Visualizing Program Executions on Large Data Sets Using Semantic Zooming

Jeyakumar Muthukumarasamy
John T. Stasko

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

E-mail: {*jk*|*stasko*}*@cc.gatech.edu*

Technical Report GIT-GVU-95-02

Abstract

Understanding and interpreting a large data source is an important but challenging operation in many technical disciplines. Computer visualization has become a valuable tool to help capture and portray characteristics of large data sets. In software visualization, illustrating the operation of very large programs or programs working on very large data sets has remained one of the key open problems. Here, we introduce an approach that uses *semantic zooming* to depict large program executions. Our method utilizes abstract, clustered graphics to portray program operations on the entire data set. Then, by interacting with the presentation, a viewer can zoom in to examine details and individual values. At this "magnified" level, the presentation adjusts to reflect displays common in existing algorithm animation and program visualization systems.

1 Introduction

Software Visualization is the use of visualization and animation techniques to help people understand the characteristics and executions of computer programs[SP92, PBS93]. By facilitating program understanding, one can assist students who are learning new programming methodologies, and software developers who are testing, debugging, maintaining, and optimizing their code. Recent advances in software visualization systems coupled with wider availability of computer graphics hardware have helped to broaden the application of software visualization and to illustrate how software visualization can contribute to program comprehension.

Software visualization systems can present a wide range of attributes about programs including views of data structures[Mye83, SI91], system functionality[Rei85, KRR94], algorithmic operations[Bro88, Sta90, Bro91], performance measures[HE91, SG93, RWJ93, HM94], and even software engineering metrics[ESSJ92, DPHKV93]. Although much of the focus in software visualization has been on serial programs, recent interest in visualizing the operation of concurrent programs has grown[KS93] as well.

One of the key open problems in software visualization, and a valid criticism of the area, is that most software visualizations are of smaller, laboratory-created programs[PBS93]. That is, software visualizations do not scale up well, and they poorly portray large systems or program executions on large data sets. Recent research has begun to address this issue, however.

The Seeplex system by Couch[Cou93] provides scalable execution views for parallel programs, ones that do not change in size or meaning as the number of processors increases. Graphical elements portray categories of processors which are grouped by behavior. Different categories are depicted using color and texture. Users also can create new categories by interactively selecting portions of views.

Leban has developed the notion of a *representative view*[Leb93], one in which both the elements and the behavior of a program are mapped many-to-one to a relatively small number of graphical elements. In other words, graphical objects in a view represent collections of values or operations in a program. A scripting language is provided to describe how data values in a program should be mapped to their visual depictions.

Hackstadt, Malony, and Mohr have presented visualization techniques for illustrating large data-parallel program executions[HMM94]. They list four techniques to help create scalable visualizations:

- adaptive graphical representations
- reduction and filtering
- spatial arrangements
- generalized scrolling

The program views utilized in this work differ from those used by Couch and Leban in that these use 3-D graphics. In particular, the views were created using the IBM Data Explorer scientific visualization environment. Techniques like this allow software visualizers to leverage the large body of work already existing in the scientific visualization community[Fre88, DBM89, R+94].

Also taking a 3-D approach, Stasko and Wehrli have developed views that can portray large data sets manipulated by large parallel programs[WS93, SW93].

All the work mentioned above was developed in the context of visualizing *concurrent* programs. This is not unexpected, since one of the primary advantages of a parallel computer is to facilitate computation on extremely large data sets. Research on visualizing large serial programs also exists, however.

The SeeSoft system[ESSJ92] provides a columnar representation of source files in which one line of pixels on a display corresponds to one line of source code. In this way, thousands of lines of code can be presented succinctly on one display. By using different color mappings, the system can present a number of software engineering metrics about the code: who last modified a line, when was it modified, has there been a bug on the line, how many times was the line encountered during an execution, etc.

Strata-Various uses tabular and chart views to portray many different levels of a program execution including hardware, operating system, user libraries and the program itself[KRR94]. The system provides a variety of views and coordinated navigation techniques to facilitate viewer inspection of what a program did during a particular execution.

Our work builds on all this prior research. Our particular focus herein is on 2-D, color graphical displays. This makes it potentially useful to a large number of software developers. Like Couch[Cou93], we choose not to fallback on the use of scrolling and scrollbars to address scale. Like Leban[Leb93] and Hackstadt, *et al*[HMM94] in particular, and characteristic of all work in this area, we use graphical images to represent abstractions or clusters of values from a program. In particular, we utilize the notion of *semantic zooming*, a visualization and interaction technique with three key components:

1. Particular graphical objects in views can represent individual values or they can represent cumulative, clustered data about a program.

2. A visualization begins by showing the global computation (with all data captured in some manner) then the viewer can focus or zoom in on particular details.

3. Viewers navigate between abstracted and detailed views by interacting with the graphical objects on the display.

We seek to provide generalizable views and techniques that are suitable to many different types of programs. Given any particular program, it is not too difficult to create abstract, global views that capture its essence. But doing this on a program-by-program basis is simply too application-specific and not suitable as a debugging aid. We seek general techniques that can easily be adapted to many different programs.

In the remaining sections of the article, we elaborate on the concept of semantic zooming and we provide example visualizations that use the technique. We also describe some of the graphics and system support utilized to create these views.

2 Semantic Zooming

All of the visualizations of program executions on large data sets that we have created utilize the concept of *semantic zooming*. In brief, semantic zooming allows the viewer to zoom

in/out or focus on a particular portion of the program or data set. But rather than using straightforward magnification and demagnification to zoom in and out, the presentation style of the view adjusts at the different zoom levels. When looking at a view of the entire data set, the viewer may see graphical objects that portray cumulative summaries of large chunks of the data and the program execution. When looking at a very detailed view of a small portion of the program data, the viewer may see individual values or elements. Most importantly, these two views may not look anything alike, but they should be built to be as informative and to convey as much information as possible at that level.

A variety of different systems have utilized the general notion of semantic zooming recently. The AGE Graphical Environment[SKA94] uses semantic zooming to show more detail of objects in pen-based computing environments. The Galaxy of News system[Ren94] helps support navigation through a large news article information space by providing semantic zooming operations to the viewer. Pad[PF93] and Pad++[BH94] provide general information visualization environments where semantic zooming is the primary presentation and navigation metaphor. When the user selects an object to examine, the view zooms in on that object. The zooming typically begins as a normal magnification, but the rendering of the object often changes as it grows larger and larger.

In the context of software visualization, we define semantic zooming as having the following principles:

- All visualizations begin showing a view of the entire data set of the program, usually at an abstract level due to the data size.
- At some level, all of the program data should be visible without falling back on the use of scrolling and panning. That is, the presentation of all program data should fit within one window.
- Viewers interact with a view and zoom in on a portion of the program data by interactively selecting a graphical object representing that portion of the data.
- Different zoom levels or perspectives on the program data can either be shown in the same window or in separate windows.
- At the lowest, most detailed view level, the visualizations should use recognized algorithm animation or program visualization presentation styles.
- All views update concurrently and always portray the current state of the program execution.

In the next section we provide examples of program visualizations that utilize these principles of semantic zooming.

3 Animation Examples

This section discusses two examples, sorting and graph programs, and shows how we use semantic zooming to present large data sets in these domains.

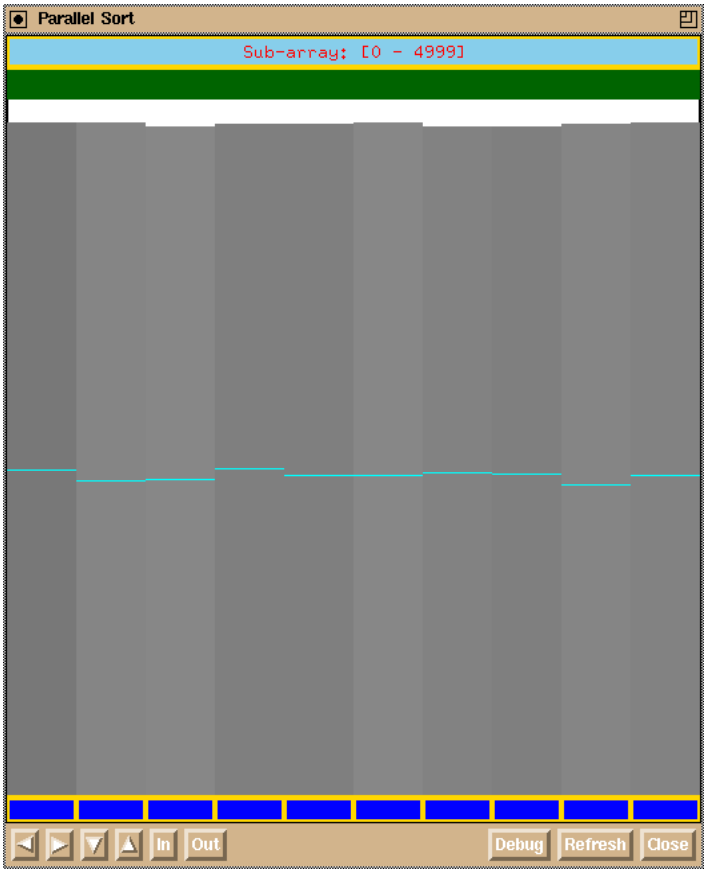

Figure 1: Our abstract blocks oriented view caught near the beginning of a sorting program.

3.1 Sorting Programs

Literally hundreds of animations of sorting programs exist, with perhaps the best known being the seminal video *Sorting Out Sorting*[BS81] by Ron Baecker. Virtually all of these existing animations, however, are of programs running on at most a few hundred data elements. What would a program animation of a sort of 5,000 or even 500,000 values look like?

We have developed such a visualization—it uses semantic zooming to portray all the data in one compact window without requiring any scrolling. The particular sorting algorithm shown here is quicksort, but the visualization itself can be applied to any sorting algorithm that uses comparison and exchange. The visualization begins by capturing all of the program's data in a block-oriented view as shown in Figure 1.

In this view, the data values are divided into i blocks (10 blocks in this particular case) based on their proximity to each other. That is, if there are n elements in a list to be sorted, the first n/i elements will comprise block 1, the second n/i block 2, and so on. Each block is displayed as a rectangle; The rectangles are organized in a horizontal row. The minimum value within a block determines the bottom y-coordinate of its corresponding rectangle, and the maximum value within a block determines the top y-coordinate of the rectangle. These y-coordinates can range from the bottom of the view (corresponding to the absolute smallest value in the entire data set) to the top of the view (largest data value). The average of all

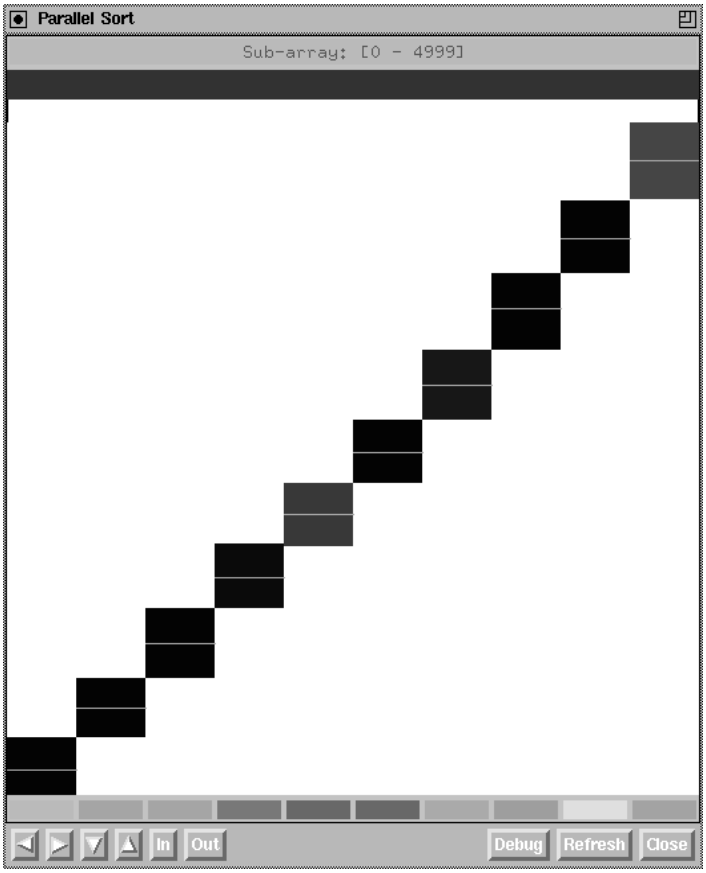

Figure 2: This view shows the blocks view at the end of the sorting algorithm when the list is completely sorted. Note that the 10 blocks are mutually exclusive and totally black.

the elements in a block is computed and represented as a bright horizontal line dividing the rectangle. Following an idea of Leban[Leb93], the shade or fill pattern of a block indicates how "sorted" the block is. The shade of a completely unsorted (data values in reverse sorted order) block is shown as white and the shade of a totally sorted block is shown as black. As a data set moves from being totally unsorted to sorted, we use increasingly darker levels of gray fill to represent just how sorted the set is.

This blocks view provides a good overview of the sorting process. When the program begins and its data is in a scattered random permutation, the blocks are all tall (they each contain at least one very large and very small value on average), they are a moderate gray color or fill (about halfway sorted on average), and they all start with a similar average value around the median of all data values. Such a configuration is shown in Figure 1. Conversely, at the end of the program when the elements are sorted, the blocks are all black and do not overlap (Figure 2).

At any point in time in the middle of the program execution, the blocks view gives a good indication of the composition of the set of elements being sorted. For example, in Figure 3, the average value of block 1 is near the bottom of its corresponding rectangle, indicating that this section consists mostly of elements with small values. The block is still quite tall, however, indicating that at least one very large data value is still stored in this section. Most importantly, however, the blocks view can be used to represent any amount

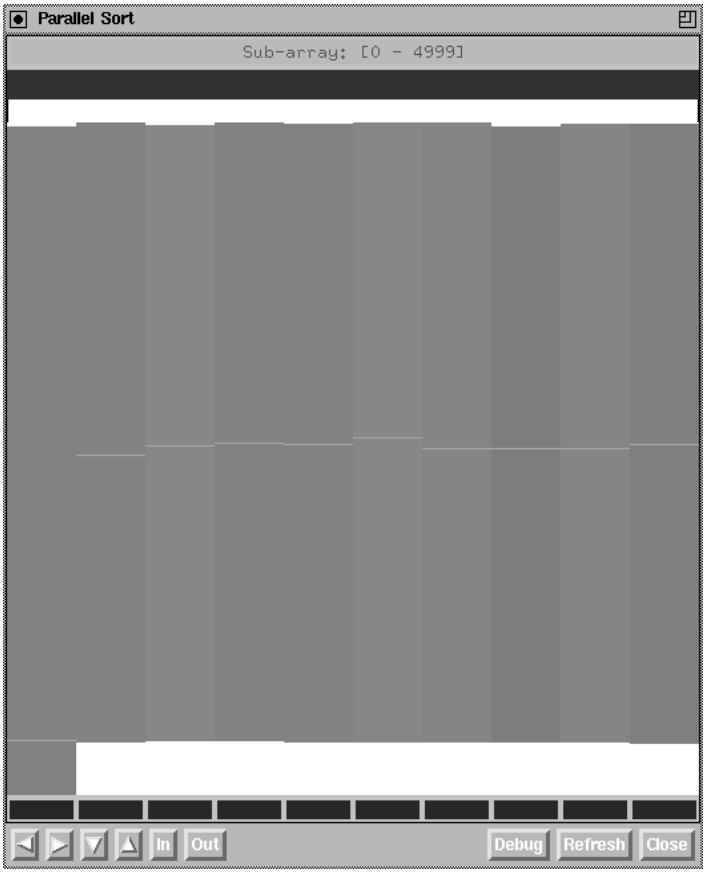

Figure 3: This blocks view is from part way through the sorting program. The first block of elements on average has small values but it still has at leat one very large value.

of data without loss of any of the above information.

The particular sorting program shown in our figures is actually a parallel program running on a KSR-1 machine. We utilize the color bar at the bottom of the blocks view (row of short rectangles) to additionally indicate the processor that last accessed an element in the block.

The minimum, maximum, average, and sortedness values for each block typically are not directly available from the sorting program being visualized, so they must be computed by the animation at run-time. Also, since the blocks are divided based on proximity (array position), the elements which define a block are continuously changing as values are swapped during the program's exchange operations. As a result, the attributes of a block are continuously changing. Ideally, we would like to compute the changes to the attributes of a block incrementally in constant time after each exchange in the sorting program.

Changes to the average value of a block can be done in an incremental fashion and are computed easily in constant time with respect to each exchange. We simply maintain the sum of all values in a block and recompute this total after an exchange. The average is easy to calculate from there.

Computing changes to the minimum, maximum, and "sortedness" values of a block are more challenging, however. Whenever a new data value is swapped into or out of a

block, the minimum or maximum value might change for the block, and we may need to recompute the new minimum or maximum value from scratch. (This assumes we do not keep a sorted copy of all the elements in each block. Clearly, this would be too expensive. We do maintain the minimum and maximum values in each block, however, so we do know when a new minimum or maximum occurs.) If a new minimum or maximum is swapped into a block, our work is easy. If a minimum or maximum value is swapped out of a block, recomputing a new one has a complexity of $O(m)$, where m is the number of elements in each partition ($m = n/10$ in our case). In general, on any particular exchange operation, the probability that a minimum or maximum value is swapped out of a block is fairly small, so this $O(m)$ cost does not occur too often.

Computing the sortedness of a block is also tricky, especially without a formal definition of sortedness. In our implementation, we use sortedness to indicate the relative number of values in order at any given instance. That is, we compute the sortedness of a block as the sum of the lengths of all continuously increasing sub-sequences with length of at least 2. For example, the sortedness for the sequence $1, 2, 9, 5, 4, 6$ will be 5 since it has two continuously increasing sub-sequences $1, 2, 9$, and $4, 6$, of lengths 3 and 2, respectively. The sortedness of a completely sorted sequence of numbers will be equal to the length of the sequence, and the sortedness of a completely backwards sequence of numbers will be 0. This method allows us to calculate the sortedness of a series of numbers in time linear to the length of the series at start-up. Once this original linear calculation is performed, however, we can recalculate the sortedness of a block after an exchange in constant time. We simply examine the local effect of a new value by examining both the old and new values with respect to their left and right neighboring values.

In summary, recalculating the average and sortedness value of a block requires constant time after an exchange operation. Recalculating the minimum or maximum also takes constant time, except when a minimum or maximum value is swapped out of a block. In that case, time linear to the size of the block is required. In general, for this particular sorting animation, we do not update the display after each exchange. That would be too costly (slow) on the extremely large data sets it uses. Rather, we only update the display after some number of exchanges occur. (This value can be set through a parameter, we often use 10.) In that case, we also cache minimum and maximum exchanges out of a block, and we only perform the linear recalculation once, if necessary, when the display is updated.

The blocks view as described so far provides a good overview of all the data, but what if a viewer needs to examine individual data values too? We overcome this problem using semantic zooming. In essence, each block representing a section of the program's data is an active object which can itself become the "active focus." By simply selecting a block with the pointing device, the viewer makes it the focus and it expands to become the entire view. The chosen block is sub-divided into i blocks just as was done initially. Now, however, a smaller portion of the entire data set is shown, only the section of interest to the viewer. The same abstractions for minimum, maximum, average, and sortedness are used just as before. Figure 4 shows a "zoom" onto the first block of Figure 3. This is the first 10% of the entire array. As we can see, the first 90% of this segment has small values, but some very large value still exists near the end of this segment.

This focusing operation can be carried out recursively and the view maintains the abstraction of blocks as long as the number of elements in a block is too large to be displayed as individual elements. Once the viewer has focused enough to be able to portray all the

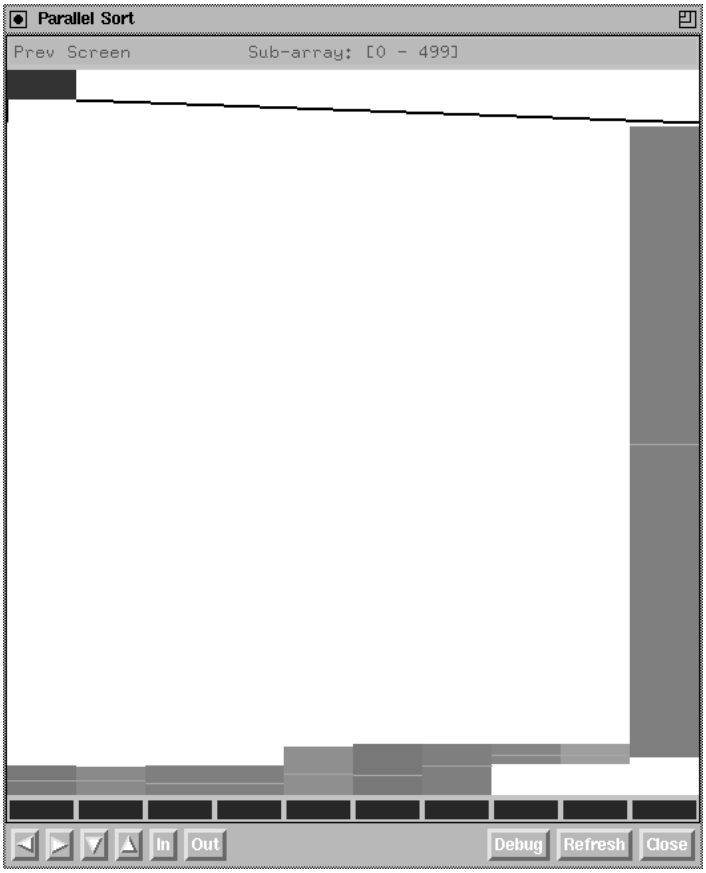

Figure 4: This view shows the contents of the first 10% of the entire area. The viewer has chosen to look at this segment by selecting with the pointer the first block in the global view. The slanted line(s) and darkened rectangle near the top help indicate to the viewer what segment of the entire array is being examined.

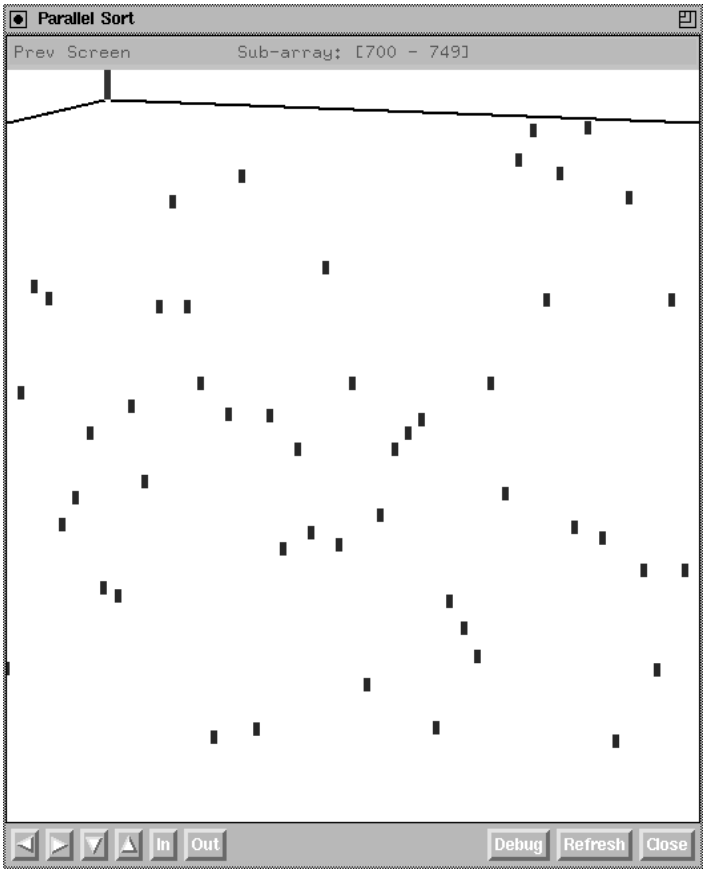

Figure 5: The view transforms its presentation when a small enough set of elements are to be shown in the display. Here, a traditional algorithm animation scatter plot sorting presentation is used where the x-dimension corresponds to position in the array, and y-dimension encodes the relative value of an element (smallest at bottom and biggest at top).

data values of a block adequately within the view, the display format of the view changes to better represent the individual items. For example, Figure 5 shows the same program executions portrayed in the prior figures. Here, however, we have zoomed in far enough so that the data values shown can be depicted in the classic scatter-plot dots algorithm animation viewing technique[BS81]. Alternatively, the individual data elements could be shown in any other traditional algorithm animation or program visualization style, such as height-scaled bars[BS85]. At this level, detail about particular elements now is available. To move back out a level (focus on a larger portion of the set again) the viewer simply selects the "Previous Screen" option at the top of the view or clicks the right mouse button anywhere in the view.

3.2 Graph Programs

Another domain of algorithms that frequently operates on very large data sets is the domain of graph algorithms.[1] As with sorting, building scalable visualizations of graph programs is

[1]Here, we assume that the graphs utilized in programs are planar graphs with x, y coordinates on the vertices.

challenging because viewers may need to see both an overview and the details. One possible solution is to use fisheye view techniques[SB94]. This approach alleviates the problem to some extent, but it will break down on a graph with a very large number of vertices, too many to properly fit within a display window. In our implementation, we solve this problem by using semantic zooming and two views of the graph data, one which shows an overview of the entire graph, and the other which shows details. The two views are a blocks grid view and a zoomed vertex-edge view, respectively.

The blocks view is an abstract view. Abstraction is achieved by proximity similar to the sorting visualization. The difference is that proximity in this case is two dimensional. The graph is assumed to consist of blocks in a grid as shown in Figure 6. All blocks have equal width and height. This grid corresponds to the 2-D geometry of the graph from a "zoomed out" perspective. That is, consider the window's view to be pulled back enough so that all the vertices and edges of the graph fit inside. Then, all the nodes and edges that fall within a block comprise that block.

But rather than trying depict vertices and edges, we use summary blocks to portray information about all the vertices and edges in that region of the graph. In particular, a variety of visual encoding techniques communicate information about the different regions of the graph. For example, the shade of a block can indicate something about the nodes and/or edges which comprise the block. In a depth first search algorithm, the shade of the block might be used to indicate the number of nodes that have been visited, and in a minimum spanning tree algorithm, the shade might be used to indicate the number of edges in the block that belong to the tree. Similarly, the blocks can be colored to indicate some value, or small rectangles in a corner of each block (not shown in our example) can be used to represent some other attribute of the algorithm. The borders of a block also can be highlighted to indicate an attribute of the program execution, such as the region of the graph containing the vertex that the algorithm is processing at a given time. The usefulness of the visualization depends on the set of attributes being mapped and the mapping itself.

Like the sorting visualization, this graph visualization provides a semantically zoomed view (Figure 7). It shows the vertices and edges of a small region of the graph in complete detail. At this level, we can see the specifics of the algorithm's operation on the individual nodes and edges that may change color, flash, and change fill, as often seen in traditional algorithm animations. By picking on a spot in the blocks view, the viewer chooses a region to focus on and show in this zoomed view. Like the sorting visualization, when the data being presented is small enough to fit in a window, the view adjusts to show this detailed view. We also have found it useful to introduce a mode where the detailed view is always present and it automatically follows the focus of the algorithm.

4 Support Environment

The visualizations shown in this article were built using the Polka animation system[SK93]. Polka provides an object-oriented framework for building animations, including individual windows or *Views* in which animations occur. Within a View, graphical objects (called *AnimObjects*) such as lines, rectangles, circles, text, etc., can change position, size, color, and so on, thus presenting an animation. These changes are called *Actions* in Polka. Because Actions on AnimObjects are scheduled independently of other Actions and AnimObjects, it is easy to synchronize animation activities and build concurrency into a View or Views.

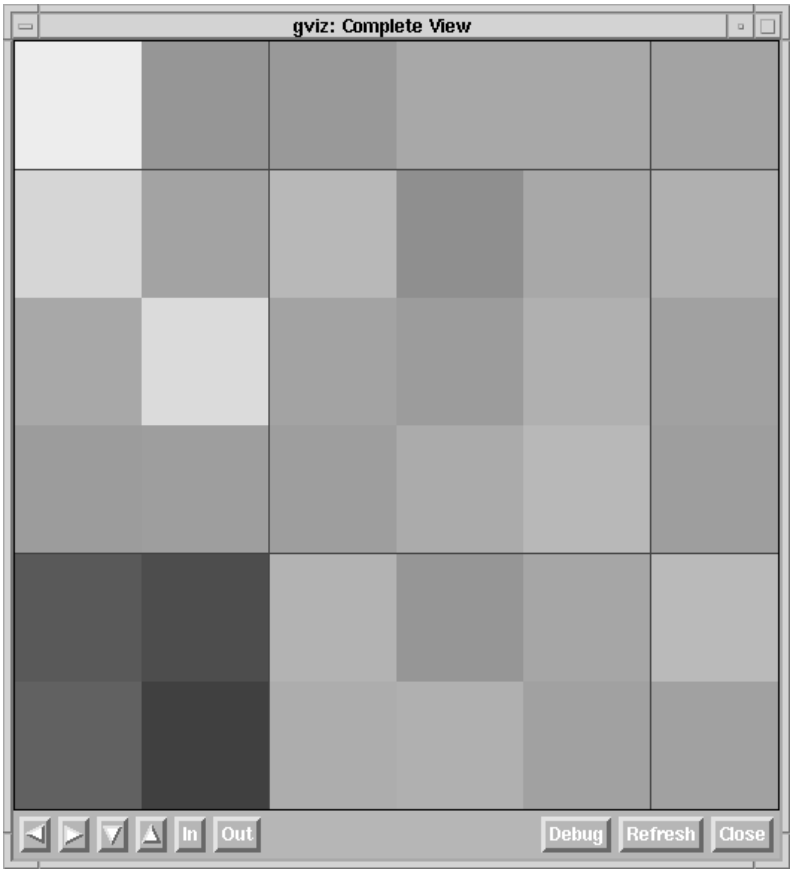

Figure 6: The graph overview visualization. Each block summarizes some cumulative attribute or attributes of all the vertices and/or edges lying within that region of the graph.

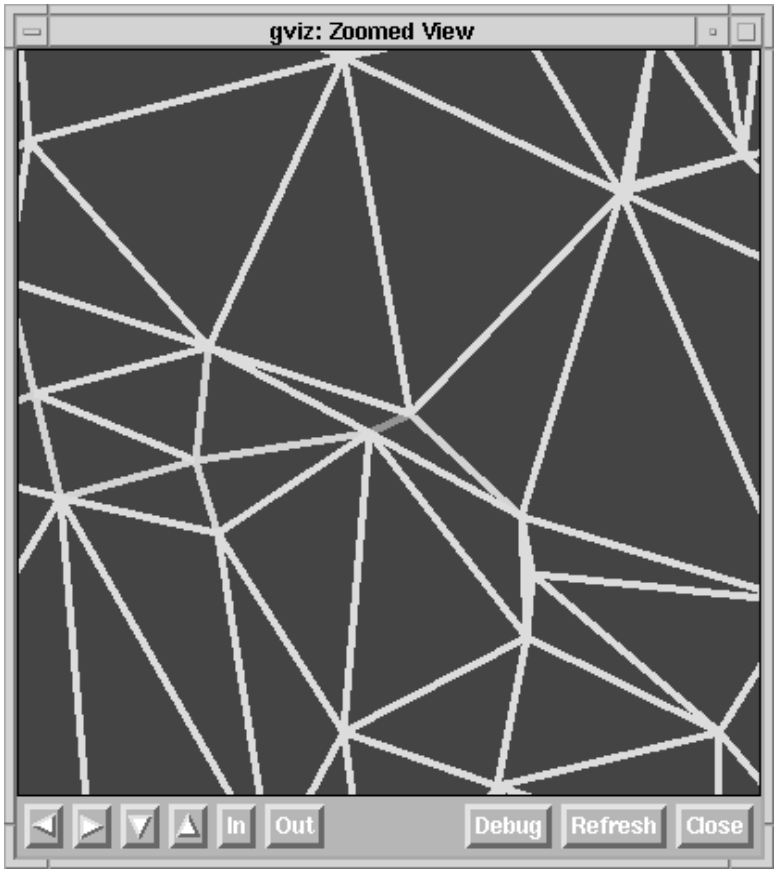

Figure 7: The more traditional zoomed view of a graph visualization that depicts individual vertices and edges of interest.

For animations such as the graph example shown earlier in which the different semantic zoom levels are shown in different physical Views (windows) on the display, the basic Polka framework was sufficient. But in the animations where the different levels are always shown in the same window (earlier sorting visualization), we needed to extend Polka. We added the concept of a *Portal* which corresponds to a window on the display. Portals then support the notion of presenting different independent Views within them. (The concept of one particular View being tied to a window was broken). This was necessary because semantic zooming frequently generates a completely new and different set of graphical objects from zoom one level to the next. Each zoom level now corresponds to a View.

More specifically, a Portal can have any number of animation Views associated with it. Only one of those Views, however, can be the *Active View*, that is, the one currently visible in the Portal window. All other Views are considered to be hidden offscreen. Offscreen Views can have animation Actions scheduled and executed, however, so they can be kept up-to-date. Then, if the viewer returns to a level via a sequence of zoom operations, that View will reflect the current proper state of the computation.

Typically, in the animations we have built, we use a lazy View instantiation process. That is, we only create a new zoomed View when the viewer asks for it. Once the View is created, however, we do keep the View up-to-date whether it is the active View or not. If a View is created and becomes active, then goes inactive or offscreen and subsequently returns active again, we simply re-install the View rather than instantiating all of its graphical objects again.

In order to allow viewers of the animations to select or pick AnimObjects as the impetus of a zoom, Polka provides the concept of AnimObject callback routines. All graphical objects in a View can have an associated callback function that is invoked whenever an animation viewer selects that object with the mouse. Inside the callback routine, we typically determine which AnimObject was selected, how much of the data will be presented at the next zoom level, and whether we need to instantiate a new View or re-install an existing one. Zooming back out, which is usually done by selecting the special zoom-out box, is easier because we know that the View to be shown must already exist.

5 Conclusion

We have presented an approach for visualizing program executions on very large data sets. This approach utilizes semantic zooming and allows a viewer to adjust the level of detail presented in the display. If an overview of all the data is desired, values are clustered into a few graphical objects that portray large sections of the data. In particular, we introduce a blocks view technique where graphical attributes of the blocks (geometry, fill, color) map to attributes of the data. To focus on more details, a viewer can interactively select these clustered graphical objects. This changes the focus of the display to be the desired smaller portion of data. Once a view is zoomed enough so that all the elements of a focus region can adequately fit within the display window, the data presentation adjusts to reflect a traditional program visualization or algorithm animation presentation style. Our approach uses descriptive graphical encodings and navigation flexibility to facilitate a *focus and context* software visualization strategy.

6 Acknowledgments

Support for this project has come from the National Science Foundation under grant CCR-9121607.

References

[BH94] Benjamin B. Bederson and James D. Hollan. Pad++: A zooming graphical interface for exploring alternate interface physics. In *Proceedings of the 1994 ACM Symposium on User Interface Software and Technology*, pages 17–26, Marina del Rey, CA, November 1994.

[Bro88] Marc H. Brown. Exploring algorithms using Balsa-II. *Computer*, 21(5):14–36, 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.

[BS81] Ronald M. Baecker and David Sherman. Sorting Out Sorting. 16mm color sound film, 1981. Shown at SIGGRAPH '81, Dallas TX.

[BS85] Marc H. Brown and Robert Sedgewick. Techniques for algorithm animation. *IEEE Software*, 2(1):28–39, January 1985.

[Cou93] Alva L. Couch. Categories and context in scalable execution visualization. *Journal of Parallel and Distributed Computing*, 18(2):195–204, June 1993.

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

[DPHKV93] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of the ACM OOPSLA '93 Conference*, pages 326–337, Washington, D.C., October 1993.

[ESSJ92] Stephen G. Eick, L. Steffen, Joseph, and Eric E. Sumner Jr. Seesoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.

[Fre88] Karen Frenkel. The art and science of visualizing data. *Communications of the ACM*, 21(2):110–121, February 1988.

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

[HM94] Steven T. Hackstadt and Allen D. Malony. Next generation parallel performance visualization: A prototyping environment for visualization development. In *Proceedings of the Parallel Architectures and Languages Europe (PARLE)*, pages ???–???, July 1994.

[HMM94] Steven T. Hackstadt, Allen D. Malony, and Bernd Mohr. Scalabale performance visualization for data-parallel programs. In *Proceedings of the Scalable High Performance Computing Conference (SHPCC '94)*, pages 342–349, Knoxville, TN, May 1994.

[KRR94] Doug Kimelman, Bryan Rosenburg, and Tova Roth. Strata-Various: Multilayer visualization of dynamics in software system behavior. In *Proceedings of the IEEE Visualization '94 Conference*, pages 172–178, Washington, D.C., October 1994.

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

[Leb93] Bruce Leban. *Representative Views of Parallel Computation.* Unpublished Ph.D. thesis proposal, M.I.T., 1993.

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

[PBS93] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, September 1993.

[PF93] Ken Perlin and David Fox. Pad: An alternative approach to the computer interface. In *Proceedings of the 1993 ACM SIGGRAPH Conference*, pages 57–64, Anaheim, CA, August 1993.

[R+94] Lawrence Rosenblum et al., editors. *Scientific Visualization: Advances and Challenges.* IEEE Computer Society Press, Los Alomitos, CA, 1994.

[Rei85] Steve P. Reiss. Pecan: Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, SE-11(3):276–285, March 1985.

[Ren94] Earl Rennsion. Galaxy of news: An approach to visualizing and understanding expansive news landscapes. In *Proceedings of the 1994 ACM Symposium on User Interface Software and Technology*, pages 3–12, Marina del Rey, CA, November 1994.

[RWJ93] Diane T. Rover and Charles T. Wright Jr. Visualizing the performance of SPMD and data-parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):129–146, June 1993.

[SB94] Manojit Sarkar and Marc H. Brown. Graphical fisheye views. *Communications of the ACM*, 37(12):73–84, December 1994.

[SG93] Sekhar R. Sarukkai and Dennis Gannon. SIEVE: A performance debugging environment for parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):147–168, June 1993.

[SI91] Takao Shimomura and Sadahiro Isoda. Linked-list visualization for debugging. *IEEE Software*, 8(3):44–51, May 1991.

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

[SKA94] Samudra Sengupta, Takayuki Dan Kimura, and Ajay Apte. An artist's studio: A metaphor for modularity and abstraction in a graphical diagramming environment. In *Proceedings of the 1994 IEEE Symposium on Visual Languages*, pages 128–136, St. Louis, October 1994.

[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.

[Sta90] John T. Stasko. TANGO: A framework and system for algorithm animation. *Computer*, 23(9):27–39, September 1990.

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

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