

[11] John T. Stasko. The PARADE environment for visualizing parallel program executions: A progress report. Technical Report GIT-GVU-95/03, GVU Center, Georgia Tech, Atlanta, GA, January 1995.

[12] 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.

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

[14] V.S. Sunderam and Vernon J. Rego. EcliPSe: A system for high performance concurrent simulation. *Software: Practice & Experience*, 21(11):1289–1219, November 1991.

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