Visualizing the Execution of Threads-based Parallel Programs

Qiang A. Zhao
John T. Stasko

Graphics, Visualization and Usability Center
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
Email: {*azhao,stasko*}*@cc.gatech.edu*

Technical Report GIT-GVU-95-01

January 1995

Abstract

One popular model of concurrent computing is threads-based parallel programming on a shared memory parallel computer. A variety of different vendors and machines provide such capabilities, and support for threads programming has begun to appear in desktop multiprocessor systems such as the Sun SPARCstation 20. Unfortunately, building parallel programs that use threads is still quite challenging, even for veteran serial programmers. This stems from the difficulty of controlling communication and synchronization between the different processes. We believe that the use of program visualization tools that graphically depict the state of a program's execution can help programmers develop, debug, and understand their code faster and more easily. Most program visualization systems for parallel computation have focused on performance views and views of message passing systems. Here, we introduce a presentation methodology and a set of views particularly appropriate for depicting the execution of threads-based parallel programs. These views were created specifically for the pthreads programming library on a KSR machine, but they could easily be adapted to other threads-based systems. We also describe our techniques for gathering program execution data to drive the views, and we discuss what an ideal program tracing environment should provide to support the views we developed.

1 Introduction

In shared memory threads-based parallel computation, a program begins with one thread of control that subsequently creates (or "forks") new threads. Each active thread can access common, shared memory areas in addition to having its own local memory. Communication and cooperation between threads occur via barrier synchronizations, the use of mutually exclusive code segments, and other synchronization primitives.

Making the threads communicate and synchronize properly is the primary challenge of this type of programming. Small errors in the logic of a design or in its implementation can result in erroneous programs that fail in any variety of ways, or that simply deadlock.

Debugging threads-based programs is difficult for a number of reasons. First, the multitude of processes simply provide more entities to keep track of. Also, examining an individual process using a traditional serial debugger may not provide the information necessary to determine the cause of the problem. Bugs in threads programs happen when communication between processes does not occur in the planned fashion. Often, it is necessary to track the state of multiple concurrent threads to diagnose a bug. To address this problem, programmers usually annotate their code to produce trace events that have timestamps. By tediously examining many textual trace logs of the timestamped events, some understanding of a global system behavior (hopefully) may be gleaned.

We propose another solution to this problem: The use of program visualization tools that help programmers understand and analyze their code. Effective visualizations can depict how threads communicate and can provide a better depiction of the global state of an execution, all in a manner that textual tracing tools cannot. In particular, visualizations provide two key benefits:

1. They encode a great deal of information in a relatively compact area, highlighting the most important features of a computation while abstracting away unnecessary details.
2. The visualizations can more accurately portray the temporal ordering of program operations to the viewer.

More specifically, the goal of this work is to provide a set of program views that will help programmers better understand what is occurring during a program's execution, and consequently be able to more quickly determine fixes for bugs. To achieve this goal, *effective* program views are necessary. Below are the criteria that we use to characterize effectiveness.

First, the views must focus on the critical aspects of threads-based parallel computing. They must illustrate the program entities and operations most critical to the success or failure of the program. In this domain, therefore, the views must display the state of the individual threads, barriers, and mutexes in the program under examination.

Second, the views must show the current state of execution in detail as a program is run or retraced. A viewer should be able to gather the required information to assess problems in the program. Also, the views must provide some notion of the history of the execution, allowing viewers to "look back" in time and examine the program state at earlier times.

Third, the views should be natural and easily understood by programmers. One way

to achieve this goal is to use clear, familiar graphical notations such as program callgraphs, timelines, and simple symbolic imagery in the views.

Finally, program views are more effective if it is easy for the programmer to use the views. That is, it should be relatively straightforward to elaborate the mechanisms that generate the program views. Extensive modifications of the underlying program should be avoided, and elaborate graphical specifications or manipulations should not be required.

Prior work in visualization of concurrent programs has not placed a major focus on depicting the executions of shared memory threads programs[KS93]. Significant work has been done, however, on performance visualization, where the best-known example is the ParaGraph system[HE91]. ParaGraph provides views of processor utilization and message passing using a set of views such as Kiviat and Gantt charts. Other significant work on visualizing the operations of message-based distributed systems includes Seeplex/Seecube[Cou88], HyperView/Tapestry[MR90], Belvedere[CHK92], and Xab[BDGS93]. Similarly, systems for visualizing compiler-driven (FORTRAN) parallel programs such as SCHEDULE[DS87] and SHMAP[DBKF90] have been developed. Our work focuses specifically on threads programs and programming primitives, and is pioneering in its rigorous depiction of this paradigm.

In the next section we describe the system support and framework for the views we developed. This includes both the parallel computation platform and the visualization package. In Section 3 we describe the threads views in detail and motivate the reasons why we chose these particular views. Section 4 discusses the limitations of our approach and the more general environment in which the threads views can operate. Finally, Section 5 describes the status of the project and our future plans.

2 System Foundations

The particular threads implementation we chose as a prototype was Pthreads[Ken92], a standard thread library on KSR parallel machines. This platform was available to us locally and provided a good example of a threads implementation. Other similar threads implementations include Solaris threads from Sun and cthreads[Muk91]. We used POLKA[SK93], an object-oriented software visualization methodology and library, as the platform to implement the graphical views.

2.1 Parallel Execution Environment — KSR Pthreads and Gthreads

The KSR Pthreads interface is based on draft 4 of the proposed POSIX *Threads Extension for Portable Operating Systems* standard [IEE90]. The term *pthread* refers to a single sequential flow of control within a process, which in turn is a Mach operating system task. Usually a program begins with one *pthread* and creates others to accomplish its task concurrently.

2.1.1 Pthreads Overview

The Pthreads C library supports several types of objects such as pthreads themselves, mutexes, condition variables, and barriers.

Regular data structures can be declared to be shared among all the pthreads by prepending `__shared` qualifiers on the declaration lines. The `__private` qualifier is available to specify thread-specific variables which each pthread receives a private copy of upon thread creation.

One pthread is automatically created by the system when a user process starts. A new independent pthread is forked by calling library routine `pthread_create()` which takes a function pointer as one of its parameters. For example:

```
pthread_t       thread_id;
void           *worker_thread_function(void *);

pthread_create(
        &thread_id,                 /* get the thread id */
        pthread_attr_default,       /* default attrs for new thread */
        worker_thread_function,     /* new thread starts there */
        (void *) NULL               /* arg to worker function */
        );
```

The new pthread starts in the specified function (such as the `worker_thread_function()` in the above example) and executes from there. When that function returns, the corresponding thread terminates with it.

A pthread can terminate at any time by calling `pthread_exit()`. The Pthread library uses either the data pointed to by the parameter passed in to `pthread_exit()` or the return value of the thread start function as the *exit status* of the terminated thread. One thread may call `pthread_join()` to wait until another thread terminates and acquire its exit status.

Mutexes are synchronization objects that serialize access to shared variables or segments of code. A mutex is initialized through the call `pthread_mutex_init()`. A segment of code that needs to be serialized should be surrounded with calls to `pthread_mutex_lock()` and `pthread_mutex_unlock()`:

```
pthread_mutex_lock(&ioLock);
/* ... critical region ... */
pthread_mutex_unlock(&ioLock);
```

Condition variables control conditionally locking and releasing mutexes. If in a critical region a pthread needs to wait until some other work has been done by other threads, it uses `pthread_cond_wait()` to release the mutex (so that some other thread may proceed to access the protected resource) and block on a condition variable:

```
__shared char       something_is_not_ready;
__shared pthread_cond_t ioCond;     /* initialized elsewhere */

pthread_mutex_lock(&ioLock);
```

```
/* ... some processing ... */
if (something_is_not_ready) {
    pthread_cond_wait(&ioCond,
            &ioLock                 /* release the mutex */
            );
    /* ... the system will give me back the mutex ... */
}
pthread_mutex_unlock(&ioLock);
```

When the prerequisite work has finished, a `pthread_cond_signal()` must be called to wake up the waiting thread and lock the mutex again to ensure mutual exclusion to the critical region:

```
pthread_mutex_lock(&ioLock);
/* ... some processing ... */
something_is_not_ready = 0;         /* set to FALSE */
pthread_mutex_unlock(&ioLock);
pthread_cond_signal(&ioCond);       /* signal one waiting thread */
```

As an extension to the POSIX standard, barriers help independently executing pthreads to synchronize at certain rendezvous points in the program.

The size of a barrier, n, is set at initialization time to be the number of pthreads that will participate in its use. Each pthread is assigned a unique logical sequence number ranging from 0 to $n - 1$ when accessing the barrier. The pthread which uses the sequence number 0 is called the *master* and all other threads are called *slaves*.

Two different barrier synchronization actions exist: *checking in* and *checking out*. When checking in, the master continues and the slaves block in the barrier library routine until the master initiates the same action. When checking out, the slaves continue and the master blocks until all the slaves have checked out.

Barriers can be used in many different ways. An example application would be a parallel program to find solutions to partial differential equations — all tasks have to complete one iteration before they begin the next iteration. There is a coordinator thread which manages the execution of a set of slave threads:

```
__shared pthread_barrier_t barr;    /* global variable */
__shared char   moreWork = 1;       /* global variable (boolean) */

int             n;                  /* # of workers; set elsewhere */
int             i;                  /* loop control variable */
pthread_t       thread_id;          /* dummy */
void           *worker_thread_function(void *);

/* initialize the barrier - count me in we have (n + 1) threads */
pthread_barrier_init(&barr, pthread_barrierattr_default, n + 1);

/* create the threads */
for (i = 1; i <= n; i ++) {
    pthread_create(&thread_id, pthread_attr_default,
            worker_thread_function, (void *) i);
}
```

```
    /* ... set up the work assignments ... */
    /* coordinate with the workers */
    while (moreWork) {
        /* I wait for the worker threads to finish their work */
        pthread_barrier_checkout(&barr, 0);

        /* see if there's more work to do and set "moreWork" */

        /* the worker threads wait for me */
        pthread_barrier_checkin(&barr, 0);
    }
```

The worker threads have a similar `while` loop:

```
void *
worker_thread_function(void *arg)
{
    /* my sequence number has been assigned */
    int             my_sequence_number = (int) arg;

    while (moreWork) {
        /* ... compute ... */

        /* the coordinator waits for all of us to finish */
        pthread_barrier_checkin(&barr, my_sequence_number);

        /* wait for the coordinator to make high-level decisions */
        pthread_barrier_checkin(&barr, my_sequence_number);
    }

    return some_kind_of_result;
}
```

2.1.2 Gthreads

To acquire the tracing data needed to drive any visualization, we have added another layer to the Pthreads interface, *Gthreads*. It uses macros to "front" the standard pthreads calls. Thus, programmers do not have to add any code to their program beyond one explicit initialization routine that sets up the thread monitoring environment. Our macros simply replace the old pthreads calls (the same name is still used) and add the tracing capability. Each macro first logs an annotation in a trace file, then invokes the normal pthread operation. For example, a Pthreads program may use `pthread_create()` to create new threads. With Gthreads, our macro first logs this operation then calls `pthread_create()` in the usual fashion.

Unfortunately, this macro approach is not sufficient to generate all the information required for our views. In particular, we wanted to include a view of the function calls of a program; Without proper representation of function invocation at the program level, a parallel program can be confusing and difficult to understand. Synchronization events can appear to run together and cooperation among threads can be very difficult to observe without functional context for thread execution. Function calls cannot be traced using our

macro approach, however. Therefore, we use a compromise: we supply two supplemental macros, `gthread_enter()` and `gthread_back()`, that programmers add to their code to allow the program to signal function entry and exit, tracing information not available from the basic Pthreads calls.

When the multi-threaded program (compiled with the Gthreads library) runs, it writes the tracing data to a file that is subsequently used by the visualizer to drive the animations. Programmers can turn off Gthread tracing completely via a compile-time flag when their parallel program needs to be tested under the native system environment.

Obviously, this macro approach is not an ideal solution. A better approach would be to instrument the compiler and/or the run-time environment in order to reduce programmer involvement in the program tracing process. Such an approach has been used successfully by other systems that had access to these capabilities[TSS94]. Unfortunately, such support was not available to us. Our methods do illustrate, however, what can be done in a minimal support environment, thus providing a truer lower bound on a visualization environment for threads programs.

2.2 Visualization System — POLKA

We developed all the threads visualizations using the Polka animation toolkit[SK93, Sta94]. Polka is an animation design methodology and accompanying implementation that is particularly well-suited to building software visualizations. Polka is especially appropriate for developing animations of parallel programs. This is because the Polka model makes it easy to define animations with concurrent actions occurring in the display, a necessity to properly visualize parallel programs.

Polka supports a variety of 2-D, color, graphical objects that can be combined, manipulated, and modified by a variety of action primitives. For example, we can create a blue rectangle and then schedule it to move along a path, to resize, and to change color, all simultaneously. Similarly, we can make multiple objects change at once.

The toolkit was designed to be easy to learn and use, yet it still provides a very sophisticated set of operations that can be used to build complex animations in relatively little code. It is built in C++ and runs on top of the X Window System and Motif.

Polka can be used by anyone to build an individual animation for a particular program or algorithm. In this work we pre-built the animation views and routines, which then are provided to other users who need not do any Polka coding.

3 Gthreads Graphical Views

A programmer begins a debugging session by using the Gthreads macros and routines to gather a program trace. The programmer then invokes our visualization program, *threadview*, using the program trace file as input. At start-up threadview displays the generic POLKA control panel (top-left thin window in Figure 1) which controls the speed and provides a single step operation. Next, the threads, function, and history views are displayed. They are automatically created each time the visualizer is run. Mutex and barrier views are

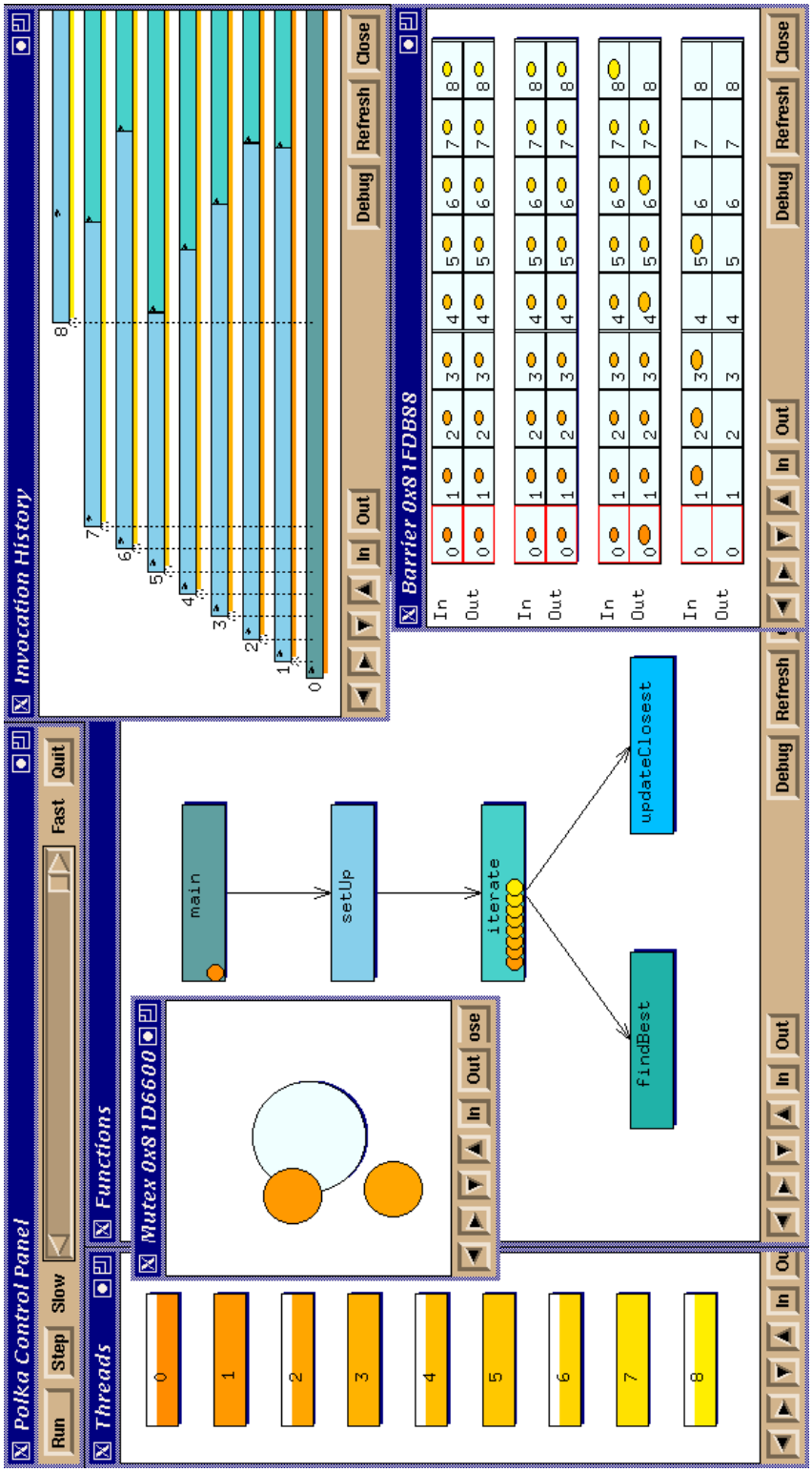

Figure 1: An example threads viewing session utilizing all the different program views.

created when needed by the visualizer. Figure 1 shows a viewing session with one example of each different view. Below, we discuss the particular views and how they work in more detail.

3.1 The Thread View

For a programmer it is often desirable to see how many threads exist in the program, and some simple status information such as whether they are active or not at a certain point in time. The thread view (Figure 2) satisfies this need by listing the threads in the program. Each rectangle represents one thread, and the rectangles are assigned unique colors to help differentiate the threads.

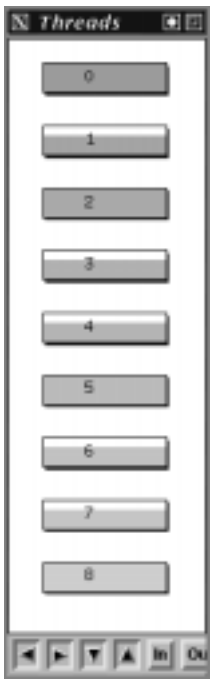

Figure 2: The Thread View

At a particular execution point, if a thread is ready to carry out instructions or is running, then its corresponding rectangle is fully colored. If the thread is idle and blocked, possibly waiting on a mutex or in a barrier synchronization phase, its rectangle is half colored. When a rectangle turns to black, it means that the thread has terminated but its address space has not been reclaimed. The disappearance of the rectangle signifies that the thread has been detached and its address space has been reclaimed.

This view is used as a programming and debugging aid to remind the programmer about shared resources, some of which may need to be better utilized and some of which may need to be better protected from concurrent access. In the case of a deadlock situation or other synchronization errors, the programmer can find and focus on the set of questionable or strangely behaved threads. This view also acts as a legend for other views in the visualizer as we shall see later.

3.2 The Function View

Beyond simple status information about the threads, it is helpful if a programmer can determine where the threads are executing at a particular time in the program execution. Providing a textual line number for the current execution point of a thread might be useful, but it requires too much effort of the viewer to coalesce these values and understand their significance. An alternative, which is more abstract than a reference to program source code, is to show a call graph of the program that relates program trace events to the logical program structure.

The function view (Figure 3) shows the static call structure of a program. Each rectangle represents a function in the parallel program. Again, unique colors (different than those of the threads) are used to differentiate the functions. In this view, threads are represented by small circles or disks with color matching that in the thread view. If a disk (thread) moves along an arrow in the call graph, it is calling the corresponding function. Movement in the opposite direction signifies a return from the invoked function. When a thread of control is executing a particular function, the corresponding disk simply takes a position inside the rectangle of the appropriate function.

Figure 3: The Function View

Each newly created thread is initially drawn as a tiny point which gradually grows to its regular size. The new thread also moves from where its parent thread is executing to the function in which it will begin execution.

Another interesting feature of the function view is that the program call graph is constructed dynamically as an execution is traced. The visualizer initially may only know the start up function, such as `main()`, and this is the only rectangle shown. As new threads

are created and new functions are called, the visualizer learns more about the program, creates new function rectangles, and adjusts the call graph layout to illustrate this knowledge. We attempt to produce aesthetically pleasing graph layouts that indicate typical call sequences by making levels in the graph, and by minimizing the edge crossings of the function invocations.

A side-effect of this dynamic behavior is that this view does not show the complete call structure of the program. Rather, it displays the specific call structure for the particular program execution on which the trace events were generated. The program could have unvisited functions and different possible calling patterns, but we chose to abstract away that information to focus on the specifics of this particular debugging session.

3.3 The History View

Although the function view depicts the overall state of an execution at any time, it does not capture the history of functions encountered by the threads during execution. The history view (Figure 4) was developed to show the execution history of the threads. In this view each thread occupies one row. Time is encoded along the X-dimension and flows to the right, so threads grow to the right until they terminate. The drop shadow color of each bar (thread) identifies the thread by matching its original color in the thread view. The lifetime of a thread (main part of the thread bar) is divided into segments, each corresponding to a time period during which the thread executes in a particular function. These segments of the bar are colored to match the colors used in the function view. A tiny arrow head in a segment pointing to the right stands for a function invocation. Similarly, a tiny arrow head pointing to the left indicates a function return.

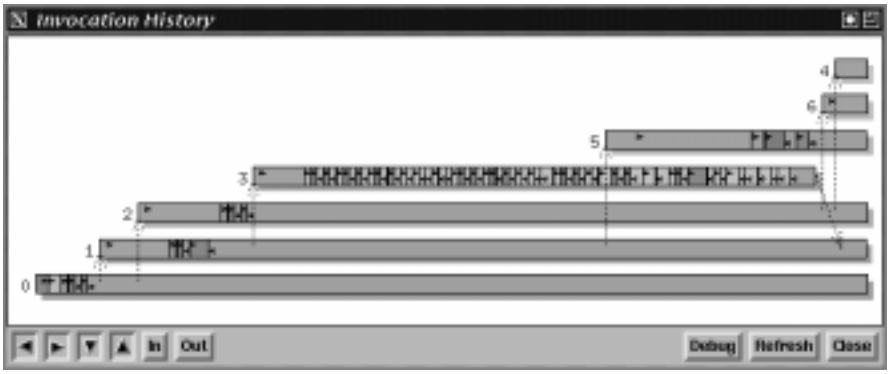

Figure 4: The History View

Thread creation is denoted by drawing a dotted vertical arrow pointing from the parent thread to the beginning of the new thread. Thread joining is indicated by a dotted arrow pointing out from the joining thread to the joined thread.

As time proceeds, the history view automatically scrolls to the right so that the current point in time is always shown. Viewers can, however, use the Polka scroll buttons in the lower left portion of the view to move back in time (scroll to the left) to examine previous points of the computation.

The history view and the function view together can help verify a program execution

against the programmer's mental model of the computation. For example, a chain of unwanted recursive calls can be easily noticed in the function view. Similarly, a function call outside of a normal pattern of calls can quickly be spotted in the history view, simply by matching the patterns on the thread bars.

The three views discussed so far provide information about the thread objects themselves in a multi-threaded program. For other kinds of objects such as mutexes and barriers, we have developed separate views.

3.4 The Mutex View

Mutexes are primitive objects for protecting shared resources. Common errors in using mutexes include circular waits which could lead to deadlock and failure to achieve mutual exclusion. Knowledge of which of the threads are trying to lock the mutexes and which of the threads own the mutexes are necessary information for detecting such errors.

As opposed to the previous views, a mutex view is not created automatically at start up time. Rather, it will be created only if the target parallel program uses a mutex, and it will be created on a case-by-case basis as each new mutex is utilized.

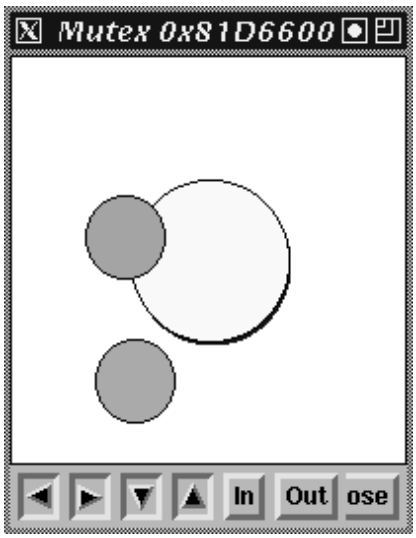

Figure 5: The Mutex View

A mutex is displayed as a big circle. When a thread is trying to lock the mutex, it appears as a smaller disk (color coded to the thread view) somewhere around the circle. When a thread gets the lock, it moves into the circle and remains there until it unlocks the mutex. Then, it leaves the circle and disappears. Figure 5 shows a mutex with a thread caught in the middle of a move, and another thread waiting outside to lock the mutex.

Deadlocks are obvious in the mutex views. For example, in a simple case there might be two mutex views: mutex view 1 shows that thread 1 is holding the mutex and thread 2 is waiting outside; mutex view 2 shows that thread 2 is holding the mutex and thread 1 is waiting outside — these two threads are deadlocked.

3.5 The Barrier View

In threads programs, barriers are relatively difficult to understand and program. The barrier view helps programmers to better understand barrier operations, and it illustrates how the threads cooperate in the synchronization.

Like the mutex views, barrier views are created dynamically on a per barrier basis. A barrier (Figure 6) is presented as an array of boxes. Threads are represented by small colored disks that will fill in the corresponding boxes to designate the check-in and check-out phases. When a thread finishes a barrier synchronization phase, it is not removed from the designated box, however. Rather, it is resized smaller to indicate that the corresponding synchronization has been performed. Removing the threads disks would create a confusing display where it would be difficult to assess the current status of the barrier. When a new synchronization phase of the barrier begins, a new array of similar boxes is created below the old one, and the threads start to use the new one.

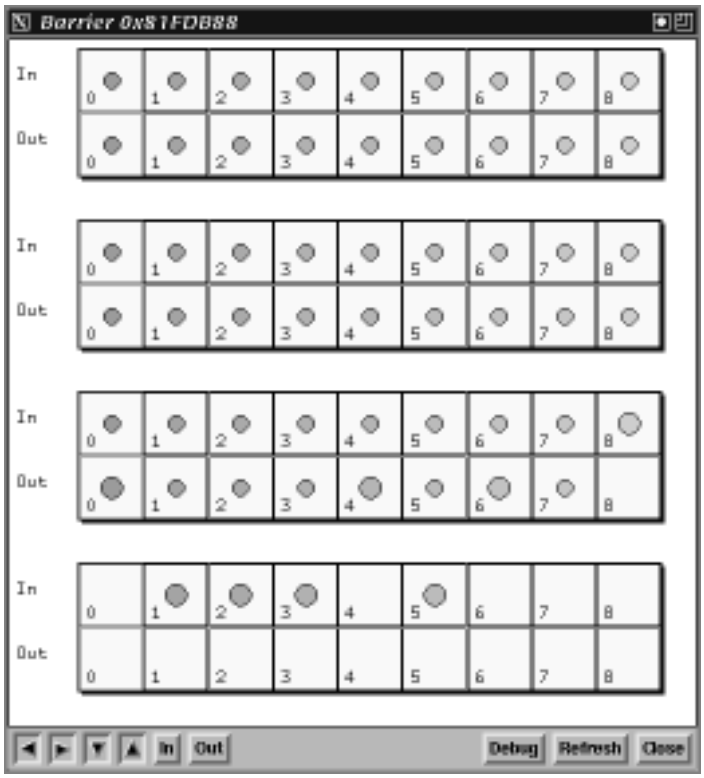

Figure 6: The Barrier View

Like the thread history view, the barrier view scrolls when necessary to keep the current barrier configuration displayed inside the window.

In a particular program that computed minimum spanning trees on graphs, we modified a parameter and the program subsequently failed to produce the correct result. Using the barrier view we found that the array of rooms for the barrier had a *gap*, and the colors for the threads to the left of the gap seemed to be shifted. This looked like a sequence number problem, and a little further investigation did reveal that some of the threads were not using the correct sequence numbers.

4 Limitations

The threads views we developed have proven useful for helping to understand and trace program executions. Nonetheless, our use of the views has exposed certain limitations. In this section we discuss these limitations, and we describe potential improvements to the views.

Two important issues which the Gthread visualizations do not adequately address are scale and interaction. Our views work well and are easily understood when the number of threads is rather modest, say, less than 40 or 50. On many systems the number of threads in a program stays well below this threshhold due to operating system and library limitations. However, many systems, particularly distributed threads systems, have programs with hundreds or thousands of threads. Such programs would require new views to handle this much bigger scale—Our views still would be useful to show the state of a particular smaller set of important threads.

This last issue highlights another problem, viewer interaction with the views. It would be advantageous if the viewer could easily compress/expand the Gthreads depiction of particular threads, functions, barriers, and mutexes to tailor the program presentation as desired. This would allow the viewer to focus on information of interest and remove unwanted details. Also, the views should provide simple user interface interactions to support these focusing operations.

Another important limitation of our approach concerns the simple manner that program event tracing is performed. Since all the threads write their trace events to one particular file, inaccuracies in logical event ordering, so-called tachyons[BS93], can easily occur. For example, a thread can appear to join a second thread before the second thread has properly terminated.

Any number of solutions to this problem are possible: The events can be written to separate files based on process-id (this causes other subtle complications in processing and visualizing the events later); The event file can be post-processed to remove logical inaccuracies[BS93, NMGM95]; A logical time tracing approach can be used[TC93, TSS94]. Perhaps the easiest of these methods is to post-process one trace file. Even if the logical time errors are corrected, this approach still has another problem. When the one file is read and visualized by *threadview*, an essentially serial animation results. That is, each event is animated in order as the file dictates. One event is processed, it is animated, and then the next event is processed—The concurrency is effectively removed.

Fortunately, we remedy all these tracing and timing problems by using the Gthreads views in a broader context, a comprehensive framework called PARADE (PARallel Animation Development Environment) we are developing[Sta95]. PARADE provides the context for visualizing many different types of concurrent systems and its three primary components involve

- Gathering tracing information about a program's execution
- Mapping the execution information to appropriate actions in the presentation component

- Generating the visualizations and animations of the program execution

Key to solving the timing problems discussed above is the middle component, the mapping from program events to animations. In PARADE a tool called the Animation Choreographer fills this role[KS94]. It can take an event log such as a single Gthreads trace file produced by our macros or a set of per-thread trace files, and "re-introduce" the potential concurrency of the logged program events. The Animation Choreographer analyzes the trace data and produces a program event graph depicting logical constraints between the different events. Viewers then choose one of a number of different logical/temporal perspectives on the event graph. Finally, the corresponding program animation with all the Gthreads views is generated to match the selected perspective.

Using the Animation Choreographer, viewers can watch an animation of their program in which events are shown happening with respect to a global timestamp on each event (here, tachyons might exist and they would be flagged as necessary). Alternately, a serial animation of the program events could be shown, much as it might appear with the simple *threadview* program. Finally, the Choreographer could be used to visualize all the potential concurrency in the program execution. The Gthreads views would then have many simultaneous actions being displayed.

5 Conclusion

We have created a set of graphical views designed to help programmers better understand the structure and execution of multi-threaded parallel programs. The views depict thread state, thread history, function callgraph, barrier and mutex primitives within a program. The views use color, layout, and familiar graphical notations to help detail the state of the program execution to the viewer. Our approach minimizes the need for program annotation by having the thread library automatically generate trace events whenever possible. This is performed by providing a wrapper for the library primitives using macros. The parameters to the primitives provide us with the information necessary to generate the views. Finally, the graphical views we have developed use smooth animation to help viewers track a program execution. They provide both a gestalt view of the entire computation and details of the individual operations critical to the success of the computation.

All the code and systems described in this paper are implemented and functional. The code for the program monitoring and the graphical views is available via anonymous ftp from `ftp.cc.gatech.edu` as the files `pub/people/stasko/gthread.KSRtracing.tar.Z` and `gthread.Animations.tar.Z`. Further information including color images of the views is also available on the WWW through URL

`http://www.cc.gatech.edu/gvu/softviz/parviz/gthread/gthread.html`.

Our future plans involve view support for other threads libraries and for on-line threads visualization support. Improvements to the views and the view library might include new views (e.g., access/synchronization dependency graph), optimization (e.g., minimizing visual changes in the function view to preserve context), better interaction (e.g., dynamically

choosing from a set of levels of details, collapsing and grouping some items together), and the issue of scale (e.g., visualizing programs with hundreds or thousands of threads).

6 Acknowledgments

Support for this project has come from the National Science Foundation under grant CCR-9121607 and Kendall Square Research. Eileen Kraemer has provided help with the graphical views and the content of this report.

References

[BDGS93] Adam Beguelin, Jack Dongarra, Al Geist, and Vaidy Sunderam. Visualization and debugging in a heterogeneous environment. *Computer*, 26(6):88–95, June 1993.

[BS93] Adam Beguelin and Erik Seligman. Causality-preserving timestamps in distributed programs. Technical Report CMU-CS-93-167, Carnegie Mellon University, Pittsburgh, PA, June 1993.

[CHK92] Janice E. Cuny, Alfred A. Hough, and Joydip Kundu. Logical time in visualizations produced by parallel programs. In *Visualization '92*, Boston, MA, October 1992.

[Cou88] Alva Couch. Graphical representations of program performance on Hypercube message-passing multiprocessors. Technical Report 88-4, Tufts University, Boston, MA, April 1988.

[DBKF90] Jack Dongarra, Orlie Brewer, James Arthur Kohl, and Samuel Fineberg. A tool to aid in the design, implementation, and understanding of matrix algorithms for parallel processors. *Journal of Parallel and Distributed Computing*, 9(2):185–202, June 1990.

[DS87] Jack Dongarra and Danny Sorenson. SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs. In L.H. Jamieson, D.B. Gannon, and R.J. Douglass, editors, *The Characteristics of Parallel Algorithms*. The MIT Press, Cambridge, MA, 1987.

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

[IEE90] IEEE. *Threads Extension for Portable Operating Systems (P1003.4a)*, 1990.

[Ken92] Kendall Square Research Corporation, Waltham, MA. *KSR C Programming*, February 1992.

[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.

[KS94] 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.

[MR90] Allen D. Malony and Daniel A. Reed. Visualizing parallel computer system performance. In Margaret Simmons, Rebecca Koskela, and Ingrid Bucher, editors, *Parallel Computer Systems: Performance Instrumentation and Visualization*. ACM, New York, 1990.

[Muk91] Bodhisattwa Mukherjee. A portable and reconfigurable threads package. In *Proceedings of Sun User Group Technical Conference*, pages 101–112, June 1991.

[NMGM95] Gary J. Nutt, James E. Mankovich, Adam J. Griff, and Jeffrey D. McWhirter. Extensible parallel program performance visualization. In *Proceedings of the MASCOTS '95 International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages ??–??, Durham, NC, January 1995.

[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.

[Sta94] John T. Stasko. POLKA Animation Designer's Package. Unpublished System Documentation. Available via anonymous ftp from ftp.cc.gatech.edu as pub/people/stasko/polka.tar.Z, 1994.

[Sta95] John T. Stasko. The PARADE environment for visualizing parallel program executions: A progress report. Technical Report GIT-GVU-95/03, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, January 1995.

[TC93] Stephen J. Turner and W. Cai. The 'logical clocks' approach to the visualization of parallel programs. In G. Kotsis and G. Haring, editors, *Performance Measurement and Visualization of Parallel Programs*, pages 45–66. Elsevier, 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.