Applying Program Visualization Techniques to Aid Parallel and Distributed Program Development (Work In Progress) Technical Report GIT-GVU-91-08

William F. Appelbe
John T. Stasko
Eileen Kraemer

Technical Report GIT-CC 91/34

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

bill | stasko | eileen@cc.gatech.edu

October 16, 1993

Abstract

Parallel and distributed programming is intrinsically more difficult than sequential programming, yet few effective tools or methodologies have been developed to help programmers understand the behavior of their parallel programs. Browsing source code and tracing program output are tedious and often ineffective approaches for parallel program understanding.

Program visualization, which relates a program's behavior to the programmer's model of the system's components and interactions, has been shown to be a novel and highly effective approach to program and algorithm comprehension. Extending and adapting program visualization to parallel programming can aid comprehension of the complex concurrent events and transitions that occur in parallel programs.

We are defining a model for the capture and display of parallel program events and transitions, based upon the path-transition paradigm for animation, and partial ordering of events. Using this model, we are developing a prototype for visualizing parallel programs, and testing the model and prototype upon a suite of scientific parallel programs.

1 Introduction

Parallel and distributed[1] program design and development continues to be an area in need of new and improved methodologies and software tools. The myriad of system architectures and programming paradigms has complicated parallel program development and restricted the growth and utility of multiprocessor programming. One cause of this problem, as in serial programming environments, is the lack of tools to aid debugging. Parallel program development, however, is further complicated by the need for performance tuning – after a parallel program has been sufficiently debugged and works correctly, tuning is usually required to refine the program's processor and memory utilization (i.e., obtain better "speedup").

During software system development, programmers typically create, either explicitly or implicitly, their own semantic model or specification of how the system should function. This model may correspond very closely with the system as in the case of physical simulations, or it may be a convenient abstraction as is often the case for parallel systems software, such as sorting programs. Parallel program development would be significantly improved if programmers could quickly and easily generate visualizations of these semantic models, and then map their programming systems onto the visualizations. Thus, as program execution is traced, system performance could be monitored and analyzed by viewing the program visualization.

The goal of this study is to create methodologies and tools to aid parallel program development through the use of program visualizations. Our primary emphasis is upon making the creation and operation of appealing visualizations as simple as possible. "Average" parallel programmers must be able to rapidly create their own program visualizations. This goal requires us to solve subproblems such as providing simple means to specify informative graphical views, gathering application-specific semantic data to generate these views by program monitoring or tracing, and coordinating and synchronizing the visual elements during a program animation from the gathered data. Continuous, synchronized motion (i.e., animation), is critical for visualization of parallel programs. Animation is important because of the continuous nature of many physical simulations to be analyzed, and because a visualization with choppy, discrete updates is significantly more difficult to understand than one presenting a continuous stream of changes.

The methods we are utilizing to meet these goals involve refinements of parallel code analysis tools and serial algorithm animation techniques that we have previously developed. This foundation upon which to build is important. However, the unique problems in developing parallel program visualizations are quite challenging. They provide a number of interesting conceptual and systems-oriented challenges, which we describe in the following sections of this report.

In section 2 we discuss, in more detail, background and motivation for the research, highlighting the utility of our approach. In section 3 we discuss the specific goals to be met by the developed tools. In section 4 we describe our technical approach, examining the program data dispersal, animation coordination, and graphical view generation processes individually and the research problems they address. We also discuss subsequent implementation plans.

[1] From this point on, we will just use "parallel" for brevity, except where the technical differences are critical.

2 Parallel and Distributed Program Development

The primary application of parallel computers is the simulation of complex physical systems, such as fluid flow, seismic waves, atomic and molecular physics. Developers of these types of parallel programs usually have a sequential algorithm or program as a starting point, and they parallelize their program by recoding it using parallel constructs or compiler directives. Unfortunately, the process of parallel program development is extremely difficult in part because parallel program constructs have been grafted onto existing sequential languages such as Fortran in an extremely ad hoc, machine dependent manner[KBI88]. Erroneous use of these constructs often results in inefficient, or worse incorrect, programs. Also, few tools for parallel program development have been developed, and those that have, such as debuggers and performance monitors, are often poorly integrated and difficult to use.

Ideally, developers of parallel programs would benefit from tools which could isolate bugs in their parallel programs and help them understand the output of their programs. Such tools fall into four classes:

1. Static analyzers and parallelizers, which analyze source code to determine which sections of code can be parallelized, transform source code into parallel forms, and determine if user-inserted parallel constructs and directives are erroneous[CKT86, AB+88, ASM89].

2. Performance monitors, which graphically display statistics such as processor utilization, interprocess communication and process scheduling, usually using runtime trace or log files[HHK85, LR85, HC87, NE88, MR88, FLMC89, Hea89, ZPS89, Sto89, RRZ89, L+89, GJG+89, MRA+89, DBKF90, M+90, KS91].

3. Algorithm animations, which present visual abstractions of a program's semantics and which display program *events*, program state changes that correspond to significant physical state changes, in a form that illustrates physical behavior (e.g., a 2-D state space, in which points signify particles, and curves model wavefronts)[SBN89].

4. Scientific visualizations, which display the program states in a format specifically designed to closely match the physical system being modeled (e.g., cloud and tornado formation over a plain, with the cloud displayed using shading and volumetric rendering)[DCH88, DBM89].

Due to successes in serial program development[AB89] and as evident by ongoing research in classes 2, 3, and 4 above, increasing attention has been directed toward visualization techniques in order to help developers understand the runtime behavior of their parallel programs. The motivation underlying this approach is that graphical displays can supply, concurrently, a great quantity of important contextual and state information about the computation state, processor utilization, and program data, operations, and semantics.

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

Of the classes of tools above, 2, 3, and 4 are also all intended to help developers understand the runtime behavior of their programs. Tools in class 2, performance monitors, are most useful in isolating performance problems, e.g., why a parallel program is not exhibiting a significant speedup. However, these tools often cannot uncover subtle bugs in which a parallel program's output does not reflect the physics of the system being modeled. Examples of such difficult bugs have occurred often. For example, in a monte-carlo parallel simulation of gamma particles[ASM89] the original sequential algorithm had a bug whereby particles travelling very close to perpendicular caused invalid interparticle interactions. The bug took months to discover by traditional techniques, but an animation system with simple facilities for displaying particles would have rapidly isolated the bug.

Current performance monitors can only display runtime information generated by runtime system events (e.g., task creation, waits on events), although more sophisticated performance monitors can display runtime information for program specific events (e.g, variable access/update). However, such displays simply consist of graphics library objects (widgets) such as bar-charts, kiveat charts, timelines, and pie-charts, with independent animation of each widget. There is no mechanism for defining new graphics representations, (composed of circles, boxes, arcs, etc.) and their animation, to suit a particular application.

In algorithm animations, class 3 above, the program developer is presented with a palette of graphics objects, such as lines, circles, rectangles, and text for associating with program objects and operations, together with graphic motions, such as move, glide, and blink for associating with program events, such as a particle bouncing, or colliding with another particle. The purpose of the majority of numerical scientific programming is modeling and simulating physical systems. The scientist usually has some expectations concerning the physical behavior of such systems. In particular, the majority of physical systems are continuous. Algorithm animation can greatly help in comprehending parallel program behavior by providing a means to display program behavior so that it can be readily related to the continuous physical system being modeled or simulated.

Algorithm animation also offers the possibility of exploring alternative program behavior, in which the program developer can control the rate of execution of different tasks. Such animation techniques can be used to analyze a simulation to determine if, for example, two particles can collide.

Although scientific visualization offers the best, and most spectacular, understanding of a physical system, it suffers from the drawback that the effort devoted to developing the animation is high, and highly application specific. By contrast, algorithm animation is relatively simple to instrument, yet offers critical insight into parallel algorithm behavior.

2.1 Related Work

Recently, several systems for debugging parallel or distributed programs via some form of graphical aid have been developed (survey in [PU89]). For the most part, these systems have focused on displaying views of the architecture, processor structure, parallel computation model, task structure, and processor state, in essence, the "machinery" of the parallel computation, making them performance monitors as in class 2 above.

Malony, Reed, and others[MR88, MRA+89] have created a number of flexible X-Windows based performance monitoring widgets. Histograms, dials, gauges, and Kiviat diagrams reflect message traffic and processor actions. Similarly, Heath[Hea89] provides nine different performance views, including Feynman diagrams and Gantt charts among others, of programs running on message-passing multiprocessor architectures. The SHMAP[DBKF90]

tool provides views of memory access and utilization of parallel Fortan matrix algorithms. The system's designers note that a better understanding of processor memory usage can help to improve program performance and promote experimentation with alternate usage strategies. ChaosMON[KS91] uses a program execution summary database and view specification "mini-languages" to allow viewers to rapidly specify the aspects of a program's performance of primary interest.

These systems, some of the more recent examples of a large history of projects, provide views on how the parallel system architecture is supporting the program being run. In our work, we are more interested in displaying views of the application program itself—its basic data and operations, as well as visual abstractions of its fundamental semantics. Our approach does not preclude also providing a library of performance monitor views, however. Our general framework, rather than the view library approach, will promote this type of flexibility.

McDowell and Helmbold have characterized these two different parallel visualization domains as "time-process" displays and animations of programs and data, respectively[MH89]. They argue that both are needed in an ideal debugger for a concurrent system. Our focus is on the latter; it has received relatively little study to date.

An existing system near to our work is Voyeur, which supports application-specific parallel program animation views primarily as an aid for debugging[SBN89]. Voyeur strives to simplify the creation of such types of views by programmers. The system separates the data gathering process from the animation display process. It provides a variety of program views ranging from text views of specific variables to more abstract graphical presentations of the program's state. Our system differs in that we seek to provide further support for the program-to-animation mapping, and to allow simplified direct manipulation creation of multiple program views with more continuous, smooth imagery changes.

LeBlanc, Mellor-Crummey, and Fowler[LMCF90] have created an impressive system for gathering and visualizing execution information from parallel programs. Their approach provides a multiplicity of different program views, each chosen to best illustrate a particular aspect of the execution for debugging aid, together with a programmable interface based upon Common Lisp. In their system, program execution is captured in a synchronization trace, and later converted into a graph representation. A user manipulates the graph representation to create and modify the different program views. These views fall into three main categories: process interactions, process states, and time.

Sarukkai[Sar90] also is developing a system with an emphasis on many flexible views, ranging from low-level process monitors to high-level data structure animations. His system creates a relational database of program execution information, and allows users to pose post-mortem queries against the database. His most high-level application data views are similar to those found in the sequential visualization system BALSA[Bro88b].

Our approach differs from the two previous systems in an increased emphasis on the resultant visualizations and animations. We want to support highly-semantic application views that exhibit smooth, gradual changes over time. We also want end-users to be able to develop *new* views on their own, in a fairly short period of time, without learning a complex visualization/animation programming language or environment, or programming the animation in a general-purpose language such as Lisp with embedded animation primitives. Our work also seeks to identify the elements necessary to produce these visualizations and which are common across a variety of parallel and distributed programming models.

Quite a different approach to visualizing parallel programs is taken by Roman and Cox[RC89] who utilize a declarative, rather than imperative, approach to the problem. To

create a visualization of a program, they define two parts: an abstraction function that maps program states to to objects, and a rendering function that maps objects to images. Exhibiting similarities to proofs of program correctness, their approach seeks to build visualizations without any source code modification. That is, their visualization component must examine the program data during execution and build the appropriate mapped visualization. This method can be successful when visualizing shared dataspace parallel programs, but it is significantly more difficult for other parallel programming models. An imperative animation specification is more desirable for animation–showing path motions and controlling relative transition speeds.

The vast majority of research on graphical display of programs has been for serial programming. These *program visualization* systems[Mye90] can be divided into two broad categories: data structure display and algorithm animation systems.

Data structure display systems[Mye83, Bas85, ISO87, Moh88, RMD89] provide automatic, canonical views of a program's data structures. They utilize sophisticated screen layout algorithms to show program data while a user traces a program in a debugger. These tools provide little in the way of application domain specific information, so they are not sufficient for our purposes.

Algorithm animation systems[Bro88b, Bro88a, LD85, Dui86, Dui87, HHR89], conversely, present dynamic graphical views of programmer-defined abstractions of the important data, operations, goals, and actions in programs. We shall adapt techniques developed in these serial algorithm animation systems in order to aid parallel program visualization and debugging. In particular, we shall utilize a improved modification of the *path-transition paradigm*[Sta90a, Sta90b]. The paradigm supports a building-block style of color imagery (lines, circles, rectangles, text, etc.) along with smooth, visually pleasing animations such as changes in position, size, color, visibility, fill style, and so on. This method is appropriate for our goals because it stresses ease of design and use by programmers, as well as clearly defined, rigorous semantics. A variety of projects such as the Tango algorithm animation system[Sta90c], a system for context-sensitive animated help in user interfaces[Suk90], and a visualization system for examining the reasoning process of an expert system[SJ90] have already been developed using ideas from the path-transition paradigm. Our experience has been that programmers can learn Tango and create new animations within a day.

3 Objectives

The goal of this research is to create a methodology and system that will simplify parallel program understanding and debugging through the use of animated program visualizations. The following objectives support this goal and clarify our plans:

- **Generality of animation display capabilities.** The visualization system's graphics capabilities should be general-purpose and not restricted to a particular set of images and interaction techniques. The system should be unbiased toward any application domain, providing equal utility for animating, say, sorting programs, physical simulations, scheduling algorithms, and so on. This will be achieved by providing a simple but general set of graphics animation primitives, together with tools for developing complex animations using these primitives. Our previous work on the Tango algorithm animation system provides a sound basis, with extensions (1) based upon feedback of users over the past year and (2) to support the particular requirements of parallel programming.

- **Ease of instrumentation.** Creating an animation requires a user to

  1. Design an animation model, and specify it in an animation language or graphical design editor, focusing on a graphical representation for program states, together with the representation for transitions between these states.
  2. Identify the program events which are significant, i.e., which will result in a change of the display state. This is the mapping from the program runtime domain to the animation domain.
  3. Modify the source program to include runtime calls to gather data for the animation at program event locations.

  Ease of animation can be obtained by a simple yet powerful animation paradigm, together with tools to automate the process of modifying the source program. It would be unwise to speculate that a visualization system meeting all our requirements would be extremely simple to use. However, it should be possible to allow fairly sophisticated application programmers (graphics non-experts), not just the visualization system creators themselves, to create program animations. We will support this goal by providing, in addition to a simple animation paradigm, a direct manipulation graphical editor to be used for designing program visualizations by demonstration. In essence, the visualization system should be an open system encouraging design of new animations, not an electronic library of views only to be played back.

- **Importance of view aesthetics.** The visualization system should support the production of aesthetically pleasing, continuous, color animated views. While it is not our goal to provide highly realistic, three-dimensional visualizations as evident in high-end scientific visualization workstations, animated views in the proposed system, nevertheless, should achieve a level of aesthetics that commands attention and entices further viewing.

Animation of parallel programs is intrinsically far more complex than animating sequential programs because the runtime order of program events is not necessarily the same as their desirable animation order. The runtime order of program events is perturbed by operating system task-scheduling which should not necessarily affect the animation; an animation should be able to reflect logical as well as physical time. Animation is intended to reflect the algorithm's feasible behavior, not the specific operating system runtime scheduling behavior. Also, gathering runtime data from a parallel program can perturb the operating system run-time scheduling behavior.

Hence, for parallel programs, additional goals are:

- **Minimal intrusion.** Parallel programs are often non-deterministic, so that monitoring of the state of a parallel program either for real-time or off-line animation can perturb program output. The methods used to gather information that will drive an animation should be as non-obtrusive as possible to the parallel program being viewed. Providing the animation tool with necessary information for its visualization should not adversely affect or alter the application program's performance. Although it would be desirable to display program animations in real-time during application execution, this goal may entail the use of a saved application transcript which is replayed later by the visualization system.

- **Alternative behavior exploration.** The visualization tool should support programmer exploration of alternate application program executions. That is, the visualization system should provide straightforward mechanisms for postulating alternative feasible computation states, processor orderings and commitments, and execution sequences. This experimentation will foster improved algorithms and code.

4 Technical Approach

4.1 Events/Program Monitoring

An implementation of program animation has two distinct phases: display data gathering and creation of an animation from the display data. These two phases can be coroutines, in the case of concurrent execution and animation, or distinct, when posthumous animation occurs after a program run has completed. We intend to support both animation modes, although posthumous animation is simpler to implement, and offers the advantage that it does not perturb parallel program behavior as significantly as concurrent animation.

Data gathered consist of an event log, either input directly to the animation program or saved in a file for posthumous animation[BW83, LMC87]. Each event is usually timestamped with the system clock time at which the event occurred, event parameters, and a task-id. In our model two types of program events are required: *task events* and *display events*. A parallel program consists of a collection of possibly dynamically created tasks, which interact (i.e., communicate and synchronize) via system dependent task events such as POST_EVENT, WAIT_EVENT, CREATE_TASK, PARALLEL_DO, etc. A display events signifies an important *semantic* change in the state of a program, and accordingly usually the state of the animation display (e.g., creating a graphical object, starting a path-transition, etc.).

There are two different mechanisms for gathering event data: *snooping*, in which a monitor task running concurrently with the program monitors the state of the program by accessing global data, and *tracing*, in which the program makes explicit calls to generate events. Snooping is less intrusive[AG89], but more complex to implement and unnecessary for posthumous animation. It is exceptionally difficult to acquire display events via snooping also[Bro88c]. Hence, we use event tracing for the implementation of our system, focusing on a flexible self-documenting trace format such as described in [ROA+91].

To understand display events better, let us examine their role in sequential algorithm animations. There, display events are usually "important" events of special semantic significance to the program being animated. For example, a sorting program might contain *input value*, *compare values*, and *exchange values* events. Under this paradigm, the animation designer has the responsibility of specifying where in the source program the appropriate events are generated. In a large, complex parallel system, however, this specification may be quite difficult. Therefore, in addition to supporting this class of events (it certainly makes animation design easier), we plan to create a display event subclass that needs not be as "specific." For instance, in a particle path simulation program, particles could report their positions every 10 position updates as a display event, instead of at every change of velocity, such as a path endpoint or particle collision. Identifying where in a source program events of this less specific class should be generated is much easier than for the semantic class of events. Unfortunately, the less specific information makes animation design more of a challenge. Balancing these tradeoffs is an interesting research problem of this component.

In order to gather an event log it is necessary to modify the object program, either by

- **modifying source code.** To generate explicit calls to event log library routines.
- **modifying library source code.** Modifying the system's subroutine library for parallel constructs.
- **modifying object code.** Generate calls to event log library routines by means of compiler flags, etc., to a modified compiler.

We will adopt a simple prototype, in which the event log is generated by user-inserted calls for display events, and a modified library for generating task events. Later we intend to use our existing tools, PAT[ASM89, SAS90, ASS91a] and IOS[ASS91b], to help automate the insertion of code to generate display events.

PAT is an interactive parallelizer for Fortran, developed by one of the authors at Georgia Tech over the past four years. PAT is hosted by a range of UNIX workstations, and targeted at several multitasking Fortran dialects, including Sequent multiprocessors and the IBM 30390. PAT takes as input a sequential or partially parallelized Fortran F77 source program, performs syntactic and semantic analysis to generate the callgraph, flowgraph, symbol table and dependence graph. PAT locates loops that can be parallelized and under user control performs source code transformations to remove dependences such as replication, alignment, peeling iterations, inserting LOCKs and EVENTs. PAT has been distributed to over 20 sites and has been used to parallelize a wide range of production Fortran programs[SA89].

Recently PAT has been recoded in C++, and the source code analysis and program data structures "unbundled"[ASS91b] so that they can be used by tools other than PAT. These analysis modules, referred to as IOS, can be incorporated into our proposed system. IOS provides operations to browse and find locations in the source code where events need to be inserted (e.g., after the first assignment to a program variable in a specific subroutine) then generate and insert code fragments. The insertion of display events cannot be fully automated[Bro88c], but can be greatly simplified by IOS' capability to display the program structure (e.g., call graphs, flowgraphs, and source), and to insert code using templates. For example, if a programmer wishes to find references to a given variable, these can be browsed. Unlike editors which are only capable of name matching, IOS tracks variable aliases, such as those through procedure calls.

In animating a parallel program it is necessary to relate the display events to the task state to determine which event is associated with which task (e.g., if several tasks are used in a "data parallel style", in which each task executes identically on different data sets). It is also necessary to relate task states to display events to ensure that the order in which display events are shown is consistent with the task state.

Hence, in order to correctly animate task behavior, task synchronization events must be logged, together with display events, and the animation must be correlated with the task state. This correlation should be automated, so that a programmer does not need to be aware of details such as the runtime task-id's, operating system task scheduling strategies, and so on. In particular, the animation of two runs of a parallel deterministic program ought to be identical, unless the programmer explicitly controls the rate of task animation, or uses the runtime clock returned by events, from the event log.

In summary, some of the research issues being addressed via this section include

- Is a particular trace format or task event structure particularly well-suited for helping to create animations? We intend to target both shared memory and distributed memory multiprocessors. As explained below, the semantics of task events are encapsulated by the next component of our model.

- How do different parallel and distributed programming paradigms affect the relation between task and display events?
- What is the minimal type of application-specific display event needed for driving program animations?
- How do we help a programmer identify the points for inserting display events into a large complex program or system?
- Can alternative approaches to interesting events, such as browsing a program execution database, be used for driving animations?

Addressing these issues effectively will require distributing our tool to several sites and user communities and incorporating their feedback into the functionality and interface of the tool. We intend to devote considerable effort to ensuring that the tool prototypes are hosted upon widely available UNIX workstations, are robust, and ideally are fun to use.

4.2 Choreographer

In animating a sequential system it is important that all transitions are "smooth", and that the programmer (viewer) is provided with tools to control the speed of display of transitions. The animation time-frame is the same as the program event order: once a runtime program event occurs the animation display is updated. Since the animation is intended to reflect the program state, there is usually no need to wait for future events before displaying the current state.

However, in an animation of a parallel system an event cannot be displayed until all events which may have logically occurred earlier have occurred and have been displayed. In a parallel simulation program, tasks which have no interaction do not need to synchronize, and hence may generate animation events out of sequence from the order in which they should be displayed.

Basically, a sequential program's runtime behavior can be defined by a sequence of state transitions. By contrast, from a given state a parallel program can often make transitions to a set of states. For any given run of a parallel program, a sequence of state transitions in the program's state transition graph, of reachable states, will occur. Considerable research has been devoted to the problem of statically analyzing a parallel program to determine all reachable states (e.g., by analyzing the program flowgraph and synchronization operations)[McD89]. A simpler and more tractable problem is, given a time-stamped log of display and task events, animate the display events in an order feasible with the task synchronization events.

One possibility is to have the animation component perform state transition analysis. We believe that placing this duty on the animation component is unwise and overly complex, however. Therefore, we introduce the notion of an animation *choreographer* that analyzes program task and display events, then provides the animation component with a stream of animation events in a consistent time frame.

In simulations of physical systems there is often a "physical time" that can be used to determine the animation time-frame. If this physical time is available as a run-time state variable, it can be used to time-stamp program display events. For parallel programs lacking such a physical time, the animation must be based either upon global time, individual processor time-stamps, individual processor time-stamps with analysis of the relation between display events and synchronization events, or viewer control of animation speeds

of different tasks. It is the latter possibilities that give rise to animations of alternative program behaviors.

More specifically, the order of execution of tasks in a parallel program is controlled by the task states and the execution of synchronization events. It is the responsibility of the choreographer, by analyzing the global task synchronization state from the event log, such as which tasks are READY or RUNNING, which tasks are SUSPENDED or WAITing on an EVENT, to determine *feasible* animations. In order to ensure that feasible animations are correct, i.e., could have occurred under different rates of task execution, it is necessary that the choreographer is specific to the runtime system for task scheduling and events, and that the program does not use global memory for implicit synchronization (for example, Dekker's algorithm for mutual exclusion), that cannot be detected by the choreographer. Fortunately, parallel programs which make use of global memory for implicit scheduling are atypical, and generally examples of poor programming style.

Distributed memory multiprocessors, such as the Intel iPSC, use message passing for synchronization and communication. Shared memory multiprocessors, such as the Sequent Symmetry, use locks and events. Both architectures often provide higher-level synchronization, such as parallel loops for the Sequent, that can be defined using the low-level primitives. The choreographer will use a simple task-state model, in which tasks are either READY, RUNNING, BLOCKED (on a lock, event, or message) or SUSPENDED, together with target specific code for particular distributed memory or shared memory multiprocessors. We do not intend to target SPMD systems in this proposal, although the synchronous data parallel operations of an SPMD architecture would be simpler to choreograph than a multiprocessor.

The choreographer must "parse" program execution information and store it in a convenient form for later usage. Various strategies for the form's implementation range from databases[HC87, Sar90, KS91], to directed acyclic graphs[LMCF90], to program activity graphs[M$^+$90], to hierarchical structures[DBKF90]. We are examining all these approaches and determining if one is most useful for storing the information necessary to generate the highly application-semantic algorithm animations we display. Possibly, we will need to develop a new representation.

The notion of reproducing event sequences is recognized as an important tool for parallel debugging. Systems such as Instant Replay[LMC87] can recreate events within partial orderings according to the logical time[Lam78]. Others such as Bugnet[Wit89], designed for monitoring distributed systems, use periodic sampling methods to help reproduce the real-time ordering of processor events.

Helmbold, McDowell, and Wang[HMW90] have created an interactive tool for analyzing trace information in parallel Fortran programs. The system can determine race conditions and temporal orderings of task events. Our choreographer must handle these duties, as well as interpret display events, and coordinate this information with the animation component.

Hough and Cuny have addressed the notion of the ordering of abstract events in parallel programs[HC90]. Their work on *perspective views* provides algorithms for reordering events into logical units that are more useful for developing program visualizations. These results and algorithms should be useful in developing our animation choreographer.

Our plan for the choreographer is to emphasize the creation of a user interface tool that will allow a programmer to easily explore the feasible program behaviors. It will support direct manipulation of the task and display event-to-animation event mapping, e.g., using sliders to modify execution speed, etc. A good example of this is in the HyperView system[MRA$^+$89], primarily used for displaying hardware performance of programs on a

Hypercube. Frames can be single-stepped or run through quickly, and they are controlled by buttons and sliders. We want to make the same form of utility available for browsing semantic display events, and then controlling their mapping to the animation engine. The difficulty here lies in the interface from the choreographer to the animation component.

Some of the research issues being addressed via this section include

- How do we address the stream of program events emanating from the application program and map these to illustrative "animation ordered" events?
- What is the best abstract representation (e.g., database, graph, tree) of a program execution that allows users to map the information into the accompanying animation?
- What type of user interaction model and interface will allow users to easily encapsulate program execution and explore other feasible executions?
- How does the choreographer adapt to different parallel and distributed programming models and task event types?

4.3 Visualization/Animation Paradigm

The animation component of our system must address the seemingly opposing objectives of providing highly sophisticated graphical views and making their design as easy as possible. To achieve these goals, we are utilizing a derivative of the path-transition animation paradigm[Sta90a, Sta90b].

The path-transition paradigm supports the design of color, 2.5-dimensional (2-dimensions plus image layering due to color), continuous animation sequences through the use of four abstract data types: the graphical *images* on the screen, the *locations* that images and other objects occupy, the *transitions* that the images make, and the *paths* that modify the images' transitions.

The paradigm's fundamental idea is that the action of an image being modified along a path can be stored as an object (a transition) for subsequent use, both individually and in combination with other actions. Three key features of the paradigm motivate us to continue its use:

1. The notion of describing an action, such as movement or a change in size, by specifying an image and a control path is much more straightforward and easier to define than a series of "draw-erase" pairs with incremental calculations. This idea follows the pioneering *p-curve* work of Baecker[Bae69].
2. The paradigm supports animation routines which are data-driven, that is, animations able to be encapsulated in procedures receiving parameters controlling the animation's format, the object to be moved, the destination location of an object's movement, the size that an object should grow to, etc. Because the proposed visualization system animates programs, it must support animation routines which can adapt to various sets of application program input data.
3. The paradigm has well-defined semantics including rigorous definitions for all its data types and operations.

Unfortunately, the path-transition paradigm also maintains aspects which make it too restrictive to be used directly in our parallel visualization system. For instance, the paradigm

is relatively easy to extend (add new image and transition types) by the system designer, but it is not *user-extensible*. That is, animation designers cannot add their own basic images and transitions for use in new animation scenarios.

A second, and more subtle restriction, occurs in the design of overlapping animation actions. For example, consider an animation of ten objects (they can be balls, particles) moving inside a chamber. Suppose that ball 1 begins moving and just before it stops, ball 2 begins movement. Then, just before ball 2 stops, ball 3 begins moving, and so on. To describe this animation in the path-transition paradigm, all the movements must be combined into one complex transition, due to the small overlaps. Consequently, only after the last ball notifies of its completion, can the entire action be defined and performed. What we would like to provide in a parallel visualization system is a method for beginning the first ball's movement and then combining the others "on-the-fly."

One interesting general difficulty in animating parallel programs as opposed to sequential ones occurs when instantaneous program operations are illustrated in multi-frame animation actions. What does this imply about the ordering, timing, and display of subsequent program operations? We alluded to this problem in the previous two subsections. How do we insure that an animation does not falsely depict program behavior and timing, thereby misleading a viewer?

In order to address these issues and continue to focus on our primary goal of simplifying animation, we introduce a three-level framework for creating animations. The first level consists of a new animation paradigm, derived from the path-transition paradigm, but in a more distributed and formal object-oriented programming style. The second level includes a library of common algorithm animation actions, easily accessible to animation designers. The third and most important level involves the development of a direct manipulation graphical editing tool for designing animations by demonstration. Below we describe each level in more detail.

We have already begun work on a new animation paradigm that will be particularly well-suited to animations of parallel programs. The paradigm is being implemented in C++ and uses some simple techniques from the path-transition paradigm and many important new modifications for this particular problem. Figure 1 depicts a high-level overview of the model. It includes a central controller (class Animator) that receives all the animation events from the choreographer. The controller is a type of animation supervisor, maintaining global data and state information. The controller contains system code for handling event transmission and reception, as well as user-designed code for more semantic, application-specific actions. Through the controller, a designer creates one or more different views of the program (class View).

Each animation view is a user-designed conceptualization of the program being visualized. A view is broken into a set of animation scenes or procedures. For example, a sorting program's animation might have *display*, *compare*, *exchange* and *in-place* scenes. Each animation scene will create and/or manipulate the graphical objects in the animation. We have identified three important classes of objects necessary for the animation design: AnimObjects, Locations, and Actions. These classes are similar to the abstract data types of the path-transition paradigm, but with important differences to reconcile the difficulties mentioned earlier in this section. One key difference is that both AnimObjects and Actions will be user-extensible for greater flexibility.

In this new animation model, we also introduce an explicit animation frame time, an important change from the path-transition paradigm. The current time is maintained in each animation view, and is manipulated in the individual animation scenes. For example,

Figure 1: Configuration of the new object-oriented paradigm that will be the basis of the animation component.

an AnimObject can now be "programmed" to perform an Action at an explicit animation time. The animation designer can step through that time on a frame-by-frame basis. If a new Action arrives and should begin in the middle of the above Action, this new behavior can simply be programmed in. This new model replaces the notion of composition of transitions in the path-transition paradigm, which was problematic in cases like the one of balls moving inside a chamber as described above. We also plan to explore how to incorporate sound information[FAJ91] into the animation design kit.

The second level of the animation framework consists of a library of prototypical algorithm animation actions, the animation scenes noted earlier, available for use by designers. In a study of the many animations developed under the Tango system, we identified a small core of operations that predominated throughout. These operations, such as

- move an object to a location.
- change an object's color.
- exchange two object's positions.
- make an object flash.

made up over 90% of the actions shown in all the animations. By creating a library of these actions, invocable by a simple procedure call, animation development is further simplified.

The final level of the animation framework is a direct manipulation graphical editor that will allow an application developer to create an animation graphically and interactively. Because we want our system to be usable by programmers who do not have detailed knowledge of the animation paradigm, this demonstrational design tool is extremely important and vital to the success of the project. The main idea is that the system should not force a programmer to learn a new "language" and use textual specification. Rather, a programmer should be able to illustrate and focus on how an animation should look and act. After a

programmer demonstrates what should happen, the editor will automatically generate animation code in the object-oriented paradigm described earlier. The code should be directly compilable and loadable without further interaction. The intermediate code representation is advantageous, however, if a designer does wish to "tweak" the animation view.

To implement the direct manipulation design tool, we shall use our earlier experiences with the Dance graphical editor[Sta91] for the Tango system as a foundation. Figure 2 includes a window dump of the Dance editor being used to develop an animation. In Dance, each abstract data type has a convenient graphical depiction as well as a "pickable" label denoting its variable name. A designer can create objects, invoke operations through pull-down menu commands, and select objects directly to be used as parameters to operations.

The new tool must be tailored to the classes and objects of the object-oriented animation paradigm described earlier. It must also address deficiencies and problems in Dance. For example, in Dance users design animations on a scene-by-scene basis; animation data types are cleared between scene definitions. We have found that this is problematic because a designer loses context. Hence, the tool should preserve objects throughout an entire animation design. Also, the tool should allow for features such as abstraction, indexing, and iteration in a more consistent manner. Fundamentally, the tool must become more of a visual programming system[Cha87, Shu88].

Some of the research issues to being addressed via this section include

- How do we make animation creation as simple as possible, thereby allowing "average" programmers to develop their own visualizations?
- What is required in an animation toolkit that supports continuous movements, overlapping complex motion sequences, multiple views, user-extensibility, etc.?
- How do we create a direct manipulation graphical editor that promotes animation design without programming, but that still supports sophisticated complex animation scenarios?
- How do we intelligently coordinate the animation event information received from the choreographer?
- How do we help support animations that are true to their program execution bases, and do not somehow misrepresent what really happened in the program?

4.4 Implementation

The major implementation decision is the choice of a target parallel programming environment. We are using a Sequent multiprocessor, with Sequent's Parallel Fortran and C. The choice of Fortran is dictated by the widespread use of Fortran for numeric/scientific programming, the widely used parallel dialects of Fortran for multiprocessors, and the availability of tools for analysis and transformation of Parallel Fortran Dialects (PAT and IOS). We intend to use a test-suite of Parallel Fortran programs from Illinois, Rice University, Cornell Supercomputer Center, NASA-ICASE, and Los Alamos National Laboratory, and other centers with whom we have close contacts.

Supporting C as well as Fortran is possible because of Sequent's common multitasking library and PAT/IOS's C front-end. We intend our animation model to be language independent, although initially targetted to the shared memory multitasking model of parallel computation. The only language specific tools are those which automate the process of

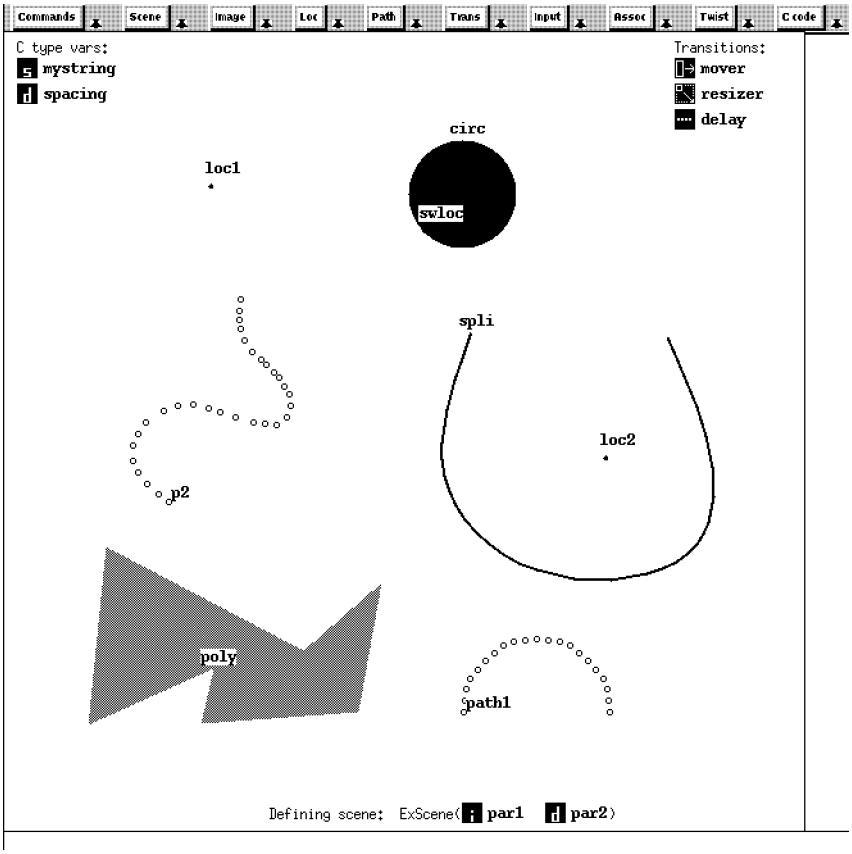

Figure 2: The Dance direct manipulation animation design editor being used to create an animation.

inserting calls to gather event data into the source code. Supporting C is advantageous as it is in widespread use in the systems programming community.

Secondary targets for our system basis are the Intel iPSC distributed memory multiprocessor and the BBN Butterfly. The College of Computing at Georgia Tech has a 16-processor Intel iPSC, a 32-processor BBN Butterfly, and 10-processor Sequent Symmetry available for this proposal.

Calls to log display events will initially be inserted by the programmer, though we intend to modify PAT later to help automate the process of locating the correct source line to insert display calls. This is relatively straightforward, as PAT performs dependence analysis which locates every variable access, and links every read/write to future read/writes of the same variable. PAT maintains the original source, and uses templates to generate new source code. Hence, given a user-selected variable, PAT can insert code around variable accesses selectively.

The choreographer will be a major implementation effort, though we will use techniques developed by McDowell, etc., for recreating the sequence task synchronization states from a trace file.

The development of an animation specification language will be based upon a revised and extended version of the path-transition paradigm, with a new distributed/interruptible method for animation. A major focus of the initial development is a semi-formal specification of the paradigm and animator environment, as was done with Tango.

Our implementation of the animation component is window-based using the X11 Window System. Although X windows does not provide ideal graphics speed for animation, its availability, portability, and network capabilities make it a practical choice. We are implementing both the animation choreographer and the graphical object class library in C++. This forces animation designers to code in C++, a restriction that we feel is mitigated by the extensibility that an object-oriented design offers.

5 Summary

Parallel program animation is intrinsically far more difficult and challenging than sequential program animation, yet more necessary, just as parallel program debugging is recognized as more difficult and challenging than sequential program debugging. Unlike textual debugging tools, which focus upon analyzing and comprehending program source code, animation tools focus upon comprehending overall program behavior, and by implication, the relationship between a program's behavior and the programmer's semantic model of the application.

Three key objectives must be met by our parallel program visualization tools:

1. The tools should not be restricted to creating visualizations of programs from a small set of application domains–they should be general purpose, not special purpose.

2. Developing parallel program visualizations should not be overwhelmingly complicated. If animations require weeks of development time, they are not useful.

3. A broad group of users must be involved in testing the tools and their interface. Both PAT and Tango have been widely distributed to dozens of other sites, primarily by anonymous ftp. Feedback from these sites has been used to improve the systems. We will follow the same approach with this work.

Sequential program data display, program visualization, and algorithm animation systems have become more sophisticated and increasingly accepted in recent years. Applying techniques developed in these systems to parallel program development is a natural and necessary next step.

References

[AB+88] F. Allen, M. Burke, et al. An overview of the PTRAN analysis system. In *Proceedings of the 1987 International Conference on Supercomputing*. Springer Verlag, February 1988.

[AB89] Allen L. Ambler and Margaret M. Burnett. Influence of visual technology on the evolution of language environments. *Computer*, 22(10):9–22, October 1989.

[AG89] Ziya Aral and Ilya Gertner. High-level debugging in Parasight. *SIGPLAN Notices*, 24(1):151–162, January 1989. (Proceedings of the Workshop on Parallel and Distributed Debugging, Madison, WI, May 1988).

[ASM89] Bill Appelbe, Kevin Smith, and Charlie McDowell. START/PAT: A parallel programming toolkit. *IEEE Software*, 6(4):29–38, July 1989.

[ASS91a] Bill Appelbe, Kevin Smith, and Kurt Stirewalt. PATCH - a new algorithm for rapid incremental dependence analysis. In *Proceedings of the 1991 International Conference on Supercomputing*, Cologne, Germany, June 1991. To appear.

[ASS91b] Bill Appelbe, Kevin Smith, and Kurt Stirewalt. Unbundling a compiler – interfacing parallel programming tools together. Submitted to Supercomputing '91, 1991.

[Bae69] Ronald M. Baecker. Picture-driven animation. In *Spring Joint Computer Conference*, volume 34, pages 273–288. AFIPS Press, 1969.

[Bas85] David B. Baskerville. Graphic presentation of data structures in the DBX debugger. Technical Report UCB/CSD 86/260, University of California at Berkeley, Berkeley, CA, October 1985.

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

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

[Bro88c] Marc H. Brown. Perspectives on algorithm animation. In *Proceedings of the ACM SIGCHI '88 Conference on Human Factors in Computing Systems*, pages 33–38, Washington D.C., May 1988.

[BW83] Peter Bates and Jack C. Wileden. An approach to high-level debugging of distributed systems. *SIGPLAN Notices*, 18(8):107–111, August 1983.

[Cha87] Shi-Kuo Chang. Visual languages: A tutorial and survey. *IEEE Software*, 4(1):29–39, January 1987.

[CKT86] Keith Cooper, Ken Kennedy, and Linda Torczon. The impact of interprocedural analysis and optimization in the r^n programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

[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.

[DBM89] Thomas A. DeFanti, Maxine D. Brown, and Bruce H. McCormick. Visualization: Expanding scientific and research opportunities. *Computer*, 22(8):12–25, August 1989.

[DCH88] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. *Computer Graphics: SIGGRAPH '88*, 22(4):65–74, August 1988.

[Dui86] Robert A. Duisberg. Animated graphical interfaces using temporal constraints. In *Proceedings of the ACM SIGCHI '86 Conference on Human Factors in Computing Systems*, pages 131–136, Boston, MA, April 1986.

[Dui87] Robert A. Duisberg. Visual programming of program visualizations. A gestural interface for animating algorithms. In *Proceedings of the 1987 IEEE Computer Society Workshop on Visual Languages*, pages 55–66, Linkoping, Sweden, August 1987.

[FAJ91] Joan Francioni, Larry Albright, and Jay Alan Jackson. Debugging parallel programs with sound. *SIGPLAN Notices*, 26(12):60–68, December 1991. (Proceedings of the ACM/ONR '91 Workshop on Parallel and Distributed Debugging).

[FLMC89] Robert J. Fowler, Thomas J. LeBlanc, and John M. Mellor-Crummey. An integrated approach to parallel progam debugging and performance analysis on large-scale multiprocessors. *SIGPLAN Notices*, 24(1):163–173, January 1989. (Proceedings of the Workshop on Parallel and Distributed Debugging, Madison, WI, May 1988).

[GJG+89] Vincent A. Guarna Jr., Dennis Gannon, et al. Faust: An integrated environment for parallel programming. *IEEE Software*, 6(4):20–27, July 1989.

[HC87] Alfred A. Hough and Janice E. Cuny. Belvedere: Prototype of a pattern-oriented debugger for highly parallel computation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 735–738, August 1987.

[HC90] Alfred A. Hough and Janice E. Cuny. Perspective views: A technique for enhancing parallel program visualization. Technical Report COINS 90-02, Univ. of Massachusetts, Amherst, MA, January 1990.

[Hea89] Michael T. Heath. Visual animation of parallel algorithms for matrix computations. In *Proceedings of the Hypercube Conference*, pages 735–738, 1989.

[HHK85] Paul K. Harter, Dennis M. Heimbigner, and Roger King. IDD: An interactive distributed debugger. In *Proceedings of the Fifth International Conference on Distributed Computing Systems*, pages 498–506, May 1985.

[HHR89] Esa Helttula, Aulikki Hyrskykari, and Kari-Jouko Räihä. Graphical specification of algorithm animations with Aladdin. In *Proceedings of the 22nd Hawaii International Conference on System Sciences*, pages 892–901, Kailua-Kona, HI, January 1989.

[HMW90] David P. Helmbold, Charlie E. McDowell, and Jian-Zhong Wang. Traceviewer: A graphical browser for trace analysis. Technical Report UCSC-CRL-90-59, Univ. of California at Santa Cruz, Santa Cruz, CA, October 1990.

[ISO87] Sadahiro Isoda, Takao Shimomura, and Yuji Ono. VIPS: A visual debugger. *IEEE Software*, 4(3):8–19, May 1987.

[KBI88] A. H. Karp and R. G. Babb II. A comparison of 12 parallel Fortran dialects. *IEEE Software*, 5(5):52–67, September 1988.

[KS91] Carol Kilpatrick and Karsten Schwan. ChaosMON–application-specific monitoring and display of performance information for parallel and distributed systems. *SIGPLAN Notices*, 26(12):57–67, December 1991. (Proceedings of the ACM/ONR '91 Workshop on Parallel and Distributed Debugging).

[L+89] Ted Lehr et al. Visualizing performance debugging. *Computer*, 22(10):38–51, October 1989.

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

[LD85] Ralph L. London and Robert A. Duisberg. Animating programs using Smalltalk. *Computer*, 18(8):61–71, August 1985.

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

[LMCF90] Thomas J. LeBlanc, John M. Mellor-Crummey, and Robert J. Fowler. Analyzing parallel program execution using multiple views. *Journal of Parallel and Distributed Computing*, 9(2):203–217, June 1990.

[LR85] Richard J. LeBlanc and Arnold D. Robbins. Event-driven monitoring of distributed programs. In *Proceedings of the Fifth International Conference on Distributed Computing Systems*, pages 515–522, May 1985.

[M+90] Bart Miller et al. IPS-2: The second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):206–217, April 1990.

[McD89] Charles E. McDowell. A practical algorithm for static analysis of parallel programs. *Journal of Parallel and Distribted Computing*, July 1989.

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

[Moh88] Thomas G. Moher. PROVIDE: A process visualization and debugging environment. *IEEE Transactions on Software Engineering*, 14(6):849–857, June 1988.

[MR88] Allen D. Malony and Daniel A. Reed. Visualizing parallel computer system performance. Technical Report UIUCDCS-R-88-1465, Univ. of Illinois, Urbana, IL, September 1988.

[MRA+89] Allen D. Malony, Daniel A. Reed, James W. Arendt, Ruth A. Aydt, Dominique Grabas, and Brian K. Totty. An integrated performance data collection, analysis and visualization system. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, March 1989.

[Mye83] Brad A. Myers. A system for displaying data structures. *Computer Graphics: SIGGRAPH '83*, 17(3):115–125, July 1983.

[Mye90] Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, March 1990.

[NE88] Kathleen M. Nichols and John T. Edmark. Modeling multicomputer systems with PARET. *Computer*, 21(5):39–48, May 1988.

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

[RC89] Gruia-Catalin Roman and Kenneth C. Cox. A declarative approach to visualizing concurrent computations. *Computer*, 22(10):25–36, October 1989.

[RMD89] Steven P. Reiss, Scott Meyers, and Carolyn Duby. Using GELO to visualize software systems. In *Proceedings of the ACM '89 SIGGRAPH Symposium on User Interface Software and Technology*, pages 149–157, Williamsburg, VA, November 1989.

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

[RRZ89] Robert V. Rubin, Larry Rudolph, and Dror Zernik. Debugging parallel programs in parallel. *SIGPLAN Notices*, 24(1):216–225, January 1989. (Proceedings of the Workshop on Parallel and Distributed Debugging, Madison, WI, May 1988).

[SA89] Kevin Smith and Bill Appelbe. Interactive conversion of sequential to multitasking FORTRAN. In *Proceedings of the 1989 International Conference on Supercomputing*, pages 225–234, Crete, Greece, June 1989.

[Sar90] Sekhar R. Sarukkai. Performance visualization and prediction of parallel supercomputer programs: An interim report. Technical Report 318, Indiana University, Bloomington, IN, November 1990.

[SAS90] Kevin Smith, Bill Appelbe, and Kurt Stirewalt. Incremental dependence analysis for interactive parallelization. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 330–341, Amsterdam, Netherlands, June 1990.

[SBN89] David Socha, Mary L. Bailey, and David Notkin. Voyeur: Graphical views of parallel programs. *SIGPLAN Notices*, 24(1):206–215, January 1989. (Proceedings of the Workshop on Parallel and Distributed Debugging, Madison, WI, May 1988).

[Shu88] Nancy C. Shu. *Visual Programming*. Van Nostrand Reinhold, New York, NY, 1988.

[SJ90] William J. Selig and James D. Johannes. Reasoning visualization in expert systems: The applicability of algorithm animation techniques. In *Proceedings of the Third International Conference on Industrial and Engineering Applications of Artifical Intelligence and expert Systems*, Charleston, SC, July 1990.

[Sta90a] John T. Stasko. The Path-Transition Paradigm: A practical methodology for adding animation to program interfaces. *Journal of Visual Languages and Computing*, 1(3):213–236, September 1990.

[Sta90b] John T. Stasko. A practical animation language for software development. In *Proceedings of the 1990 IEEE International Conference on Computer Languages*, pages 1–10, New Orleans, LA, March 1990.

[Sta90c] John T. Stasko. TANGO: A framework and system for algorithm animation. *Computer*, 23(9):27–39, September 1990.

[Sta91] John T. Stasko. Using direct manipulation to build algorithm animations by demonstration. In *Proceedings of the ACM SIGCHI '91 Conference on Human Factors in Computing Systems*, pages 307–314, New Orleans, LA, May 1991.

[Sto89] Janice M. Stone. A graphical representation of concurrent processes. *SIGPLAN Notices*, 24(1):226–235, January 1989. (Proceedings of the Workshop on Parallel and Distributed Debugging, Madison, WI, May 1988).

[Suk90] Piyawadee Sukaviriya. Coupling a UI framework with automatic generation of context-sensitive animated help. In *Proceedings of the '90 ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 152–166, Snowbird, Utah, October 1990.

[Wit89] Larry D. Wittie. Debugging distributed C programs by real time replay. *SIGPLAN Notices*, 24(1):57–67, January 1989. (Proceedings of the Workshop on Parallel and Distributed Debugging, Madison, WI, May 1988).

[ZPS89] M. Zimmerman, F. Perrenoud, and A. Schiper. Graphical animation of concurrent programs. In *Proceedings of the ACM/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 342–344, May 1989.

Appendix A, Example Visualization

In this appendix, we provide an example of our initial work on animating parallel programs. The program described here is the prefix sums problem, in which we are given a sequence of n numbers `Data =data[1],data[2],...,data[n]`, where $n > 1$, and must compute all n initial sums, `Prefix_Sums = s[1],s[2],..., s[n]`, where `s[i] = data[1] + data[2] + ... + data[i]`, for i=1,2,...,n. These sums are known as the prefix sums. The sequential algorithm is shown below.

```
procedure SEQUENTIAL SUMS (Data, Prefix_Sums )

Step 1:  s[1] <- data[1]

Step 2: for i=2 to n do
           s[i] <- s[i-1] + data[i]
        end for.
```

The prefix sums problem is of significance in that it represents a class of problems in which an associative binary operation is performed on a series of values. The summing procedure can be easily modified to solve any of the problems in this class.

We implemented a parallel version of the prefix sums algorithm on a Sequent, the algorithm for which is shown below. The algorithm assumes that the number of processors is a power of two. An additional array of per-processor sums is maintained, and each processor has a temporary variable.

```
procedure PARALLEL SUMS (Data, Prefix_Sums, N)

Step 1:  s[0] <- 0
         Per_Proc_Sums[0] <- 0
         partition the data into N chunks
Step 2:  for i=0 to N-1 do in parallel
   {
   within the partition, use the sequential prefix sums algorithm
   Per_Proc_Sums[i] <- s[j + offset], where j = partition size,
     and offset is the index of the first value in this partition
     of the Prefix_Sums array
   }
Step 3:  for i = 2 to N, by powers of 2, do in parallel
   {
   for k=1 to N do
       if k is a multiple of i
          tmp[k] <- Per_Proc_Sums[k] + Per_Proc_Sums[k - i/2]
   }
Step 4:  for i = 2 to N, by powers of 2, do in parallel
   {
   Per_Proc_Sums[i] <- tmp[i]
   }
```

```
Step 5:  for i= N/2 down to 2, by powers of 2, do in parallel
   {
   for k=1 to N do
       if k is a multiple of i, and totals[k] has not been updated
         in this step,
              Per_Proc_Sums[k] <- Per_Proc_Sums[k] + Per_Proc_Sums[k-i]
   }
Step 6:  for i=1 to N do in parallel
   {
   if odd(i)
      Per_Proc_Sums[i] <- Per_Proc_Sums[i] + Per_Proc_Sums[i-1]
   }
Step 7:  for i=1 to N do in parallel
   {
   for k=1 to partition size
       s[k + offset] <- s[k+offset] + Per_Proc_Sums[i]
             where offset is the index of the first value in
             this partition of the Prefix_Sums array
   }
```

We designed a graphical appearance for the program's animation, a still frame of which is shown in Figure 3. At this point the per-processor prefix sums have already been calculated, and the final prefix sum values are being calculated and placed into the array. (Of course, a static picture does not do justice to the smooth animation. Nor do the colors map well to black-and-white shading.)

We then implemented the important animation activities through animation scenes written in a new C++ and X11 graphics system called Polka.

Concurrently with the graphics development, we instrumented the parallel prefix sums source code so that it wrote out timestamped event information to an ascii text file. The event information included the event type and parameters such as processor number and values of variables. This allowed us to run the prefix sums program on one machine, and animate later on another. Each processor wrote to its own file, so there was no contention for the files by the processors. These files were then merged and processed into "Interesting Event" calls which would activate the appropriate animation routines, being developed concurrently.

The processing of log files must impose a "correct" ordering on the events, that is, an ordering which does not violate the "happened before" relation of Lamport time. The ordering may be as simple as sorting the events by timestamp, possible on a parallel computer, where each processor reads from the same clock. However, such an ordering may not give the viewer of the visualization the desired view of the program in execution. Merely sorting by timestamp will provide a serialized view of the parallel program. The viewer may wish to see the inherent parallelism more explicitly. Determining the overlap and duration of the events is the work of the choreographer we described earlier in the paper.

The process of developing the prefix animation (and a couple others) and viewing their animations on various input sets helped uncover a number of bugs in the programs' implementations. It also helped us understand better what a parallel program visualization system needs to provide.

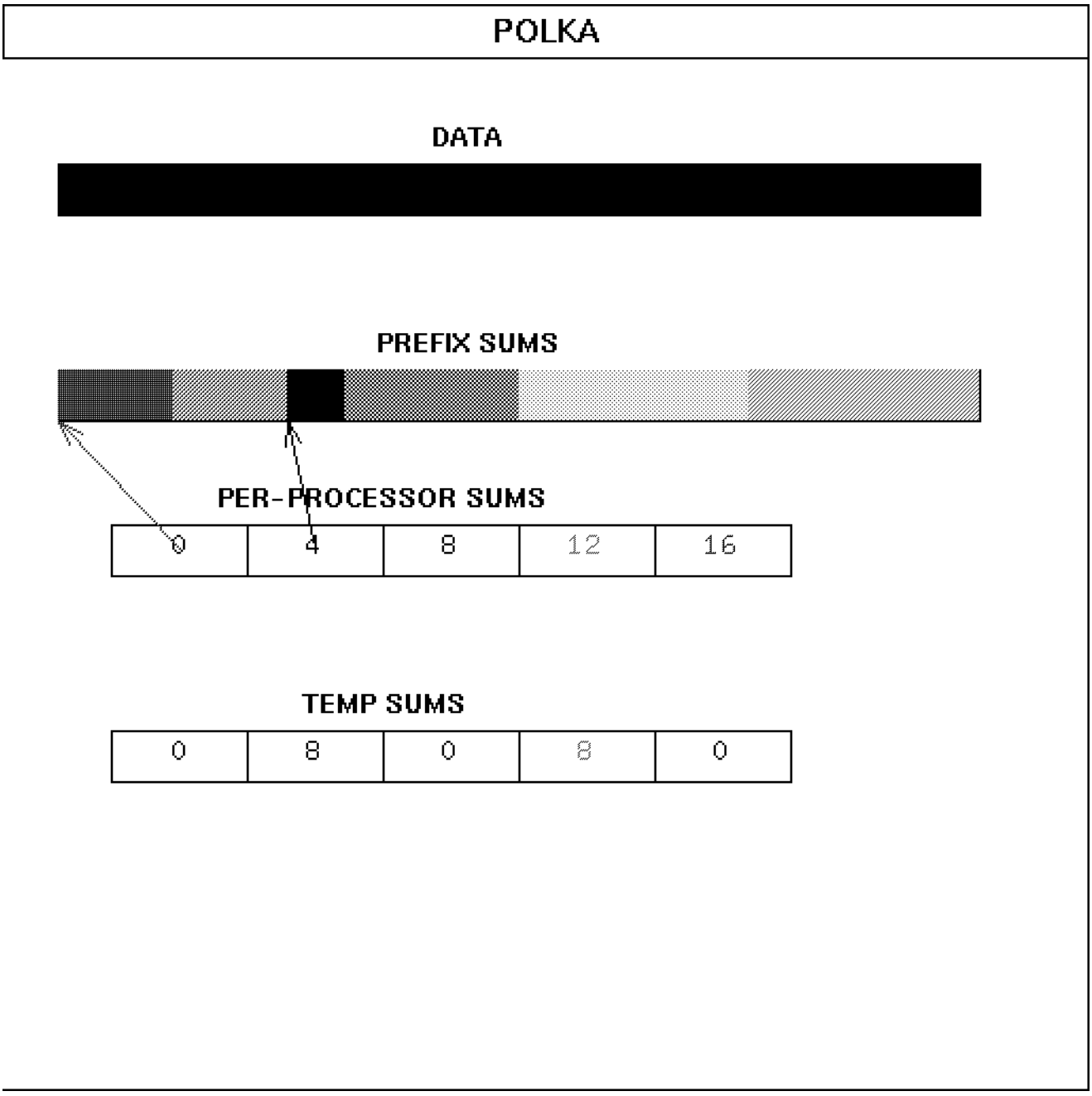

Figure 3: Still frame from the prefix animation.

These initial results have motivated us to improve the development process we undertook above and they have uncovered intersting new problems. For example, and "incorrect" program sending interesting events to the graphics engine will often send "nonsense" or error-laden events, thereby causing the graphics engine to crash. The choreographer, therefore, must act as an intermediary, monitoring what is translated through the program events.

Our early experiences also helped identify the need for animating program executions in different ways. This is another important feature of the choreographer. Initially, we plan to have the choreographer provide three types of time orderings:

1. Order by timestamp. This is possible only on systems in which there is a global clock. In distributed systems, the per- processor timestamp does not provide any information about the ordering of events between processor

2. Ignore the timestamps, and rely solely on the "happened-before" relation to order the events. Within a file, the ordering is sequential, as the events were written to the file by the processor in the order they occurred. The choreographer detects synchronization events in these files, and prevents the "happened-before" relation between processors from being violated in the visualization. The files are processed in a round-robin manner, with up to one event per processor being translated to Polka calls per round. This is done within the constraints of Lamport time, so that there may be fewer events occurring than there are processors in a given round.

3. Use the timestamps to divide the events into time-phases. The choreographer constructs a program graph, with a chain of events for each processor, and interconnections between events from different processors constructed based on synchronization events. The number of phases is based on the approximate number of desired frames in the visualization. Each phase is assigned a starting time. Any unprocessed event with a starting time less than or equal to the phase starting time which is permitted to occur within the constraints of the "happened-before" relation, is visualized. However, this still may not provide the desired visualization of the program in execution.

As described earlier, the choreographer should eventually allow the visualization designer/user to choose from a range of ordering methods. In addition to the above, the user will be able to view the program graph, each node of which will represent an event. The vertical position of the node on the screen will indicate the time phase at which the event will be visualized. The height of the node will represent the duration of the event, and the length of the connecting arc will represent the elapsed time between events.

In Hough and Cuny's paper on Perspective Views[HC90], they describe the visualization of an application running on a hypercube. In this visualization, nodes communicate in a sequence of phases, each phase crossing a different dimension of the cube. Without some intervention, the animation might show some processes working on the second or third phase of one iteration, while others may have progressed to the first phase of the next iteration. This animation would be difficult for the viewer to understand. The user might prefer to see the program's behavior in the logical view, in which the sets of processors are seen performing the event's actions in the logical phases. Hough and Cuny do this through the use of user-defined perspective views.

The choreographer should allow the user to manipulate the scheduling of the animation through direct manipulation of the program graph. For example, the user might click on a

node with animation scheduling to be altered, and drag the node up or down. The node should only move to locations which are within the constraints of Lamport time. These altered program graphs should be saveable and retrievable. The user could then view the resulting animations, either individually or simultaneously, with the result that the various schedules of the same animation may shed light on different features of the same program in execution.