A Methodology for Building Application-Specific Visualizations of Parallel Programs

Technical Report GIT-GVU-92-10

John T. Stasko
Eileen Kraemer

Graphics, Visualization, and Usability Center
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
Office: (404) 853-9386
Email: stasko@cc.gatech.edu

Abstract

Visualization of computer programs, particularly parallel programs, promises to help programmers better understand, develop, and debug their code, especially if the visualizations are relatively easy to create. We have developed a visualization methodology being used as a component in a comprehensive parallel program visualization system. The focus of the system is on application-specific user-tailored program views. An application-specific visualization of a parallel program presents the inherent application domain, semantics, and data being manipulated by the program in a manner natural to one's understanding of the program. In this paper we discuss why application-specific views are necessary for program debugging, and we list several requirements and challenges that a system for application-specific viewing should meet. The visualization methodology that we introduce includes primitives for designing smooth animation scenarios, and most importantly, for allowing designers to visualize or showcase the concurrency exhibited by parallel programs.

1 Introduction

Software visualization is the use of graphics to illustrate the methods, constituents, and purpose of computer algorithms and programs[Mye90, PSB92, SP92]. When the visualization is dynamic and it illustrates the semantics or abstract operations of a program, the visualization is often called *algorithm animation*[Bro88]. In this paper, we describe a new animation methodology particularly useful for developing dynamic visualizations of parallel programs, a relatively unexplored area of computer science. Visualization techniques can be applied to a number of activities involved in parallel program development including program design, performance evaluation, and debugging. The most successful application to date has been for performance evaluation and tuning. The focus of our work, however, is the use of visualization to assist debugging and correctness checking, an area that has not received as much attention as performance evaluation. Parallel program debugging remains an extremely challenging task, and tools to aid debugging are critically needed.

Visualizations for correctness debugging are different than those for performance evaluation because debugging requires *application-specific* program views. What do we mean by an application-specific view? This type of view illustrates the semantics of a program, its fundamental methodologies, and its inherent application domain. For example, an animation of a sorting algorithm should show the data values being exchanged. A visualization of Gaussian elimination should show the matrix of values as it is manipulated. A visualization of a particle simulation should show the particles moving about a chamber. In short, an application-specific visualization of a program should be recognizable as presenting the particular program or program class.

By presenting the execution of a parallel program in its inherent semantic format or application domain, a visualization system can provide programmers insight into the program's functionality. The same information could be acquired by examining program traces listing variables' values throughout execution, but this type of tracing is much more deliberate and it usually requires the programmer to make mental connections between variables' values and the program state at particular times.

Sarukkai and Gannon echo the importance of application-specific views in [SG92]:

> While it is convenient to have predefined visualizations of programs, the problem with such tools is that is is not easy to rapidly test new visualizations of the program execution. Further, most of these tools are strongly tied to the semantics of the trace data and hence cannot be used on similar data obtained from different machines or programming environments, without some effort.

Performance visualization differs from application-specific program visualization because performance views depict how efficiently a program is executing on a parallel computer. Performance views illustrate message passing, processor utilization, memory access, etc., and they are typically drawn from a library of graphical widgets, gauges, x-y plots, and charts. Figure 1 illustrates two examples of these types of views. Performance views, because they do not focus on the semantics of a particular program, can be reused for many different applications.

In the remaining sections of this paper we discuss what a system needs to provide in

Figure 1: Some examples of the style of views presented in performance visualizations.

order to support application-specific visualizations. We describe the visualization methodology/system that we have created to address those needs, and we give a few examples of animations of parallel programs created with the system.

2 Challenges and Requirements

Animating a parallel program is intrinsically more challenging than animating a serial program because of the non-deterministic nature of parallel programs. A serial program generates a stream of logical events that we can animate in their order of occurrence. In a parallel program, events of interest from the different processors are logged in separate streams that must be merged to create an animation. Providing a support model for coordinating the graphical elements that represent the parallel program's actions, or possible actions, is a primary challenge of a parallel program animation system.

Parallel program visualizations also are more difficult to create because of the simultaneity of program operations. In a serial program, only one logical operation occurs at a given moment, and the program's visualization reflects this. In a parallel program, many operations might be occurring "simultaneously," and the program's animation should illustrate this concurrency. Consequently, parallel program views will be more complex to generate, with many overlapping, concurrent graphical actions. A visualization system for illustrating parallel programs must be able to clearly present the concurrencies, or possible concurrencies, in the programs. The methodology described in this paper has such capabilities.

The ability to create application-specific views of programs is yet another challenge. Application-specific program views require unique graphics displays for each different class of algorithm or program to be shown. For example, a sorting view could be reused for many different sorting algorithms but it would differ dramatically from a view for graph programs. Therefore, an application-specific program visualization system cannot simply provide a predefined view library if it is to address a general set of programs. Rather, the system must provide a flexible graphics support paradigm.

One possible support platform that immediately comes to mind is a low-level graphics toolkit or library such as the X Window System's Xlib package. Unfortunately, this solution

is undesirable for a number of reasons. First, this type of library has a steep learning curve due to its complexity and size. Second, visualization development within such a complex environment is time-consuming, which is undesirable for building quickly-needed debugging views. Finally, such a library is not tuned to the domain of parallel program visualization. Because we know how the visualizations are to be used, we can eliminate some of the unnecessary functionality in a general graphics toolkit. We also can provide more extensive capabilities tuned to the particular display of parallel programs.

What are the requirements for application-specific parallel program visualizations? Below we list five capabilities that a support system should provide:

- The graphics system must provide basic 2D graphical objects such as lines, rectangles, circles, polygons, and text for its views. The system should support color views, and if possible, support 3D graphics for application programs involving 3D data.
- The graphical design and development methodology should be relatively easy to learn and use. Visualization creation and implementation should not require days or weeks of work. With that said, it would be naive to believe that sophisticated, application-specific views can be developed with little or no effort. Nevertheless, the visualization system should make every effort to foster ease-of-use.
- The graphics system should provide primitives for creating animations as well as visualizations. Programs are dynamic, time-varying entities. Animation helps illustrate how a program's execution proceeds. Continuous, synchronized motion 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 graphics system should help illustrate the concurrency inherent in parallel programs. One challenging aspect of parallel program development is control of concurrency and synchronization. If a graphics system has primitives to help illustrate programs' parallelism via concurrent graphical motions and actions, it will greatly assist program debugging.
- The graphics system should promote a straightforward mapping mechanism by which a programmer can associate program objects and actions with the graphical objects and the transitions they undergo. The system should be able to visualize programs from a variety of system models and architectures.

3 Animation Methodology

We are building a comprehensive program visualization system called PARADE (PARallel program Animation Development Environment) for developing animations of parallel programs[AS91, SAK91]. The system contains components for 1) extracting and formatting program event information; 2) mapping and restructuring the event information as input to the animation component; 3) creating the animated graphical program views. The

Figure 2: PARADE system overview, highlighting the three major components.

third component, our animation methodology, is the focus of this paper. Figure 2 illustrates the relationships of these three components.

To understand how the methodology fits in PARADE, first we'd like to briefly mention the other two components. In the first component, extracted program information can be either automatically generated system events such as control, message, and synchronization events, or it can be programmer annotated application-specific program events. We currently write the events to trace file(s). We also are exploring the use of temporal databases[Sno88] to store and manipulate event information, and the use of on-line (pseudo real-time) processing and display. In all of these methods, we do assume that the parallel program being animated and its animation are in separate process spaces so that they cannot share data structures.

The second component, which we call the **animation choreographer**, gathers the program events that have been logged in separate trace files and structure them according to user preferences. The animation choreographer displays an execution history graph based on the trace events, including synchronization events. The user interacts with the choreographer display to control the ordering of the display events and the relative speed of the displays. One moment a viewer may wish to view program execution as reflected by program maintained application-level time. Another moment the viewer may want to view program execution as related to a global system clock time. In other instances the viewer may wish to observe program behavior according to logical precedences, so-called Lamport time[Lam78], or even under some alternative feasible execution ordering. The animation choreographer, much more so than in serial program animations, is an absolutely crucial element in the system.

The third component of the system, the focus of this paper, is a visualization methodology to address the five requirements for application-specific viewing described in the preceding section. The methodology is called POLKA (Parallel program-focused Object-oriented Low Key Animation) and it is an object-oriented basis of visualization and animation that

Figure 3: The different types of classes involved in a POLKA animation. The levels in the hierarchy illustrate the "has-a" relationships.

includes high-level graphical object and motion primitives. We have implemented both 2D (on top of the X Window System) and 3D (on top of Silicon Graphics GL) versions of the methodology in C++. POLKA provides two critical features: 1) It supports *true* animation–By that we mean smooth, continuous movements and actions, not just blinking objects or color changes. 2) It supports *concurrent*, overlapping animation actions that can properly reflect the concurrent operations occurring in a parallel program.

Figure 3 illustrates the different classes of objects used in a POLKA program animation. Each program to be animated requires a top-level **Animator** object. Actually, a programmer subclasses POLKA's Animator class and builds a derived Animator specifically for a program or class of programs. The Animator class contains data members and member functions that help map program events to their corresponding actions. An Animator contains methods for registering and receiving program events, defined by string names and trailing integer, real, and/or string parameters. An Animator also must contain a *Controller*[1] method, which specifies how to react to different program events (in other words, which graphics routines should be called.) Currently, we automatically generate the *Controller* given event and animation routine descriptions. Eventually, in PARADE, the Animator's capabilities will be moved to the animation choreographer.

Each Animator includes one or more user-designed program **View** data members. Program Views are subclassed from a base View class and each is a window onto the program's execution, providing a specific graphical appearance or visual representation. Each view must have a number of animation *scenes* (member functions) defined for it. These scenes help distribute the entire animation among a number of smaller, more manageable units. The base View provides one primary method, *Animate*, and a primary data member, *time*. The variable *time* maintains the View's animation frame count, or time, which starts at zero. The *Animate* method generates a specified number of new animation frames.

A View contains three primary classes of objects that developers instantiate and manip-

[1] By convention, we will present class methods and data members in slanted typeface throughout this paper.

ulate to design a visualization: **AnimObject**, **Location**, and **Action**. The basic idea of the animation methodology is that programmers position AnimObjects (graphical entities such as lines, rectangles, ellipses, spheres, text, etc.) within the View coordinate system. To help achieve action or motion, programmers instantiate Action objects which have type, duration, and modifier attributes. For example, a simple Action might be a leftward movement of 0.4 units for 10 frames. Programmers assign Actions to AnimObjects using the AnimObject *Program* method. For example, we could *Program* the Action mentioned above into an Ellipse AnimObject to begin at frame 3 of an animation. To generate animation frames, a programmer uses the View's method *Animate*. Each View is broken into a number of animation scenes (procedures or functions) which create and manipulate these three object types. Typically, once some initial imagery is established, an animation is played by alternating through sets of *Program* and *Animate* actions as more information about a program's execution becomes available. Below we highlight a few important details about the View constituent objects.

AnimObject: An AnimObject is the base class for all types of graphical objects; it provides a set of default object method handlers. Some sample derived classes include Rectangle, Line, Circle, Spline, Polygon, Text, and in the 3D version Sphere, Cube, Cone, etc. POLKA is designed so that end-users can derive their own AnimObject subclasses as well. When an AnimObject is constructed, its attributes are specified and it is merely added to the set of available objects. To have the object appear in animation frames, the programmer must explicitly use the *Originate* method, with a given frame time argument, to map the object's graphical representation to its View. To remove an object permanently from a View, programmers must use the *Delete* operation which also takes a frame number parameter. This is useful because programmers may need to specifically remove objects at particular future frames. POLKA also provides an AnimObject of type Set for associating groups of objects together and referencing them as one.

Location: Locations in POLKA can be used to reference and remember important positions for later use. They are $\langle x, y \rangle$ markers in the real-valued View coordinate system.

Action: An Action is an object encapsulating a simple movement or change that can be applied to an AnimObject. An Action has both a type, which is simply a string identifier such as "MOVE," "COLOR," or "RESIZE" and a list of $\langle x, y \rangle$ offset pairs, defining a two-dimensional sequence in the View coordinate system. In more complex 3-D systems such as [Z+91], the offset lists or paths can consist of control points of varying types such as vectors, expressions, and strings. For our simplified world, restricting the control points to two real numbers, which correspond with the View coordinate system, is practical, efficient, and advantageous.

POLKA's animation methodology is derived from a combination of principles of the path-transition paradigm[Sta90a] of the TANGO algorithm animation system[Sta90b] in which designers create images and modify them along paths or two-dimensional trajectories and also from the techniques of more traditional production 3D animation systems.

POLKA's animation methods differ critically from TANGO's path-transition paradigm in how animation actions are defined and executed. Using the path-transition paradigm, programmers create transitions (objects encapsulating change which include image and path components) and then they perform these transitions, which execute to completion. In order to achieve simultaneous actions on one object such as changes in color, size, and

position, or to have multiple images changing simultaneously, transitions must be combined together using the compose operation into a new, more complex union transition that later can be performed. This model works well for serial programs. For parallel programs, designing overlapping actions through composition quickly becomes extremely complicated to maintain. In the next section, we will expand upon this point in the animation examples.

In POLKA, an AnimObject is *Programmed* (a formal AnimObject method) with an Action at a particular View frame time. The frames of the View's animation then are generated using the *Animate* method. A programmer has total control over when to allow new animation frames to be generated. The *Animate* method checks all AnimObjects within a View, and if they have Actions programmed to occur at the current frame, the AnimObjects are sent *Update* and *Draw* messages.

Designing overlapping motion sequences in POLKA is simple because individual objects are programmed independently without any references to other objects. No extra work is required to specify how to combine the objects' Actions. For the animation examples described later, this capability is absolutely essential.

Implementation

POLKA is implemented in C++. The 2D version is built on top of the X Window System and the 3D version is built on top of Silicon Graphics GL. POLKA is relatively small, about 5000 lines of code, thus supporting our goal of keeping the system compact. POLKA provides Animator, View, AnimObject, Location, and Action classes. Animator and View classes are not used directly; programmers derive their own subclasses, inheriting methods such as View's *Animate*, and redefining other virtual functions such as the Animator's *Controller*.

We separate an AnimObject and its visual appearance into two different classes, with one implicitly created during the other's construction. We implement AnimObjects and their subclasses using a parallel class hierarchy in C++[Mey92], making the AnimObject class a "handle" or "Cheshire Cat" class. Each AnimObject contains a pointer to an AnimObjectImpl class, or more precisely, one of the specific subclass implementation objects that defines its visual appearance. This separation is valuable for allowing AnimObjects to change their appearance (line, circle, rectangle, etc.) dynamically. Our animation example later will reinforce this point.

The primary sequence of creating an Action, programming the Action into an AnimObject, and then generating the animation is illustrated in the brief code sample below.

```
// We are within a View animation scene method.
// Assume the following types and that the
// Locations and Rectangle have initial values.
//
// Location l1,l2;
// Rectangle *r;
// Action *a;
// int len;
```

```
   // Create a straight movement action from location <l1> to <l2>.
a = new Action("MOVE", l1, l2, STRAIGHT);
   // Parameters are Action type, from loc, to loc, path pattern.
len = r->Program(time, a);
   // Action <a> programmed at time <time>.
   // Program returns Action's length in frames.
   // <time> is a data member of View.
time = Animate(time, len);
   // Animate for <len> frames starting at
   // time <time>.  Return the new <time> value.
```

The variable *time* starts at zero and is updated using the *Animate* method. Here, we have generated a number of frames in the *Animate* call, but we could have just as easily generated one frame at a time. This is important if new program activities occur subsequent to the initial frame generated by *Animate* but prior to the last frame it generates.

POLKA, taken at a basic level, does not completely meet our goal of being easy-to-use. The methodology of the animation paradigm must be learned, and admittedly, it is C++. But we have found the system to be quite accessible, much easier for programmers to learn than X Windows programming, for example. Graduate students not well versed in C++ have successfully developed their own program animations quickly. The 3D version of POLKA, in particular, provides many default parameters and simplifications so that designers need not worry about specialized, but usually superfluous, graphics details. Programmers need not know 3D graphics techniques such as shading, ray-tracing, and so on, in order to create a 3D visualization.

Most importantly, however, POLKA has constituent primitives specifically for continuous animation, a capability not found in other systems. Smooth incremental animations of programs help to preserve context and promote comprehension. For instance, rather than making the nodes of a graph flash as they are visited in a program, a graph tour can be illustrated by a lozenge that smoothly traverses both graph nodes and edges. POLKA helps remove the difficulties inherent in specifying these types of animations.

4 Example Animations

This section describes a few example application-specific program animations that we have built using POLKA.

4.1 Prefix computation

One of the first animations we developed was that of the prefix sums problem, in which a sequence of n numbers is provided and the object is to compute all n initial sums of the sequence. We implemented a parallel version of the prefix sums algorithm on a Sequent. The program assumed that the number of processors was a power of two with each processor maintaining an additional array of sums.

To begin development, we designed (conceptually) a graphical appearance for the program. (A view from the resultant animation is shown in Figure 4.) The visualization

contained four structures of interest: The array into which the data was initially read and the array in which the prefix sums were placed were both of size n, the number of values entered. The other two arrays of size $N + 1$, where N was the number of processors, were used to represent the per-processor sums and the per-processor temporary-sums.

We used a horizontal rectangle to represent each of the arrays. The per-processor sums and temporary-sums arrays were partitioned into $N + 1$ sections, each of which contains a textual display of the element's current value. The data and prefix sums arrays, containing many more elements, were not partitioned nor were the values of individual elements displayed.

As the initial values were read into the data array, a black rectangle grew in the data array from left to right, eventually filling the structure with black when all values had been read in. As the program continued, each processor read the initial data values from its assigned section of the data array. A running total was maintained and the current total placed into that processor's current element of the prefix sums array. This was represented graphically as a rectangle, colored according to processor-ID, shown growing in each section of the prefix-sums array. Half-tone shading was used here to indicate that the sums placed into the array were not complete, but merely the per-processor prefix sums.

When a processor completed its section of the prefix-sums array, its total was displayed in the per-processor sums array. An arrow from the prefix-sums array to the per-processor sums array showed the connection, as did the use of appropriately colored text labels.

As the per-processor sums were combined to calculate an initial prefix sum for each processor, the movements of values within and between the per-processor sums and temp sums array were shown using color-coded arrows and moving text. The text moved in a slow arc from one partition to another so that the user could identify both the source and the destination. This portion of the visualization highlighted several errors in our initial implementation of the program, including the erroneous use of a barrier synchronization within a conditional statement.

Once each processor had received its initial prefix sum, this value was added to each element in that processor's section of the prefix-sums array, completing the calculation. The progress of this final addition was again indicated by a growing, color-coded, horizontal rectangle. At this point, full shading was used to indicate that the final values are present.

We implemented this animation view and scenario through 10 POLKA animation scenes. Eight of the routines, the most important ones, were associated with the movement of data from one structure to another, indicating the value, the indices, and the processor. An "Init" routine began the animation by indicating the number of processors and the number of data elements, and it created the objects representing each of the arrays. A "Done" routine performed a final clean-up of objects.

Concurrently with the graphics coding, we instrumented the developing parallel prefix sums source code so that it wrote out timestamped event information to ascii text files. The event information included the event type and parameters such as processor number and values of variables. Each processor wrote to its own file, so there was no contention for the files by the processors. These files were then merged using other components of our system and processed into "Display Event" calls which activated the appropriate animation routines.

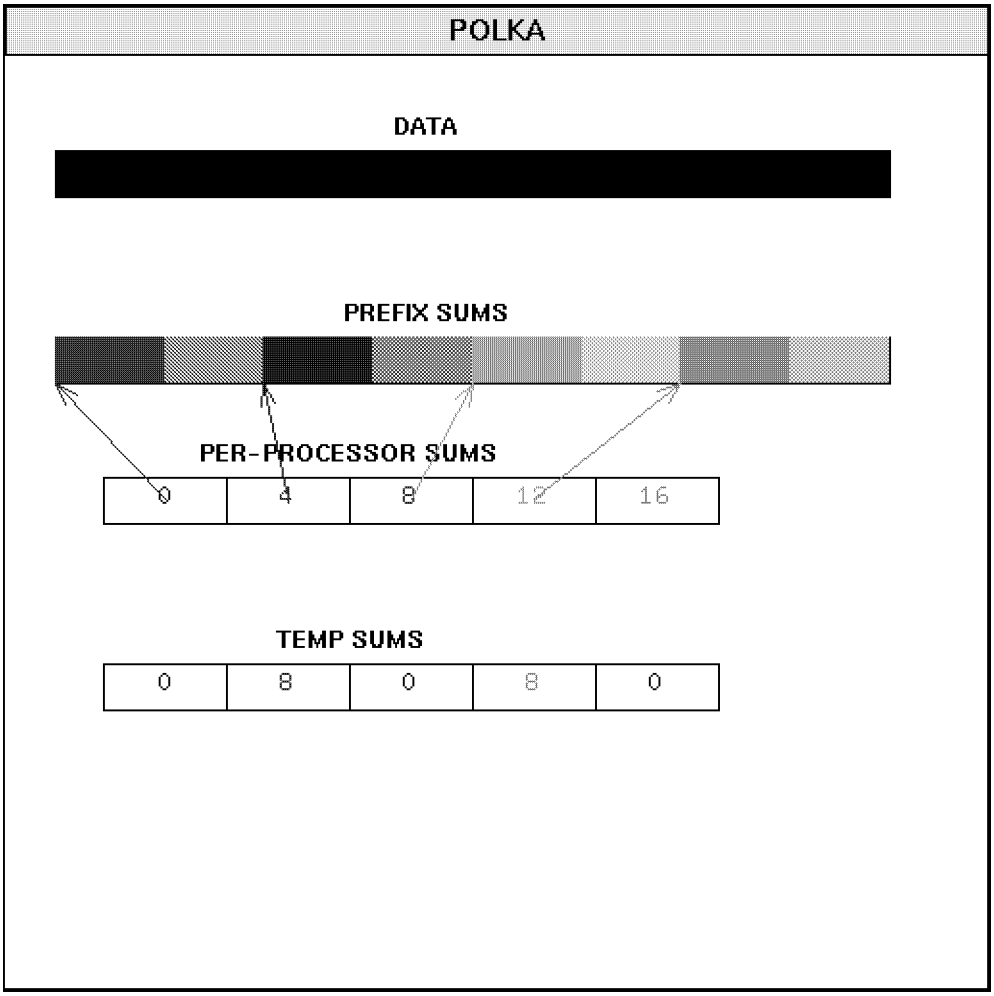

Figure 4: A still frame from the prefix animation. At this point the per-processor prefix sums have already been calculated, and the final prefix sum values are being calculated and placed into the array. (Of course, a static picture does not do justice to the smooth animation. Nor do the colors map well to black-and-white shading.)

We implemented the POLKA animation routines before the program's implementation was finished. Consequently, we were able to use the animation as a visual tracing aid. The process of developing the prefix animation and viewing its animations of executions on various input sets helped uncover a number of problems and bugs in the program's ongoing implementation. This and other early animations we developed also helped us understand better what an application-specific parallel program visualization system needs to provide.

4.2 A 3D Particle Simulation

This section discusses a program that simulates a particle chamber with many particles bouncing around inside a container or box. As each particle strikes a wall, it instantaneously changes velocity and then maintains that velocity, moving in a straight line, until it strikes another wall. The program maintains its own specific time scale. For simplicity, we ignore particles striking each other within the chamber. (That capability would add complexity to the program itself, not the POLKA animation.)

The animation of a parallel program depends upon the style of events logged in the program's trace file(s). In this example, we assume a model which records particle-wall collision events and the times that they occur. Specifically, the particle program issues four types of events. The first event initializes the animation. The second event initializes each particle. The third event is reported whenever a particle strikes a wall. The final event, called Release, is reported once each clock cycle. The Release event reports the maximal clock time up to which all particles could be traced and animated. This reported time is the minimal (earliest) time of all particles' most recent collision times. That is, the reported time is the time (relative to zero) to which it is possible to interpolate particles' positions based on their collision events already reported in the trace files. Later in this section we will discuss alternative event models and their ramifications upon the animation design code.

Our animation of this program includes four animation scenes, SetUp, InitPart, Collision, and ClearTime, that correspond directly to the program's events. That is, the Animator's *Controller* will set up a one-to-one mapping between program events and animation scenes. The SetUp scene initializes View information such as the mapping from the program's integer coordinates to the View's real coordinates. In the InitPart scene, we create an AnimObject (initially a rectangle) to represent each particle. In the Collision scene, we use the particle's previous collision position and the new collision position reported to the scene in order to calculate the appropriate wall-to-wall movement Action. We then program the Action into the specified particle's AnimObject. For this example, we simply map each program clock cycle to an animation frame. In the ClearTime scene, which is invoked for each clock cycle of the program, we call View's *Animate* method to generate the required number of frames, and we update the View *time*.

Below is the code for the particle program described using a mix of pseudo-code and a parallel C style language. Comments indicated by "//" would have the appropriate code segments filled in for the actual program. The highlighted statements are those inserted into the original particle program to generate the trace events reported to POLKA.

```
main()
{
   int x[MAX], y[MAX], z[MAX], vx[MAX], vy[MAX], vz[MAX] ;
   int started[MAX], clock=0;
   int num, winsize, bounced, mintime, i;

   // Initialize program data such as winsize

   WriteAlgoEvt("Init", winsize);

   printf("Enter number of particles\n");
   scanf("%d",&num);

   for (i = 0; i<num; i++) par_do { // in parallel
      // Assign particle[i] an initial position x[i], y[i], z[i]
      // and an initial speed vx[i], vy[i], vz[i]

      WriteAlgoEvt("Originate", i, x[i], y[i], z[i]);

      started[i] = 0;
   }

   for ( ; ; ) {
      clock++;
      for (i = 0; i < num; ++i) par_do { // in parallel
         // Update particle[i]'s position
         bounced = 0;
         if (particle[i] has struck any wall) {
              // change its velocity
              bounced = 1;
              started[i] = clock+1;
         }

        if (bounced)
             WriteAlgoEvt("Collision", i, clock, x[i], y[i], z[i]);

      }

      // Calculate min start time of all particles
      // held in started[] array
      mintime = parallel_min(started, num);

     WriteAlgoEvt("Release", mintime);

   }
}
```

The POLKA animation of the particle program, a snapshot[2] of which is shown in Figure 5, includes refinements of the Animator and View classes. The Particles View contains

[2]Of course, this frame does not adequately reflect the dynamics and smoothness of the real animation.

Figure 5: A frame from the particle chamber simulation creating using POLKA. Some of the particles have changed from rectangles to circles and have changed color (simulated by fill patterns for this figure), as described later in the paper. Although this view is simple, it reflects the physics of the simulation program quite well. (Figure missing)

its animation scenes and important data members that are manipulated by all the scenes. The data members include the array *part* which holds the AnimObjects (actually pointers). The variable *factor* helps to convert from integer program coordinates to real-value View coordinates. The arrays *px* and *py* maintain the previous position of each particle when colliding with a wall. When a subsequent collision occurs, we use the previous position as the "from" point of a computed path. The array *release* holds the animation frame time for the start of a particle's current movement.

```
class MyAnimator : public Animator {
   public:
      int Controller();
   private:
      Particles p;  // The only View in this animation
}

class Particles : public View {
  public:
      int SetUp(int);    // 4 animation scenes
      int InitPart(int,int,int,int);
      int Collision(int,int,int,int,int);
      int ClearTime(int);
   private:
      double factor;
      double px[MAX],py[MAX],pz[MAX];
      int release[MAX];
      Cube *part[MAX];
}
```

The View scene SetUp merely sets some initial parameters for the view and calculates the integer-to-real mapping *factor*.

The InitPart scene fixes the initial position and release time of a particle, then it constructs the initial rectangle AnimObject for the particle.

```
int
Particles::InitPart(int num, int x, int y, int z)
{
   px[num] = double(x)/factor;  // Initial position
   py[num] = double(y)/factor;
   pz[num] = double(z)/factor;
   release[num] = 1;      // Start time of
                          // first move

   part[num] = new Rectangle(this, vis,
           px[num],py[num],pz[num],   // Initial position
           xsize,ysize,zsize, color, fill);
   part[num]->Originate(0);  // Add the object
              // to the display list for time 0
   return(1);
}
```

The Collision scene receives particle number, time, and position arguments. It creates a movement action from the particle's previous position to the new position, and it programs

the movement Action into the particle (but it does not generate any animation). Finally, it must update the position and release fields for the particle.

```
int
Particles::Collision(int num, int clock,
                     int x, int y, int z)
{
   double dx,dy,dz;

   dx = double(x)/factor;  // Scale
   dy = double(y)/factor;
   dz = double(z)/factor;

   Loc from(px[num], py[num], pz[num]);  // Previous position
   Loc to(dx, dy, dz);              // New position

   Action a1("MOVE", &from, &to,
                 clock-release[num]+1);
     // clock-release+1 total frames in the Action

   part[num]->Program(release[num], &a1);
     // Starts at time release[num]

   release[num] = clock+1; // Update to new values
   px[num] = dx;
   py[num] = dy;
   pz[num] = dz;
   return(1);
}
```

Finally, the ClearTime scene receives an animation time up to which it is safe to animate.

```
int
Particles::ClearTime(int clock)
{
         // If we have a new advanced time,
   if (clock > time)
      time = Animate(time, clock-time);
         // animate from <time> to <clock>
         // for <clock>-<time> frames
   return(1);
}
```

This relatively simple POLKA code creates an informative, useful animation of the particle program. Adding embellishments to the View is easy also. Suppose that the Collision scene receives a color parameter to which the particle should be changed. (Perhaps this illustrates processor ID in the parallel program, and which processor is manipulating the particle.) By adding the code

```
   Action a2("COLOR", color);
   part[num]->Program(clock, &a2);
     // Recall that clock is a parameter
     // to the Collision scene
```

to the Collision scene, we achieve the desired effect.

Suppose also that we desire to change the shape of a particle from a Rectangle to a Circle at some point in the animation. AnimObjects in POLKA receive this functionality through the *Change* method. An AnimObject is a constituent entity throughout an animation. But its physical appearance, be it Line, Circle, Text, etc., is merely a modifiable attribute of the object. Below we simply construct a new appearance (but do not *Originate* it!) and *Change* the AnimObject appropriately.

```
c = new Circle(this, vis, dx, dy, dz, radius, color, fill);
  // Now copy physical attributes
part[num]->Change(clock, c);
```

This capability is extremely valuable in parallel program animation. Suppose that we have programmed an object to move across a View window, and subsequently, because of processor delays affecting event logging, we learn that the object should change appearance halfway across. In using the Change command, an AnimObject retains its programming throughout any appearance changes. Consequently, POLKA will illustrate the desired effect of the object moving, changing appearance halfway across, and continuing on its original path. This capability does not exist in the path-transition paradigm, and it is a key improvement in POLKA.

Even the simplest level of the animation described above would be exceptionally difficult to implement with the path-transition paradigm. There, movements or transitions with overlapping frames must be composed together to achieve simultaneous display. In the particle simulation, an arbitrarily complex set of asynchronous, overlapping actions occurs, and it would require one super-transition in the path-transition paradigm that could only be performed at the very end of the program's execution. The POLKA model of programming Actions and incrementally generating animation frames is much better suited for these types of situations.

The event model used in this example makes the animation design task trickier since positions of the particles are not reported regularly at each clock cycle. In reality, we may be forced to use an event model of this type. An alternative simpler event model, sending an Update-Position event with location parameters for each particle after each program clock cycle, would actually make designing the animation in POLKA easier. (We have also implemented the animation using this method.) We described the animation based on the first event scheme to illustrate POLKA's flexibility, power, and how it handles complex, overlapping actions.

4.3 Other Animations

Figure 6 shows a frame from a 2D sorting animation developed with POLKA. The sorting animation represents the data values as rectangles with their heights corresponding to the relative values of the elements. When an exchange occurs, the two rectangles smoothly swap positions and change color to indicate which processor was responsible for the exchange. The sorting animation's routines consisted of about 80 lines of POLKA code.

POLKA also can be used for animating more than only the semantic behavior of parallel algorithms. Essentially, it is a general visualization environment for all features and types of

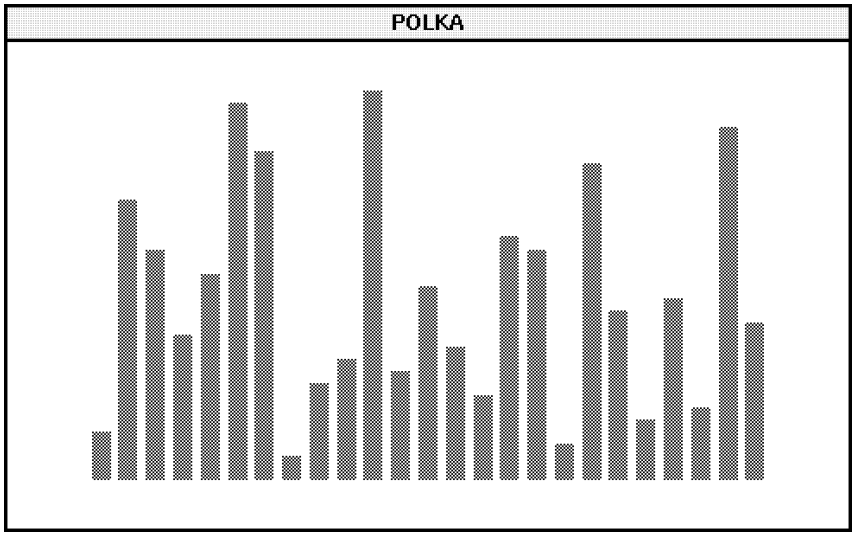

Figure 6: A frame from the sorting program animation.

programs. Figure 7 shows a view of a Kiviat diagram[KK73] created using POLKA. A Kiviat diagram (this View is modeled after one from the Paragraph system[HE91]) illustrates a set of processors as spokes on a wheel with each processor's recent average utilization reflected as distance along its spoke. When a processor is idle, its spoke is at the center of the wheel. When it is completely busy, the spoke is extended to the outer edge. This view is really more of a performance visualization than a program visualization. POLKA's animation methodology also serves equally well as a platform for developing these performance views or even algorithm animations of serial programs.

5 Related Work

Visualizing Parallel Programs

A few existing systems have addressed the need for application-specific program views. The ParaGraph system[HE91], best known for its predefined library of performance views, contains facilities for application-specific views to be added and run under the infrastructure of ParaGraph's existing support environment. This task requires basic X Window System programming, however.

Many of our goals mirror those of the system Voyeur[SBN89] which focuses on supporting easy-to-create views matching a programmer's mental model of her code. Voyeur uses a class hierarchy of views, so that derivations of views are easy. It does not appear that many types of views were implemented, however. Creating a new view, as is common in application-specific visualization, required X Window programming.

The system presented in [LMCF90] provides many different styles of program views. Their *external* view corresponds to the application-specific visualizations we seek. To create an external view, a programmer must extract information from an execution history graph,

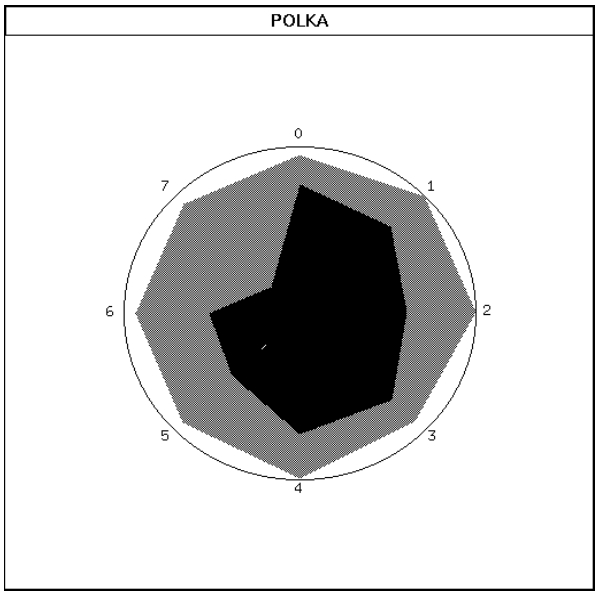

Figure 7: Kiviat diagram, created using POLKA, that shows average processor utilization over a given time interval.

design a view in a Common Lisp environment, and then input the information to other visualization tools.

The Pavane system[CR91], which operates on a shared-memory tuple space architecture, supports formal mappings from a program space to an abstract animation space that can later be rendered. This model supports highly application-specific program views. To describe a visualization, programmers must construct the mappings, which are similar to those in formal proof systems.

The IVE system[FLK+91] supports visualizations of programs on massively parallel SIMD machines. These *process visualizations* are developed using visualization templates, parametrical graphical object descriptions that can be created by refining existing templates or by using a CAD-style tool. This model appears to support static visualizations much more so than animations.

SIEVE.1[SG92], although specifically designed for performance visualization, supports program views that can be considered application-specific. Views are designed using a spreadsheet programming model. This paradigm works well for views that can be described from an algebraic specification of program execution data.

Object-oriented Graphics

The use of object-oriented techniques for computer graphics has seen increased attention recently. The GROW system[Bar86] uses taxonomic inheritance and constraints to help users build program interfaces, most often graphical editors. The InterViews Graphical Toolkit[VL88] contains an extensive hierarchy of graphical objects (written in C++) and a powerful set of methods for manipulating these objects. Many different applications such as user interface builders and graphical editors have been implemented using Interviews.

POLKA differs from these systems in its focus on object classes for animation rather than static graphics. The algorithm animation system Zeus[Bro91] makes extensive use of objects, particularly at higher, more abstract levels, such as classes for algorithms, windows, and views. Animation designers have the power to inherit properties and methods from base classes or to refine for their own particular requirements.

6 Conclusion

Application-specific views illustrate programs in ways that help programmers rapidly assess the programs' correctness. The myriad of possibilities for program visualizations requires a general-purpose graphical support methodology rather than a library of predefined views or an ad-hoc visualization technique. Achieving the tenuous balance of ease-of-use along with power and animation capability is a challenge. We have created an animation methodology, POLKA, to address this challenge and support the display of parallel programs' concurrent operations. POLKA retains the simple, yet effective, notion of modifying graphical objects along paths via the path-transition animation paradigm. POLKA adds, however, the key capability of "programming" actions into objects at desired animation times, and then incrementally updating the animation time accordinging to cues from the driving program. We have illustrated how POLKA can be used to build animations of a parallel prefix computation and a simplified particle simulation.

The methodology we have developed sacrifices some ease-of-use for visualization specification power. Improving this trade-off is our primary goal for future work. As one possible improvement, we hope to develop a direct manipulation visualization design tool similar to the DANCE tool[Sta91] for sequential programs animated using the path-transition paradigm. With such a tool, POLKA's Views and animation scenes could be generated by demonstration rather than with graphics programming. We also continue to develop the other components, program monitors and the animation choreographer, of our parallel program visualization system PARADE. Work on these tools, most notably the choreographer, will drive future capabilities of POLKA.

7 Acknowledgments

Bill Appelbe, Jim Foley, Eileen Kraemer, Noi Sukaviriya, Joe Wehrli, and Lorna Zorman read early versions of this paper and provided helpful feedback for its improvement. Joe Wehrli's aid in implementing the 3D version of the system was invaluable.

References

[AS91] William Appelbe and John T. Stasko. Utilizing program visualization and animation techniques to aid parallel program development and debugging (extended abstract). In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 207–209, Santa Cruz, CA, May 1991.

[Bar86] Paul S. Barth. An object-oriented approach to graphical interfaces. *ACM Transactions on Graphics*, 5(2):142–172, April 1986.

[Bro88] 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.

[Bro91] Marc H. Brown. ZEUS: A system for algorithm animation and multi-view editing. In *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pages 4–9, Kobe Japan, October 1991.

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

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

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

[KK73] K. Kolence and P. Kiviat. Software unit profiles and Kiviat figures. *Performance Evaluation Review*, pages 2–12, September 1973.

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

[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.

[Mey92] Scott Meyers. *Effective C++: 50 Ways To Improve Your Programs and Designs.* Addison-Wesley, Reading, MA, 1992.

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

[PSB92] Blaine A. Price, Ian S. Small, and Ronald M. Baecker. A taxonomy of software visualization. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, volume II, pages 597–606, Kauai, HI, January 1992.

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

[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).

[SG92] Sekhar R. Sarukkai and Dennis Gannon. Performance visualization of parallel programs using SIEVE.1. In *Proceedings of the 1992 International Conference on Supercomputing*, pages 157–166, Washington, D.C., July 1992.

[Sno88] Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.

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

[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. 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.

[VL88] John M. Vlissides and Mark A. Linton. Applying object-oriented design to structured graphics. In *Proceedings of the 1988 USENIX C++ Conference*, pages 81–94, Denver, CO, October 1988.

[Z+91] Robert C. Zeleznik et al. An object-oriented framework for the integration of interactive animation techniques. *Computer Graphics: SIGGRAPH '91*, 25(4):105–111, July 1991.