PROTOTYPING ADVANCED PARALLEL PROGRAM AND PERFORMANCE VISUALIZATIONS

by

STEVEN THOMAS HACKSTADT

A THESIS

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

June 1994

"Prototyping Advanced Parallel Program and Performance Visualizations," a thesis prepared by Steven T. Hackstadt in partial fulfillment of the requirement for the Master of Science degree in the Department of Computer and Information Science. This thesis has been approved and accepted by:

Allen D. Malony, Thesis Advisor

Date

Accepted by:

Vice Provost and Dean of the Graduate School

An Abstract of the Thesis of

Steven Thomas Hackstadt for the degree of Master of Science

in the Department of Computer and Information Science to be taken June 1994

Title: PROTOTYPING ADVANCED PARALLEL PROGRAM AND PERFORMANCE VISUALIZATIONS

Approved: ______________________________
Dr. Allen D. Malony

A new visualization design process for the development of parallel program and performance visualizations using existing scientific data visualization software can drastically reduce the graphics and data manipulation programming overheads currently experienced by visualization developers. Data visualization tools are designed to handle large quantities of multi-dimensional data and create complex, three-dimensional, customizable displays which incorporate advanced rendering techniques, animation, and display interaction. These capabilities can be used to improve performance visualization, but to be effective, they must be applied as part of a formal methodology relating performance data to visual representations. Under such a formalism, it is possible to describe performance visualizations as mappings from performance data objects to view objects, independent of any graphical programming. Through three case studies, this work examines how an existing scientific visualization tool, IBM's Data Explorer, provides a robust environment for prototyping next-generation parallel performance visualizations.

CURRICULUM VITA

NAME OF AUTHOR: Steven Thomas Hackstadt

PLACE OF BIRTH: Tacoma, Washington

DATE OF BIRTH: December 14, 1969

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
University of Puget Sound

DEGREES AWARDED:

Master of Science in Computer Science, 1994, University of Oregon
Bachelor of Science *summa cum laude* in Mathematics and Computer Science, 1992, University of Puget Sound

AREAS OF SPECIAL INTEREST:

Parallel Program and Performance Visualization

PROFESSIONAL EXPERIENCE:

Research Assistant, Department of Computer and Information Science, University of Oregon, Eugene, 1993-1994

Teaching Assistant, Department of Computer and Information Science, University of Oregon, Eugene, 1992-1993

AWARDS AND HONORS:

Phi Beta Kappa National Honor Society, 1992-1994
Phi Kappa Phi National Honor Society, 1991-1994
Outstanding work in Math/Computer Science, University of Puget Sound, 1992
Goman Award, Math and Computer Science Department, University of Puget Sound, 1991-1992

GRANTS:

NASA Graduate Student Researchers Program Fellowship, 1994-1995

PUBLICATIONS:

S. Hackstadt and A. Malony. *Data Distribution Visualization (DDV) for Performance Visualization*. University of Oregon, Department of Computer and Information Science, Technical Report CIS-TR-93-21, October 1993a.

S. Hackstadt and A. Malony. *Next-Generation Parallel Performance Visualization: A Prototyping Environment for Visualization Development*. To appear Proc. Parallel Architectures and Languages Europe (PARLE) Conference, Athens, Greece, July, 1994. Also, University of Oregon, Department of Computer and Information Science, Technical Report CIS-TR-93-23, October 1993b.

S. Hackstadt, A. Malony, and B. Mohr. *Scalable Performance Visualization for Data-Parallel Programs*. Proc. Scalable High Performance Computing Conference (SHPCC), Knoxville, TN, May, 1994. Also, University of Oregon, Department of Computer and Information Science, Technical Report CIS-TR-94-09, March 1994.

ACKNOWLEDGEMENTS

The author wishes to express his sincere thanks and appreciation to Dr. Allen Malony for his attention, guidance, insight, and support during this research and the preparation of this thesis. In addition, special thanks are due to Dr. Bernd Mohr for constructive comments and suggestions during the development of many of the visualizations, and for providing valuable trace data to create many of the displays.

This research was supported in part by the National Science Foundation under grant NSF ASC92-13500, by ARPA under Rome contract AF-30602-92-C-0135, by a grant from OACIS, and by IBM Research and Development contract MHVU3704, to Dr. Allen Malony at the University of Oregon.

DEDICATION

To Mom and Dad - for encouragement and trust.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

II. MOTIVATION 7

III. RELATED WORK 11

IV. DATA VISUALIZATION SOFTWARE 15

V. METHODOLOGY 17

VI. IMPLEMENTATION 21

Trace Transformation: Creating the Data Object File 24
Graphical Transformation: Creating the Visual Program 26
Performance Visualization Examples 33

VII. CASE STUDY: ADVANCED PERFORMANCE VISUALIZATIONS 38

Introduction 38
Extending Existing Visualizations 39
New Visualizations 43
Summary 48

VIII. CASE STUDY: DATA DISTRIBUTION VISUALIZATIONS 49

Introduction 49
Goals for a DDV Environment 50
High Performance Fortran 51
Establishing the DDV Problem Domain 55
An Experimental DDV Environment 58
Application: Gaussian Elimination 62
Observations 68
Summary 70

IX. CASE STUDY: SCALABLE PERFORMANCE VISUALIZATIONS 72

Introduction 72
Scalable Visualization Methods 73

Page

Applications 76
Summary 87

X. EVALUATION 88

XI. CONCLUSION 91

BIBLIOGRAPHY 93

LIST OF FIGURES

Figure Page

1. A High-Level, Abstract View of Performance Visualization Treats a Visualization as a Mapping from Performance Data Objects to Graphical View Objects and Promotes the Development of Interfaces to Existing Graphical Resources. 19

2. A Development Process Based on the Abstract Performance Visualization Methodology Can Be Realized Using Existing Data Visualization Software. 22

3. A Simple Visual Program in Data Explorer Is Capable of Creating Many Different Types of Visualizations, as Seen in Figure 8. 27

4. The Import Module Reads a Data File into the Visualization Environment. . . . 28

5. Control Panels Are Used To Create Simple Interfaces That Can Manipulate Many Characteristics of a Visualization. 29

6. A Colormap Editor Can Create Arbitrary Colormaps for a Visualization, Enabling the Analyst To Explore and Highlight Different Features of the Represented Data. 31

7. Displays from Existing Performance Visualization Tools Can Be Prototyped, and Subsequently Extended, Using Three-Dimensional Data Visualization Packages. 33

8. The Visual Program in Figure 3 Can Create a Wide Range of Performance Visualizations Depending on the Structure of the Underlying Data. 35

9. ParaGraph Uses a Kiviat Diagram Visualization To Show Processor Utilization. 39

10. The Traditional Two-Dimensional Kiviat Diagram Is Easily Implemented Using Data Visualization Software. 41

11. The Two-Dimensional Kiviat Diagram Can Be Extended to Three Dimensions by Allowing Time To Travel along a Third Axis. 41

Page

12. A Three-Dimensional Kiviat Tube Reveals Global Trends in the Performance Data. 42

13. By Combining the Two-Dimensional and Three-Dimensional Kiviat Displays, a Potentially More Useful Visualization Results. 44

14. A Three-Dimensional Processor Performance Metric Determines the Location of Processors within the "Performance Space." 46

15. Vertical Displacement and Coloring Reveal Remote and Local Data Access Patterns to a Distributed Data Structure. 48

16. The Primary Compiler Directives in HPF Allow Users To Create Virtual Processors, Align Data Structures, and Distribute Data Across Virtual Processors. 52

17. In HPF, Programmers Specify How Data Structures Are Distributed across a Set of Virtual Processors, but the Compiler Is Responsible for Mapping Virtual Processors to the Physical Processing Units of the Computer. 53

18. On a Distributed-Memory Parallel Computer, an Assignment Statement Can Be Carried Out Using Four Techniques. 54

19. The DDV Problem Consists of Three Main Issues. 56

20. The DDV Environment Incorporates the Implementation Environment of Figure 2 by Adding Capabilities for Collecting and Processing Trace and Performance Data. 59

21. The Cyclic Distribution of a 10x10 Array onto a 2x8 Grid of Processors Leaves Some Processors More Heavily Loaded with Data Elements. 63

22. An Instrumented Code Fragment Shows How the Sequential Algorithm Is Used To Imitate a Possible Parallel Implementation. 63

23. (Time 20) Data Access Patterns Are Already Evident Early in the Algorithm. 65

24. (Time 20) The Processors Do Not Yet Appear Too Unbalanced. 65

25. (Time 60) The Visualization Reflects the Beginning of the Second Phase of the Gaussian Elimination Algorithm. 66

Page

26. (Time 60) The Distribution of Remote Writes Is Considerably Different Than the Overall Distribution of All Memory Accesses. 67

27. (Time 60) A Growing Imbalance in Processor Load Becomes Evident. 67

28. (Time 89) The Algorithm Nears Completion, and the Display Seems To Suggest a Fairly Uniform Access Pattern Over the Life of the Algorithm. 68

29. (Time 89) Processors in the Left Two Columns of the Grid Have Experienced Significantly More Memory Accesses Than the Others. 68

30. A Small Data Structure (8x9) Is Effectively Portrayed by Using Discrete Spherical Glyphs. 77

31. A Medium-Sized Data Structure (16x17) Requires the Vertically-Displaced Tops of the Cylinders Be Connected to the Plane To Provide Reference Information. 77

32. A Continuous Displacement Grid Minimizes Visual Complexity for a Large Data Structure (64x65). 77

33. Isosurfaces Are Used To Portray Remote Data Accesses to 64 Data Elements Arranged in a 4x4x4 Grid At Two Different Times During the Application. 79

34. A Scaled Visualization Shows Local Data Accesses to 4,096 Elements Arranged in a 16x16x16 Grid. 80

35. A Three-Dimensional Scatter Plot Shows Which and How Often Processors Are Accessing the Elements of the Data Structure. 81

36. A Prototype of a Scaled Scatter Plot Exposes Global Data Access Patterns. . . . 82

37. A Kiviat Tube Can Be Used To Portray Data Element Accesses Instead of Processor Utilization. 83

38. A Blown-Up Region of the Kiviat Tube Reveals Three Significant Decreases in Local Data Accesses. 85

39. The Corresponding Kiviat Tube Section Showing Remote Accesses Indicates Similar Decreases. 85

CHAPTER I

INTRODUCTION

Even though we navigate daily through a perceptual world of three spatial dimensions and reason occasionally about higher dimensional arenas with mathematical ease, the world portrayed on our information displays is caught up in the two-dimensionality of the endless flatlands of paper and video screen. All communication between the readers of an image and the makers of an image must now take place on a two-dimensional surface. *Escaping this flatland is the essential task of envisioning information - for all the interesting worlds (physical, biological, imaginary, human) that we seek to understand are inevitably and happily multivariate in nature. Not flatlands.*

- Edward Tufte, *Envisioning Information* (p. 12).

The dramatic increases in the complexity and sophistication of parallel computing systems seen in recent years has outpaced the corresponding growth in the analysis environments that accompany these systems. Consequently, users of parallel computers are left with ineffective resources for analyzing and evaluating the performance of their applications. The many advances that have been made in the areas of performance monitoring and tracing now enable the collection of vast amounts of detailed performance data about applications executing on parallel computers. For the user, however, such massive amounts of data are rarely useful. The analysis problem subsequently becomes extracting the key performance characteristics that will help the user to understand or improve their program.

Visualization has long been recognized as an effective means of portraying large quantities of complex data by replacing a cognitive task (*i.e.*, analyzing a large table of numbers) with a perceptual task (*i.e.*, noticing relationships among graphical objects and characteristics) [48]. While the traditional sciences have benefitted by many advances in visualization techniques and the development of sophisticated data visualization tools and environments, visualizations targeting parallel computer systems have remained ad hoc and have failed to take advantage of more powerful visualization techniques.

Performance visualization is the use of graphical display techniques for the visual analysis of performance data to improve the understanding of complex computer performance phenomena. From the opening chapter of his book *Envisioning Information* [48], Edward Tufte foretells the future of performance visualizations. Tufte recognizes and demonstrates the necessity and effectiveness of multi-dimensional information displays. While the graphics used in current performance visualization tools are predominantly confined to the two-dimensional flatland described by Tufte, the work reported here has developed new methods for rapidly prototyping a new generation of advanced, multi-dimensional performance visualizations.

As parallel programming environments continue to evolve, the need for more sophisticated, more flexible visualizations will increase. To harness the potential of more sophisticated visualization techniques requires a visualization methodology that is more robust than the techniques currently used in many performance visualization tools. Performance visualization has the distinction of lacking a physical model on which displays can be based. Rather, performance data represents abstract relationships between virtual objects (*e.g.*, data elements, processors, or different components of a computing system). This distinction from traditional scientific visualization both hinders and benefits visualization in the context of parallel computing. It is a hindrance in that what a display is attempting to portray may not be immediately obvious to an unfamiliar user. However, the absence of a concrete physical model allows for more flexibility in the design of visualizations. For this reason, performance visualizations are more a product of how performance parameters have been mapped into graphical representations than they are a product of the physical "reality" from which the parameters have been collected.

The tendency for performance visualizations to be more abstract complicates the issues of usability and evaluation. This is best illustrated with a comparison to traditional scientific visualization. Consider, for example, modeling the interactions between thousands of molecules in a three-dimensional cube. Probably the most "useful" (and intuitive) display, in the majority of applications, would be a direct representation of the three-dimensional space and the molecules within it. The molecules might be represented as spheres that "float" around the cube, bouncing off the walls and colliding with one another. Such a visualization is a direct manifestation of the physical system (*i.e.*, the physical system defines the visualization), and few scientists would find some other representation more generally useful than that suggested by the physical system.

Now, suppose someone wished to model the interactions between thousands of processors in a massively parallel computer. Should they use the interconnection topology of the machine (*e.g.*, mesh, hypercube) as the basis for the visualization? Or perhaps they should consider the logical model imposed by a data-parallel programming language (*e.g.*, vector, 2-D or 3-D array)? Or maybe it's more appropriate to base the visualization on how the compiler decides to distribute the data among processors? Each of these represents a possible foundation (*i.e.*, context) upon which visualizations for a parallel system could be constructed, but no one of these is ubiquitous enough to capture all - or even many - of the possible performance visualization scenarios for the seemingly simple task of modeling the processors of a parallel computer. Thus, it is very difficult to determine which display might be the most useful. In fact, the concept of "usefulness" becomes extremely user- and application-dependent in the context of parallel performance visualizations. Furthermore, because performance visualizations can be more abstract, user preference plays a critical role in evaluating the effectiveness of a particular display.

Nonetheless, researchers have identified many important principles relating to the design and use of visualization in the context of parallel program evaluation [4,10,27,29,31,40]. Some of these principles include the use of multiple views, semantic context, and user interaction. *Multiple views* facilitate understanding of a parallel program's operation by capturing and analyzing data from the execution across different levels of observation and from different perspectives. In addition, it is generally believed that visualization interpretation is improved if the user is provided with a *semantic context* for understanding the parallel program data represented. Semantic context can play a role similar to that of the physical models that support traditional scientific visualizations. Finally, *user interaction* allows the viewer to select alternative views, change the level of detail or the type of data, and control display parameters in order to find the best possible visualization scenario for data interpretation.

Although these principles provide constructive guidelines for visualization design, it is still a challenging undertaking to develop generally useful parallel program visualization tools. Several projects have tried to deliver general visualization solutions, leading to widely debated concerns over usability. The focus in the parallel programming tool community on the *quality* of visualizations offered by end-user tools has been, in part, at the expense of research into developing improved visualization *techniques* that better target specific end-user requirements. With the importance of semantic context in enhancing visualization interpretation, it is good practice not to restrict visualization design creativity by requiring visualizations to have broad user appeal, particularly since a meaningful visualization for one user may not be especially meaningful for another. Rather, efforts are better directed at developing visualization techniques that can be applied in building visualization tools to address specific problems that users encounter in parallel program evaluation, while following general principles and guidelines for good visualization design [16,29,48].

To summarize, the preceding arguments lead to several characteristics that should be supported by a visualization environment for parallel systems. First, the absence of a physical model on which to base visualizations suggests the need for a more formal visualization methodology in which displays result from mapping performance data to graphical characteristics. Second, because usefulness is so dependent on the user and their application, the ability to modify and interact with a visualization is critical. Finally, a visualization environment should be adaptable to user needs and specific application contexts by focusing more on robust visualization techniques and specification rather than specific visualizations.

This thesis presents a methodology and implementation for performance visualization development that addresses these issues while greatly reducing programming overhead, facilitating rapid prototyping of displays, and allowing for effective iterative design and evaluation. By applying the tools of scientific visualization to performance visualization, this work demonstrates that next-generation displays for performance visualization can be prototyped, if not implemented, with existing data visualization software products using sophisticated graphical techniques that physical scientists have used for several years now.

The remaining chapters of this thesis fall into two main parts. Chapters I-VI provide a detailed description of the visualization process, while Chapters VII-IX each contain case studies of the use of this visualization process in specific parallel program and performance visualization contexts. Thus, this research is further motivated in Chapter II. Related research work is summarized in Chapter III, and a brief description of the IBM Data Explorer visualization package, the primary visualization development platform, is given in Chapter IV. This is followed by a detailed examination of the methodology in Chapter V, and a description of the visualization development process that was developed by applying the methodology to a particular implementation environment is found in

Chapter VI. A case study focusing on the development of new visualization concepts is explored in Chapter VII. The second study, in Chapter VIII, focuses on the use of visualization for the evaluation of parallel data distributions. Chapter IX documents the development of scalable visualizations for data-parallel programs. An informal evaluation of this visualization development technique in Chapter X is followed by a summary of the work and some possible directions for future research in Chapter XI.

CHAPTER II

MOTIVATION

The introductory chapter suggested a few characteristics of a sophisticated visualization environment for parallel programs and performance. This chapter will develop those concepts and others more thoroughly.

By now, the importance of graphical visualization in parallel performance evaluation has been established by the presence of tools such as ParaGraph [9,10], Sieve [42], Pablo [33], Voyeur [44], Seeplex [4], and Traceview [25]. These environments offer the parallel programmer access to information and insights that might otherwise go unobserved. Whether a parallel program is executing on 4 or 4,000 processors, the ability to view performance data in a useful way often enables the programmer to identify anomalous behavior within a program. Subtle changes in the way a program performs its computation, for example, could offer substantial improvements in performance. Without visualization techniques to aid in the discovery of such problems, performance improvements might never be found [9].

Clearly, performance visualization is not a panacea to the performance issues facing parallel programmers, though. As suggested earlier, visualizations have to be "useful" - in a way that elucidates performance behavior. However, "useful" is a subjective term. For instance, in the domain of scientific visualization, a display that is "useful" to the scientist in a jet propulsion laboratory may be inappropriate to the researcher analyzing ocean currents. The notion of "useful" as it applies to scientific visualization applies to performance visualization as well. The important point is that if a visualization helps a single person do their work better, then the visualization should be considered useful.

Currently, though, visualization techniques force performance tool developers to predetermine (*i.e.*, before implementation) what set of visualizations may be useful to the most people. Determining the effectiveness of visualizations is difficult without usability case studies (such as [9]) and a more formal evaluation framework (which might incorporate the ideas in [29]). While application-specific displays have their place in performance visualization, displays can be meaningful to large numbers of people. The success of the tools mentioned above is testimony to this fact. However, the degree to which a visualization can be considered general-purpose or application-specific is a difficult quantity to determine. Nonetheless, both theory and practice strongly suggest the need for a wide range of application-specific visualizations to augment a general-purpose set. To date, this need has been difficult to fulfill because of the considerable overhead in creating, evaluating, and formalizing performance visualizations, but its importance has been documented by Stasko and Kraemer [46]. Heath and Etheridge [10], creators of the general-purpose ParaGraph displays, even acknowledge the importance of application-specific displays:

> In general, this wide applicability is a virtue, but knowledge of the application often lets you design a special-purpose display that reveals greater detail or insight than generic displays would permit. (p. 38)

Unfortunately, such displays are not easily created in a tool like ParaGraph since they require special programming skills [10]. Clearly, a development process that requires little overhead and programming would enable developers to generate application-specific displays quickly in response to user needs, as well as create and evaluate general-purpose visualizations.

Computer animations of the ozone hole, a thunderstorm, or ocean currents - all appropriately described as application-specific displays - have become commonplace in scientific visualization. How have other scientists been able to overcome the overheads involved in visualization development? Scientists use generalized data visualization software products that have most, or all, of the tedious graphics and data manipulation programming already done, although there is still a creative process involved in constructing scenarios for visualizing scientific data. Performance visualization developers, on the other hand, have heretofore chosen to develop dedicated graphics and data manipulation support from the ground up, inadvertently limiting the type and variety of displays available to the user. In other words, tools developers have focused on providing the visualizations rather than robust environments and techniques to create them.

As parallel computing architectures, environments, languages, and applications continue to advance, performance visualization needs will become more demanding. Most existing tools are limited to two-dimensional displays, offer little customization and display interaction, and have strict data formats. Three-dimensional visualization in conjunction with advanced graphical techniques has opened up entirely new possibilities for researchers in scientific fields, and it stands to do the same for performance visualization.

To determine whether the field of performance evaluation can benefit by such next-generation visualizations requires a means for rapidly prototyping and evaluating new displays and display techniques. To apply the development process of existing performance visualization products to these new ideas would be to start coding hundreds of three-dimensional graphics routines and interaction techniques, not to mention advanced data representation and manipulation capabilities. Many months later, a researcher might be in a position to begin prototyping and evaluating new visualizations.

The methodology proposed herein is based on a formal foundation in which performance abstractions are mapped to visual abstractions [28]. In general, the methodology defines a framework for interfacing to existing visualization systems (*e.g.*, IBM Data Explorer, AVS), graphical programming libraries (*e.g.*, OpenGL), and other graphics resources. In this manner, an existing visualization package is but a single means of implementing the formal, high-level abstractions. While this work has focused on the use of IBM's Data Explorer, any number of similar products could also be applied. At the very least, this methodology allows visualizations to be prototyped quickly with minimal overhead, making displays available for evaluation without committing months of programming to a visualization project. The process avoids graphics programming completely, yet maintains access to numerous display styles and interaction techniques. In essence, developers are able to focus on the visualization design rather than the underlying implementation of data models and low-level graphical operations. This research offers a new technique for the development of parallel program and performance visualizations. Even if the scientific visualization package isn't suitable for the final implementation, researchers will at least be able to perform much needed usability tests and determine if their displays are effective before final implementation begins.

As a final note, while greatly facilitating the evaluation of new and existing visualizations, this research does not have evaluation as its goal at this point. Many visualizations will be presented in the pages that follow. Any "evaluation" that may take place is ultimately aimed at the development techniques being explored, not the visualizations themselves.

CHAPTER III

RELATED WORK

Many research results and discussions of visualization tools have appeared in the literature in recent years. This chapter summarizes many of the key results that have contributed to this research effort.

A recent summary of current visualization tools for parallel systems can be found in [21] where Kraemer and Stasko summarize the current state-of-the-art in visualization development for parallel systems in general (*i.e.*, parallel debugging, program visualization, and performance evaluation).

Tools such as ParaGraph [9,10], Pablo [33], and Seeplex [4] have become popular because of their portability, scalable displays, and complete performance visualization environments. The displays offered by these products can be very effective for general parallel performance evaluation. Typically, however, the creation of application-specific displays either requires considerable programming (*e.g.*, X Window programming) or is simply unavailable. With regard to ParaGraph, Heath and Etheridge [10] admit:

> Unfortunately, writing the necessary routines to support an application-specific display is a decidedly nontrivial task that requires a general knowledge of X Windows programming. (p. 38)

Stasko [46] explains the dire need for application-specific displays in the context of parallel program debugging, which he differentiates from performance evaluation by claiming that performance visualizations do not focus on the semantics of a particular program. As explained in the introduction, however, performance evaluation can be enhanced by creating visualizations which are linked to the semantics of an application. In this way, the visualization concerns of parallel program debugging and performance evaluation do intersect, and Stasko's reasons for application-specific displays become relevant to this work as well.

Many of the philosophies underlying this research are echoed in the work by Sarukkai and Gannon. In [42], they contend:

> The lack of a generalized approach for the treatment of the performance data has lead to the use of ad-hoc means of developing performance visualization systems.
>
> A truly programmable system should provide a means of easily obtaining the desired visualization and still not be tied to specific architectures or programs. To achieve this, the visualization mechanism should not be tied with the semantics of any event in the trace file. Instead it should provide a means of mapping subsets of all events in a trace file and different fields in these events to different axes of a figure and to different graphical objects such as circles, points, lines or 3-D objects.
>
> Finally, a powerful visualization tool should provide some sophisticated graphical editing capabilities such as zooming into specific locations of windows, multiple color maps, overlaying of figures, etc. (pp. 158-159)

Data visualization software, coupled with the design process proposed in Chapter V, provides some of the capabilities identified by Sarukkai and Gannon. In general, the separation of data transformation and graphics makes visualizations independent of the trace data's semantics. Flexible data models offered by most scientific visualization packages simplify mappings between data and graphical characteristics and rendering techniques, and sophisticated display interaction techniques are also supported.

Sarukkai and Gannon also make a case for the importance of application-specific displays and rapid prototyping to enable more effective evaluation:

> While it is convenient to have predefined visualizations of programs, the problem with such tools is that it is not easy to rapidly test new visualizations.... (p. 158)

The use of prototyping tools has been established by systems such as Pablo [33] and Polka [46]. Pablo promotes itself as a performance tool prototyping environment that allows and supports end-user applications. That is, the prototyping environment is the same as the one used by the end-user, but Pablo provides little support for new visualization prototyping. Polka can be used more effectively for the rapid development of algorithmic animations but is primarily suited for sequential programs.

An essential feature of next-generation visualizations is customizability of the displays. Pancake covers this topic as it pertains to parallel debugging in [31]. As with Stasko's work, many of the concepts discussed are also relevant to parallel performance visualization. Pancake's point is that visualizations based on the user's conceptual model can be more meaningful than those which are not. Therefore, giving the user the ability to customize and control visualizations should result in more meaningful and useful displays. Clearly, this notion is applicable to both debugging and performance visualizations. In [37], Roschelle argues that meaningful visualizations are not necessarily those that are consistent with an expert's mental model. Rather, users should be able to experiment with visualizations and develop their own understanding of the data. Thus, both researchers support the importance of customizable displays.

In [38], Rover proposes a paradigm that treats performance data similar to any distributed data (*i.e.*, program and system data) in the context of the data parallel programming model. Rover states:

> Visualization displays this performance data for perusal, employing the same presentation techniques in place for data visualization, such as animation, image transformations, color manipulations, statistical analyses. (p. 149)

In her conclusion, Rover states that existing scientific data visualization resources can be effectively applied to performance visualization. In a similar manner, this work treats performance data like scientific data and develops a methodology for applying scientific visualization tools which contain the presentation techniques identified by Rover to the problem of parallel performance visualization.

The literature shows at least one documented use of existing visualization software products for performance evaluation. In [39], Rover utilizes AVS and Matlab to generate performance displays. Her approach is similar to the design process proposed in this thesis in that performance data is collected, transformed into a format readable by the software, and then displayed using that software. The tools were used to generate simple, two-dimensional displays. This research both formalizes the approach and extends the use of such products into the development of new, more complex visualizations.

Reiss and Sarkar [34] present a sequential program analysis environment in which visualizations are defined as abstractions using queries over an object-oriented database of information about the program. Roman and Cox [36] define a program visualization as a mapping from programs to graphical representations. The methodology described in this paper combines these ideas and applies them to the context of parallel programs and performance by mapping abstract performance data to visual and graphical characteristics.

CHAPTER IV

DATA VISUALIZATION SOFTWARE

Before describing the visualization methodology and the implementation that forms the framework for this research, a brief overview of the general data visualization software will be useful. This chapter summarizes that software, IBM's Data Explorer.

It is important to emphasize that Data Explorer is not unique in its capability to create sophisticated visualizations and to be integrated into the formal methodology described in Chapter V. The high-level abstractions can manifest themselves practically in many ways, including other data visualization software. Data Explorer was available and seemed an appropriate test-bed for this research, though it is only one of many products that could fulfill the role defined by the methodology. The end of this section offers a list of several other similar products.

Data Explorer (DX) is an advanced data visualization software package produced by International Business Machines (IBM) which accommodates both developers and end-users of visualizations. Visualizations can be built by creating a visual program in the graphical user interface or by using a scripting language. The end-user can control many aspects of a visualization through the interface, while more advanced users can create new visualizations by adding, removing, or changing modules within the current visual program. DX also allows the creation of new, high-level data-processing or graphical modules.

The DX data model is extremely flexible, though it requires some time to understand and use. It can handle everything from rendering very regular, mesh-connected structures to completely arbitrary sets of polygons. At its heart, the DX data model utilizes sets of positions, connections, and data values associated with one or both of these sets. This simple foundation offers tremendous flexibility in what the model can represent.

In addition, Data Explorer supports a high degree of data and software reusability. Because DX data is self-describing, visual programs retain significant generality and can create a wide class of visualizations. Furthermore, a single DX data file can easily be processed by multiple visual programs. (As will be discussed later, different visualizations are created by altering the trace transformation, the visual program, or both.)

The numerous graphical techniques available in DX can be combined in hundreds of ways, provided that the data satisfies the requirements of the given techniques. DX modules generally do the "right thing" for the data they receive because the data is self-describing. For example, a module which annotates a set of locations with rendered objects whose size and color are dependent on some data value will choose the appropriate graphical technique depending on the dimension of the data (*e.g.*, spheres for scalar values or vector arrows for three-dimensional data).

The Data Explorer software runs on several architectures, including IBM RS/6000, Hewlett-Packard, Silicon Graphics, and Sun workstations. Additional information on Data Explorer can be found in [14] and [24]. Obviously, Data Explorer is not the only general data visualization software available. Some other similar products include AVS by Advanced Visual Systems, Data Visualizer by Wavefront Technologies, IRIS Explorer by Silicon Graphics, and PV-Wave by Visual Numerics. In the public domain, there exists the Geometry Center's Geomview/OOGL and NCSA's Polyview products. Any of these packages could be integrated into the methodology equally as well.

CHAPTER V

METHODOLOGY

The work described in this thesis exists in the context of a high-level abstract model that has guided the performance visualization research at the University of Oregon [28]. This chapter describes that model, and explains the aspects of it with which this research effort is concerned.

At the heart of the abstract model is the claim that if performance visualization is to play a key role in the evaluation of parallel computers and programs, then it must be established on a formal foundation that relates abstract performance behavior to visual representations (*i.e.*, a "visual performance abstraction"). The binding of a performance abstraction to a view abstraction, mapping performance object outputs to view object inputs, represents a performance visualization [28].

The notion of viewing a performance visualization as a mapping from performance data to a graphical representation was introduced in Chapter I. The absence of concrete, physical models to contextualize performance visualizations implies that such displays may take on a more abstract, impressionistic nature, potentially complicating interpretation and usability. By imposing an abstract model (*i.e.*, structure) onto the development of performance visualizations, some of these inherent difficulties may be overcome. By providing a set of graphical techniques and a robust environment for mapping performance data onto them, visualization flexibility is maintained, but structure is preserved. In other words, the user is provided with a fixed set of visual techniques rather than a fixed set of visualizations. Some preliminary research suggests, for example, that a vast array of visualizations can be modelled as hierarchical graphs of nodes and links.

A *performance abstraction* is the representation of how performance complexity is managed (*i.e.*, desired performance data analysis) and the performance characteristics to be observed. A *view abstraction* is a representation of the desired visual form of the abstracted performance data, unconstrained by the limitations of the graphics environment. Together, these two descriptions can be used to produce performance visualization software by generating interfaces to available graphical programming libraries or existing data visualization systems. Using this methodology, new parallel performance visualization environments can be developed and studied.

As seen in Figure 1, through a process of abstraction, performance data generates specifications for the performance analysis and the visual representation of the data. In practice, these abstractions would best be facilitated through a specification language (which is currently under development at the University of Oregon); the particular implementation environment, Data Explorer, does not currently offer support for this aspect of the methodology. Specifications are subsequently instantiated to implementable objects. (The term objects is intentionally general to accommodate the many realizations possible in different implementation environments.) Figure 1 indicates an overlap between performance and view objects, suggesting an interdependence between their implementations. In the ideal implementation environment, performance and view objects would be totally independent, but this research has found that the degree to which these entities can be implemented independent of one another is determined by the environment in which the implementation is taking place. In Data Explorer, this overlap is present in varying degrees. Next, the instantiated objects are combined to create the visual representation of the performance data. Finally, the actual rendering of the visualization takes place through a toolkit which interfaces with the various graphics resources being utilized. The explicit existence of the toolkit is also dependent on the implementation environment (*e.g.*, in Data Explorer, the "toolkit" interface is imbedded within the visualization system itself). For more information about this methodology, see [28].

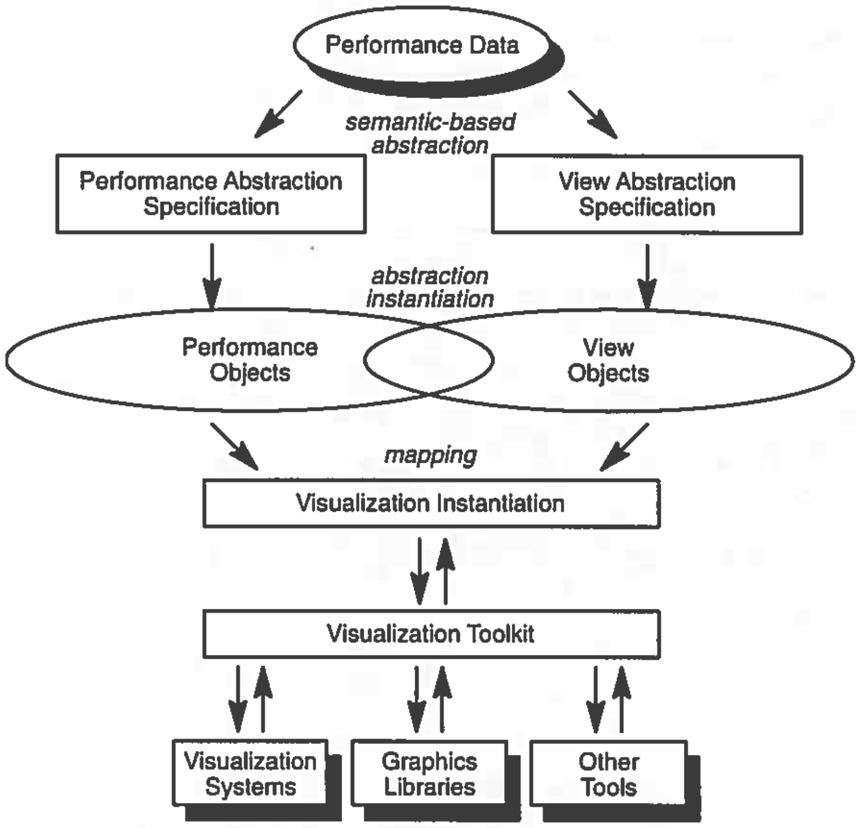

Performance Data
semantic-based abstraction
Performance Abstraction Specification
View Abstraction Specification
abstraction instantiation
Performance Objects
View Objects
mapping
Visualization Instantiation
Visualization Toolkit
Visualization Systems
Graphics Libraries
Other Tools

Figure 1. A high-level, abstract view of performance visualization treats a visualization as a mapping from performance data objects to graphical view objects and promotes the development of interfaces to existing graphical resources.

The research documented in this thesis is concerned with the application of this methodology in a specific implementation environment - a scientific visualization software system; in this case, that system is Data Explorer. As mentioned earlier, realizing this methodology in a specific environment is not always straightforward, and the correlation between abstraction and implementation is not always cleanly defined. It was mentioned above that Data Explorer does not offer support for the specification of abstractions, and consequently, this was not the focus of this research effort. On the other hand, it was possible to implement the concepts of performance and view objects in the context of an existing data visualization system. The process that has evolved implements performance objects by structuring raw trace data so that the visualization system can process it. View objects manifest themselves in the way a developer creates a visualization within Data Explorer. These correlations will be explained further in the next chapter.

CHAPTER VI

IMPLEMENTATION

It was mentioned previously that the methodology described in Chapter V is open to a variety of implementations. The methodology offers a framework for creating interfaces to existing graphical resources, an efficient means of simultaneously gaining access to a wide variety of graphical capabilities and minimizing graphical programming. In this way, an instantiation of Chapter V's methodology yields an environment in which visualizations are actually developed. This chapter discusses one such implementation that uses the Data Explorer software.

The visualization development process that has evolved from working with Data Explorer is illustrated in Figure 2. Performance visualization starts with raw trace data or statistics. The fundamental steps of this process transform the trace data into a data object file and a visual program. While trace transformations manifest themselves physically (*i.e.*, as some real program or function operating upon the trace data), graphical transformations in this implementation environment are a mental process by which analysts merge the capabilities of the visualization environment, their knowledge of the performance data, and a visualization concept - or, more formally, a view abstraction - to construct a visual program in DX that drives the creation of the desired display. It is important to note that the specification of a visualization in Data Explorer (by a visual program) does not fulfill the role of the view abstraction specification of Figure 1; that aspect of the methodology is not explicitly defined in this environment, but is, instead, embodied in the mental transformation process. In an alternative implementation environment, or as an extension to this work, abstraction specification could play an important role in developing the mapping from performance data to graphical representation.

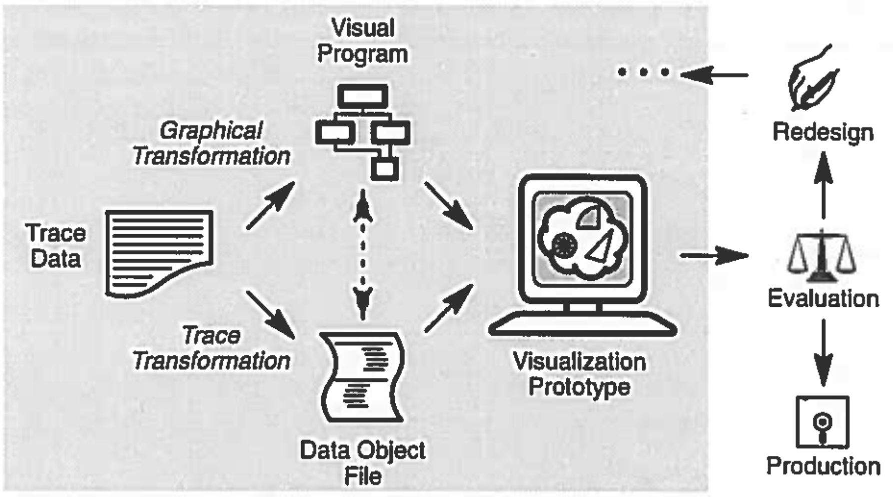


Figure 2. A development process based on the abstract performance visualization methodology can be realized using existing data visualization software.

The trace transformation may perform several operations and reductions on the data, but it ultimately creates a data object file that can be interpreted by the data visualization software. The content of these DX data object files maps nicely to the concept of performance objects. In fact, that is exactly what DX data files contain - data objects that have been constructed by the transformation of performance data. From this specially formatted data file, a visualization prototype is created by executing the visual program. The resulting display can be manipulated in many ways, including rotating, zooming, and travelling through or around the objects in the image by capitalizing on the capabilities available in the software.

Figure 2 also shows how the performance visualization prototyping environment can be integrated into an iterative design and evaluation process. Though not explicitly shown in the figure, the redesign of a visualization is accomplished by modifying the trace transformation, the graphical transformation, or both. Eventually, the visualization will be ready for production. If the software package is an insufficient environment for the final implementation (*e.g.*, because of cost or performance), then a graphics-level implementation of the visualization is appropriate at that point. Note that this occurs after the iterative design and evaluation process is complete. Ideally, though, the prototyping environment and implementation environment are the same, in which case the visualization prototype would be in its production version immediately after evaluation of the prototype is complete.

Figure 2 indicates the interdependence between the visual program and the data object file by a dashed arrow between them. In an ideal development environment, performance objects and view objects are implemented independently of one another. However, dependencies between data objects and visual programs exist in some cases. For example, much of the trace processing required to create a visualization like Figure 12 (p. 42) is in representing the polygonal facets of the cylinder. Essentially, the data object file is a precise specification for the graphical structure. Ideally, though, the developer should be able to rely on the visualization environment to provide that transformation. In other words, this sort of processing should really be part of the graphical transformation (*i.e.*, the creation of the visual program in DX). In an end-user visualization tool, such processing would be inherent in the visualization development environment, not programmed externally by the user. However, in a prototyping environment where new visualizations are created for evaluation and then modified, this sort of processing is more easily managed as part of the external trace processing. The switch from prototyping to production sees this additional trace processing ported into the implementation environment.

In fact, DX can accommodate the creation of user-defined modules that accept more general, abstracted trace data and perform the necessary structural transformations.

A developer could write a DX module that performs a specific transformation on trace data and easily integrate it into the visualization environment. This transition would make the implementation environment more consistent with the methodology from which it came, but since prototyping visualizations enables development and evaluation, this aspect of the implementation environment has not yet been explored.

In this process, graphics programming is avoided prior to production, the developer is able to focus on the visualization rather than the code that generates it, and visualizations are quickly created for evaluation, modification, and redesign. This is possible because changes to visualizations are made more easily and have fewer implications in the prototyping environment than in current performance visualization tools. Compared to existing performance visualization techniques, this method is different in that it separates the development phase from the production phase. As the area of visualization evaluation advances, a decoupled development process will be important so that modifications may be made quickly and easily. The application of this methodology creates a process that is a step toward that goal. The following sections will explore trace and graphical transformations more deeply.

Trace Transformation: Creating the Data Object File

Raw performance data is most effectively processed by Data Explorer when it is transformed into the DX data model. This section discusses some of the issues involved in creating such trace transformations.

Trace Analysis

By using an existing software product, part of the visualization problem becomes one of trace transformation. To take advantage of Data Explorer's rich data model the trace files or performance data must be transformed into a format that Data Explorer can process. A single trace file may take on several different representations within Data Explorer depending on the visualization desired. On one hand, a transformation may simply augment the existing trace data with the appropriate keywords and structure. Alternatively, it may perform extensive computations and/or reductions on the data set before it generates the Data Explorer file. It is important to note that the problem of trace analysis still exists. That is, in most cases the trace transformation is responsible for any analysis or reduction of the trace data. However, one of the advantages of the method being proposed here is that the transformation of trace data is done independently of any graphical representation of that data, a concept promoted in [33] and [42], and a key characteristic of the high-level methodology. Thus, transformations are easily modified and can be used with several different graphical techniques. In an environment of creation, testing, and evaluation, the ability to make changes with minimal intrusion on the rest of the system allows for more rapid prototyping of the displays.

Trace Transformation Programming

Trace transformations can be easily implemented in traditional programming languages, and do not require any special programming skills. It is even possible that existing performance tools can be used for more involved trace analysis problems, and then to generate data files suitable for input to Data Explorer. The underlying requirement, though, is an understanding of the data model for the product being used. In the case of DX, experience suggests that it should take no more than a couple weeks of studying examples and experimenting for an individual to become very comfortable with DX's data model. Once an understanding of the data model is achieved, trace transformation programming follows quickly and easily. Data Explorer's data model is rich enough for general scientific visualization and as a result, offers many alternatives for performance visualization.

Self-Describing Data

Data Explorer's data model offers significant advantages over traditional data representation schemes. Because Data Explorer data is "self-describing" and modules are designed with this in mind, a single DX visual program can generate many different displays depending on the data it processes. (The notion of self-describing data can also be found in Pablo's Self-Describing Data Format (SDDF) [33] and NCSA's Hierarchical Data Format (HDF) [30].) The data files imported by DX contain structural object information which determines a set of possible visualizations. It is up to the visual program to extract and process the desired information from the data file. Many visual programs are reusable, requiring only minimal changes, if any at all.

Graphical Transformation: Creating the Visual Program

In this section, several features of Data Explorer (and many other scientific visualization packages) pertaining to graphical transformations will be discussed in the context of parallel performance visualization.

Visual Programs

Once a trace has been transformed into the DX data model, it can be imported using either DX's visual programming language or its scripting counterpart. The visual programs translate directly into the more general scripting language, both of which are best described as functional and data-driven. Since the DX programs created for this work were usually quite simple, the visual programming environment was adequate. A switch between the two programming styles could have been made at any time if it had become necessary, however. Programs generally consist of three phases: importing and selecting, processing, and then rendering the data. One of the main advantages to using a product such as Data Explorer is that all of the programming that is necessary to implement these three phases is already done for the developer.

Figure 3 contains a visual program created in Data Explorer. The graphical representation of a DX function is a module with sets of "tabs" on its top and bottom, corresponding to inputs and outputs, respectively. By connecting one module's output tab to another module's input tab, the user assembles a network of modules - a visual program - that specifies and controls the visualization. Connections between modules indicate the flow of data through the network.

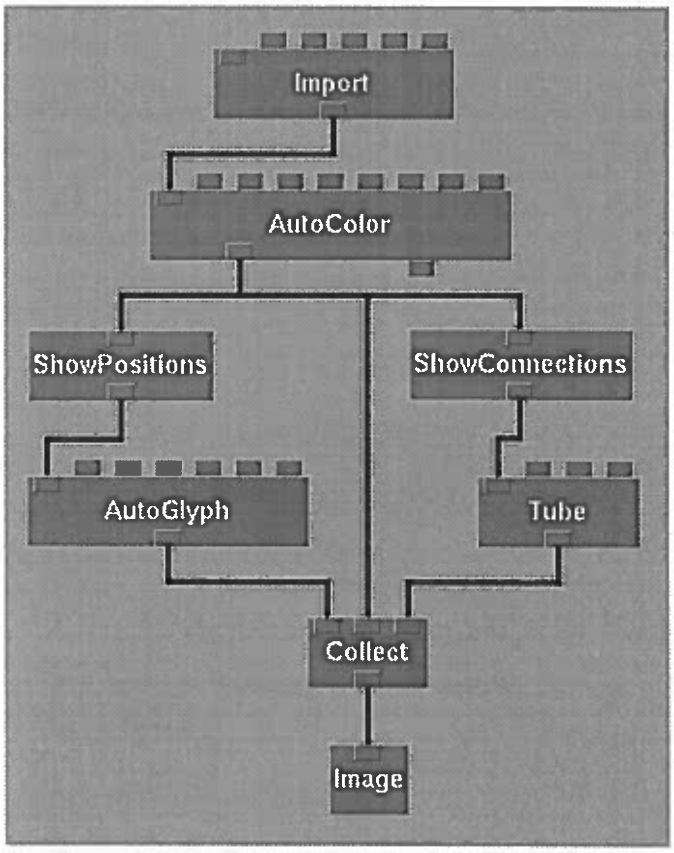
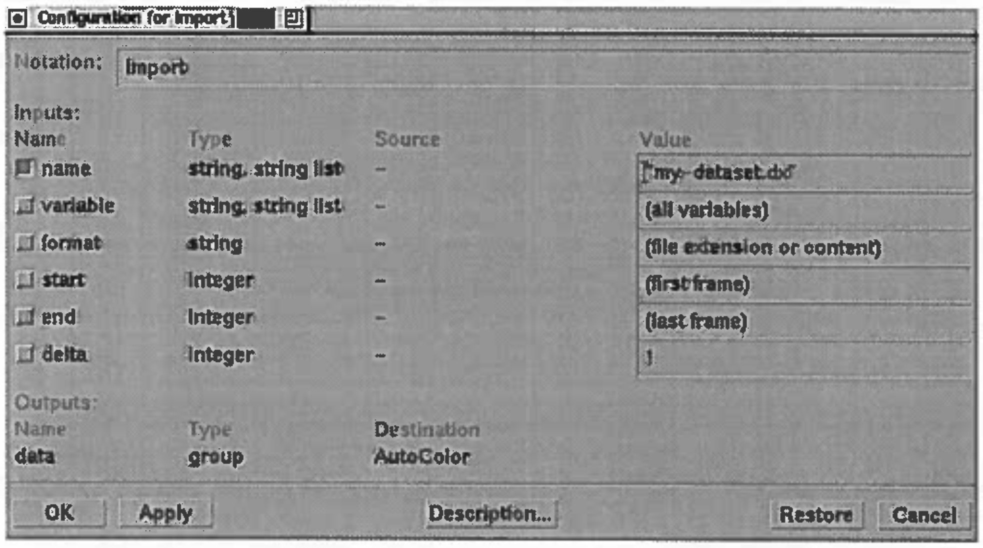


Figure 3. A simple visual program in Data Explorer is capable of creating many different types of visualizations, as seen in Figure 8.

Customization

Forming the basis of the customization opportunities within DX, modules of a visual program usually have default values that do the "right thing" to the data. However, in certain cases, it is desirable (or required) to set certain parameters. For example, to import data, the user selects the Import module from the DX menus and places it on the programming canvas. (The visual program in Figure 3, like most DX programs, uses the Import module.) The Import module requires the user to tell it which data object file is to be read. To set the parameters of a module, the user begins by double-clicking on the module, which opens into a window like the one in Figure 4. Next, the user types the appropriate information into the desired fields (*e.g.*, the "name" field in the Import module).


Configuration for Import

Notation: Import

Inputs:

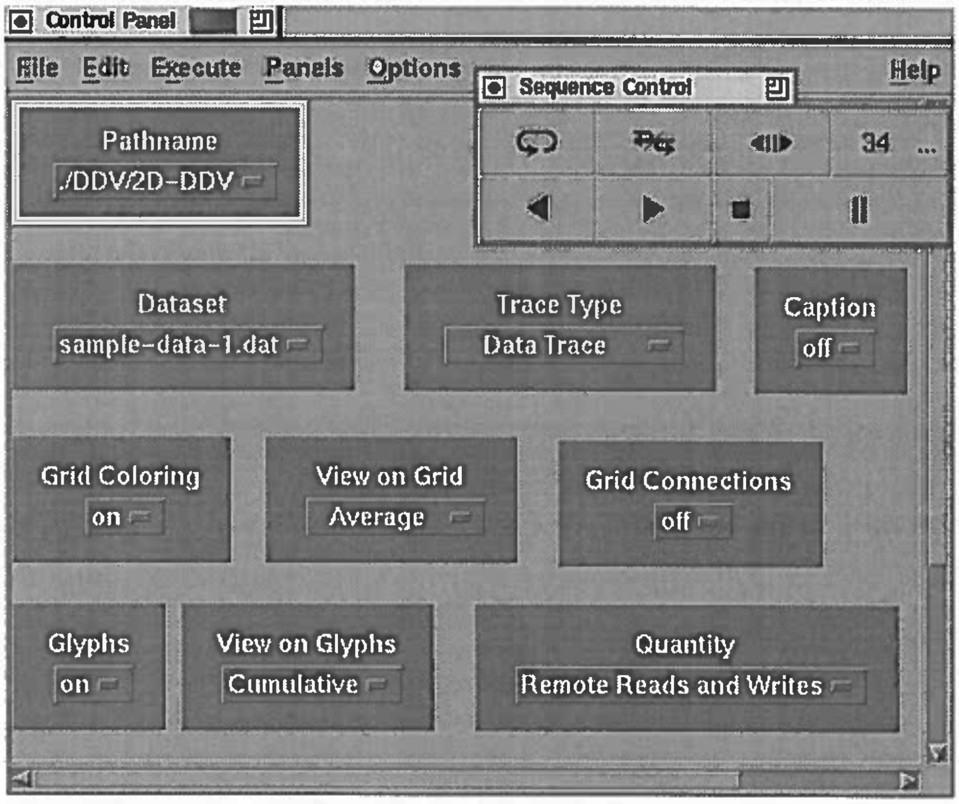
Name	Type	Source	Value
name	string, string list	–	"my-dataset.dx"
variable	string, string list	–	(all variables)
format	string	–	(file extension or content)
start	integer	–	(first frame)
end	integer	–	(last frame)
delta	integer	–	1

Outputs:

Name	Type	Destination
data	group	AutoColor

Figure 4. The Import module reads a data file into the visualization environment.

Data Explorer offers other techniques for controlling module parameters which allow the user to more easily interact with and "tweak" the characteristics of a display. By connecting objects called "interactors" to input tabs, the user can create a "control panel" that allows for easy modification of any number of different parameters. An interactor appears in the visual program as a simple module (no inputs, one or more outputs) and in a control panel as a selector, a dial or slider, a text field, or some other interaction object. (Note that the visual program in Figure 3 does not contain any interactors.) Interactors are highly configurable yet easy to use, adding significant flexibility to the visualization development process. An example control panel appears in Figure 5. As can be seen, the control panel allows the user to select import data files, alter the graphical characteristics of the display, and even change the quantities being visualized. Such a flexible environment fosters the development of customizable displays and a high-degree of user-interaction, important properties for next-generation parallel program and performance displays.



Pathname ./DDV/2D-DDV

Dataset sample-data-1.dat

Trace Type Data Trace

Caption off

Grid Coloring on

View on Grid Average

Grid Connections off

Glyphs on

View on Glyphs Cumulative

Quantity Remote Reads and Writes

Figure 5. Control panels are used to create simple interfaces that can manipulate many characteristics of a visualization.

Display Interaction

Next-generation performance visualizations are bound to take advantage of three-dimensional displays. The additional information made available by this technique will be invaluable to programmers using next-generation parallel languages. However, moving to three-dimensional visualizations necessitates additional functionality with regard to visualization control and interaction.

Adding a third dimension to a visualization increases the representation potential for the data associated with a given display. Three-dimensional rendering techniques allow the viewer to see more of that data, and display interactions increase the access to and control of visual details and display attributes. However, extending existing performance visualization tools to three-dimensional displays requires more than just adding a projection routine. Because three-dimensional displays are so dependent on the angle from which they are viewed and the rendering techniques being used, tools offering three-dimensional displays need ways for the user to interact with the objects in the display. At a minimum, this would seem to require the ability to zoom in/out on any part of an object, to rotate the object arbitrarily, and to control graphical attributes such as color and transparency. More advanced tools include control over lighting models and the surface properties of display objects (*e.g.*, specularity and reflectance).

As an example of the additional features provided by scientific visualization tools, Data Explorer's color map editor, shown in Figure 6, allows the visualization developer to customize a visualization's color map. In fact, multiple color maps can be used for different objects in the display. The importance of flexible color mapping has been documented [3,35]. Visualizations can be given completely new meaning simply by changing the color map(s) associated with the display. Such features are not trivially incorporated into existing performance visualization tools but are standard in many visualization packages.

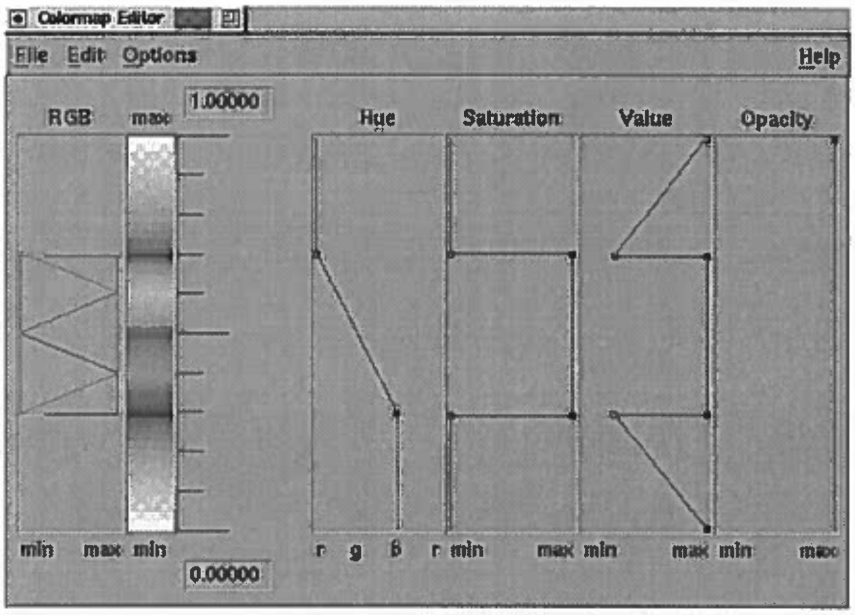


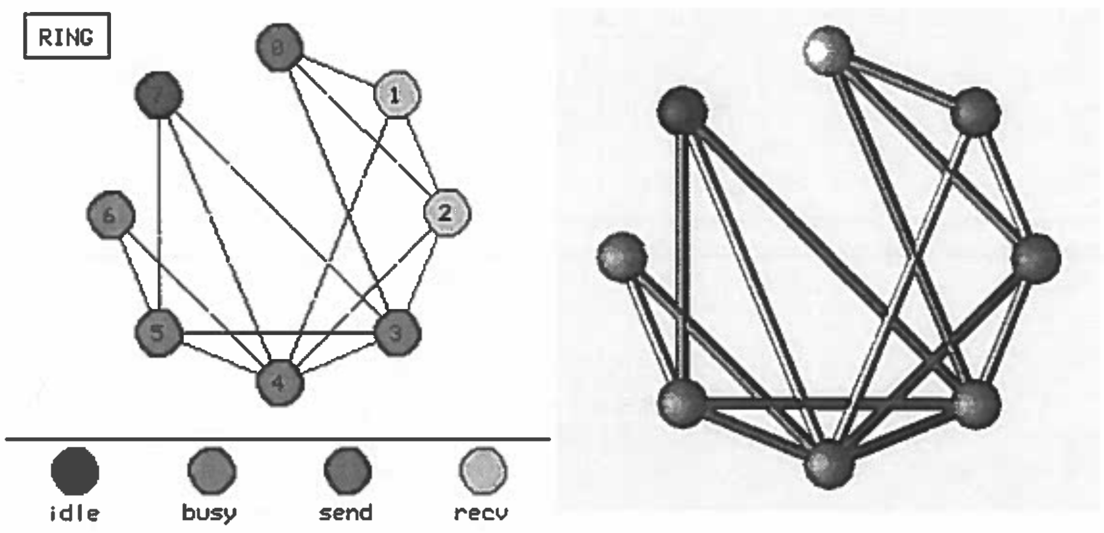
Figure 6. A colormap editor can create arbitrary colormaps for a visualization, enabling the analyst to explore and highlight different features of the represented data.

Animation

Another feature that is extremely important to performance visualization is animation, again a capability offered by most data visualization packages. Data Explorer's animation technique is similar to that of a cartoon. The DX data file often contains time series data. When visualizations for each member of the time series are displayed rapidly in succession, an animation results. This is not as flexible as the animation techniques found in some performance visualization tools (*e.g.*, ParaGraph [10] and Polka [46]) where, for example, a connection between two nodes may appear and disappear without ever having to change the surrounding image. However, once displays become three-dimensional, this type of animation becomes difficult since connections may pass through, in front of, or behind other objects in the scene, requiring complex hidden line and surface analysis algorithms for a correct visualization.

For instance, Figure 7(a) shows a display from the popular ParaGraph visualization tool, while Figure 7(b) is a prototype of the ParaGraph display generated by Data Explorer. Both displays show communication between processors. ParaGraph's display is two-dimensional while Data Explorer's is (inherently) three-dimensional, though not yet to any representational advantage. In rendering the two-dimensional display, it is known that a connection between two nodes will not interfere with any other objects in the scene (except other connections which are safely ignored since they are just pixel-wide lines) since the nodes are arranged in a circle. However, in the three-dimensional visualization, the visibility of a given connection is dependent on the orientation of the structure as well as the components making it up, which includes the other connections. The situation is worse when a truly three-dimensional display is generated. The benefit of using a tool like Data Explorer is that this display can be extended beyond the "flat" prototype into a real three-dimensional display, resulting in a much richer, information-dense display. But this added graphical complexity has its consequences as it necessitates a stricter method for animation in the generalized data visualization software products. An example of extending an existing, two-dimensional visualization is presented in Chapter VII.

In general, visualizing in three dimensions overcomes certain limitations inherent in two dimensions. For example, in 3-D there is more flexibility in layout than in 2-D, making the creation of scalable displays potentially less intractable (see the case study in Chapter IX). Advanced graphics rendering also offers more options for combining global and detailed performance visualization in a single display (as in Chapter VII). While the display in Figure 7(b) may be perfectly acceptable, three-dimensional representations offer greater possibilities to the developer and must play an integral part in the next generation of parallel performance visualization tools.



(a) (b)

Figure 7. Displays from existing performance visualization tools can be prototyped, and subsequently extended, using three-dimensional data visualization packages.

Performance Visualization Examples

In this section, examples of how the development techniques described thus far can be applied to parallel performance visualization are presented. These examples should pave the way for the specific case studies in the following chapters.

The DX data model centers around sets of positions and connections. A simple DX program might create a visualization that annotates positions with spheres and connections between positions with cylinders. Additional coloring might take place depending on the data being processed. Figure 3 contains a visual program that accomplishes these tasks for appropriately structured input data.

Performance visualizations that intend to illustrate interprocessor communication often manifest themselves in a visualization fitting the description given above. That is, processors can be represented by a set of spheres in space, while the communication between processors can be realized by links between the spheres (*e.g.*, Figure 7). Such a display can be extended in many ways and is certainly not limited to interprocessor communication.

To emphasize the use of reusable visual programs, the images in Figures 8 were all generated by the visual program in Figure 3. The only program parameter that was changed was the name of the data file in the Import module. All the other modules have default parameters that do the right thing for the data being visualized. The Data Explorer modules are able to figure out what to do with the data without the user explicitly describing it; the trace transformation process is responsible for augmenting the trace data with enough structural information so that Data Explorer modules can construct the visualization from the data. Thus, the structure and content of the data file - which is the result of a trace transformation - plays a key role in determining a visualization's appearance. In this way, a single visual program enables a set of displays to be generated. Practically, this is convenient, but theoretically, it violates the desired independence between performance and view objects, as described by the high-level methodology. Again, this is a result of the specific implementation environment and represents a design decision made so that prototyping could be better supported.

Each of the displays Figure 8 could be used in a parallel performance setting. For instance, Figure 8(a) could represent interprocessor communication in a ring topology. Similarly, Figure 8(b) could be applied to a mesh architecture where glyph size represents communication overhead, link color represents the communication load on a particular interconnection between processors, and the mesh background shows a continuous interpolation of the discrete node data, potentially useful to observe scalability characteristics. Figure 8(c) extends the previous example into three dimensions. The interior of the solid is volume-rendered to create a "cloud" of colors which can offer insight into the possible results of a scaled-up version of the application. Finally, Figure 8(d) offers a novel visual-

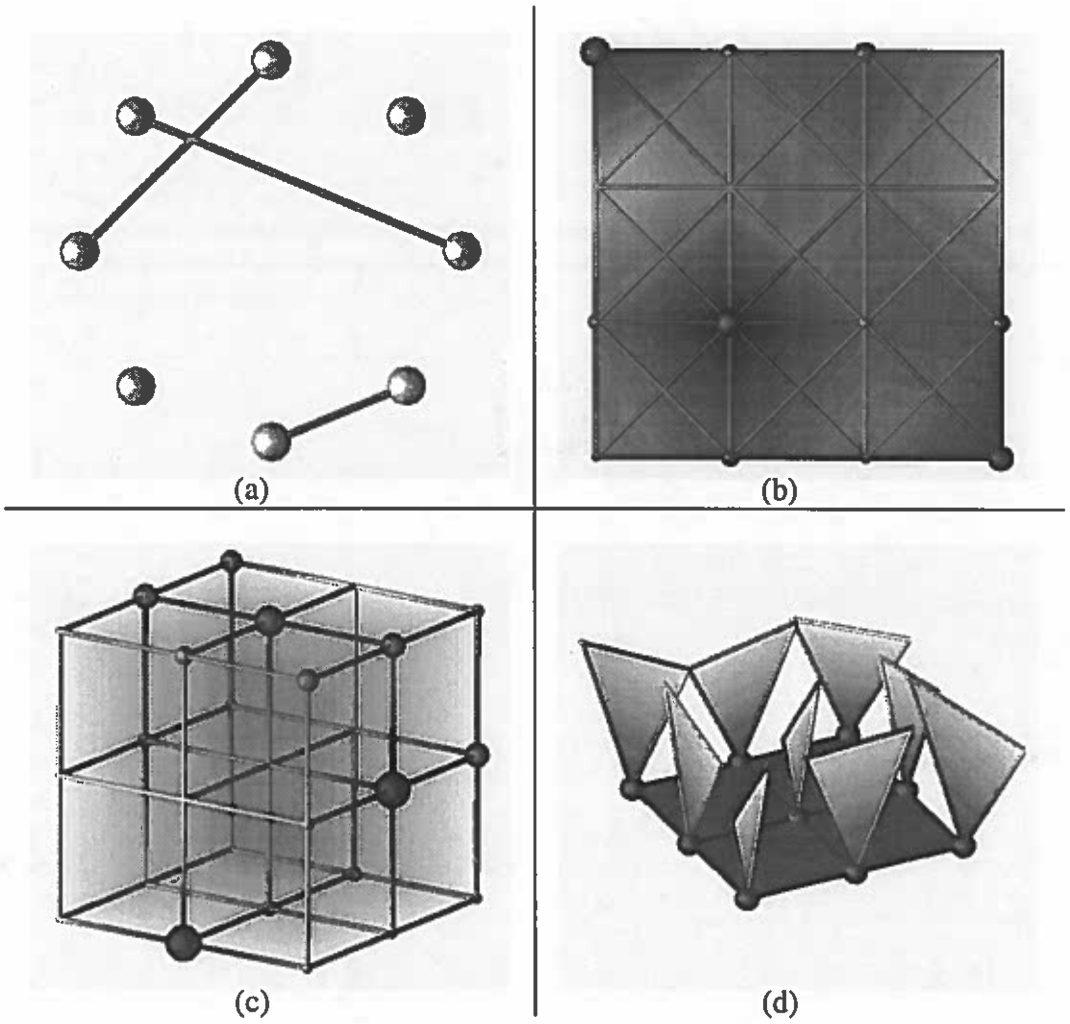


Figure 8. The visual program in Figure 3 can create a wide range of performance visualizations depending on the structure of the underlying data.

ization where processors exist in a two-dimensional grid with "sails" emanating from the glyphs. For each processor, the orientation of the sail and the height of the sail's two upper points could be controlled by a three-dimensional metric (*e.g.*, busy, idle, and overhead percentages). While these displays are significantly different graphically, they were all created by a single, simple visual program processing different data files. In terms of the methodology, the same view objects are being combined with different performance objects.

Alternatively, the same dataset can be processed by different visual programs to generate different displays, an approach common to scientific visualization. For example, a developer can apply different realization techniques to the same data by using different DX programs. One visual program may volume-render a three-dimensional structure while another creates contour surfaces within the volume. The data is the same, but the different visual programs enable different types of displays to be created.

Thus, in this methodology the developer is presented with two levels at which visualization development and modification can take place: the data object file (performance objects) and the visual program (view objects). Both can be used to control certain aspects of the visualization process, but typically one may be more appropriate than the other depending on the user's goals. If the goal is to investigate performance characteristics within a single set of performance data, then fixing the data set and changing the visual program tends to work best. On the other hand, if the goal is to compare and contrast several sets of performance data, then using a single visual program and changing the structure of the imported data can be effective. Of course, in many situations, changing both the data and the visual program generates the best results.

The strength and flexibility of a product like Data Explorer comes from both the programming behind the modules and the powerful data model it uses. The result, in terms of prototyping performance visualizations, is that displays can be created very easily and quickly. For the visualizations in this thesis, less than 100 lines of standard C code was necessary to implement the trace transformations. Also, the Data Explorer visual programs required to import and process the data files vary minimally across a wide range of visualizations - a testimony to the self-describing capabilities of the data model and the high degree of software reusability supported by DX. In all, a new visualization - trace transformation, graphical transformation, debugging, experimentation, etc. - can be developed in less than a day. Modifications to existing displays require a few minutes or less. Given that a single DX visual program may serve to create many visualizations, and of course, a single DX data file can be used in many different visual programs, the overall result is a very versatile environment for creating and redesigning performance visualizations.

To illustrate how this may benefit performance visualization developers, consider the following scenario. Suppose a certain performance visualization tool was limited to two-dimensional displays in the spirit of Figure 7(a). (Note that this supposition applies to almost all of the existing performance visualization tools mentioned throughout this thesis.) To extend such displays into three dimensions would require substantial work on the part of the tool designers and implementers. New graphic routines would have to be written, additional methods of interacting with the display would probably be necessary, and perhaps the data model would need to be extended. Using a tool such as Data Explorer, which supports three-dimensional visualizations by default, the jump from 2-D to 3-D is simply a matter of changing the data!

CHAPTER VII

CASE STUDY: ADVANCED PERFORMANCE VISUALIZATIONS

This chapter contains the first of three case studies on visualization development using the environment that has been described in the preceding chapters. As has been stated before, a primary focus of this research is to provide new visualization techniques that may be effectively applied in specific application contexts to develop visualizations useful to users within that domain. This development technology makes a wealth of new graphical capabilities available to the visualization developer. This chapter examines how such capabilities can be put to use in developing new performance visualizations [6].

Introduction

Most performance visualization tools are limited to simple, two-dimensional graphical displays. While many of these displays are very effective, the use of more sophisticated graphical techniques has gone unexplored in the context of parallel program and performance visualization. The availability of general, three-dimensional visualization packages makes a vast array of these new graphical capabilities easily available to the visualization developer. In this study, two categories of new displays are explored. First, an example of how existing displays can be extended into multi-dimensional visualizations will be given. This will be followed by some new ideas for performance visualizations.

Extending Existing Visualizations

A popular visualization from the ParaGraph visualization tool [10] is the Kiviat diagram [19]. The traditional form of this two-dimensional display has several spokes extending from a center point whose lengths change as time passes. A spoke corresponds to a single member of an object set over which a scalar parameter (or a set of parameters) is being measured. As it has been applied in ParaGraph, a spoke represents a processor in a parallel computer, and the measured quantity is the percentage of computation (*i.e.*, utilization) for that processor. Then, when the ends of adjacent spokes are connected, triangular regions result. If the quantity being represented by the length of the spokes is, say, computation, then processor utilization would be "good" when the spokes are longest. (Spoke-length is typically mapped onto the interval [0,1]. Thus, with a large number of spokes at maximum length, the Kiviat diagram approximates a unit circle, which is sometimes used as a backdrop.) Figure 9 contains an example of a Kiviat diagram from ParaGraph.

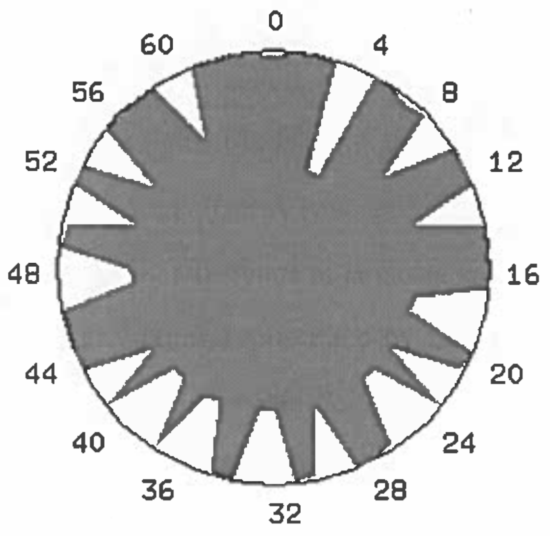


Figure 9. ParaGraph uses a Kiviat diagram visualization to show processor utilization.

Given the concept of a Kiviat diagram, one can easily represent such a structure within the positions and connections of the Data Explorer data model. The minimal amount of information necessary to create the visualization is a time series of data. Each time step contains a scalar value for each processor in the system. It was mentioned above that triangular regions result when the end-points of adjacent spokes are connected with one another. Thus, a Kiviat diagram can be decomposed into a set of triangles. In Data Explorer, a triangle is represented as three points and three connections. All triangles in a Kiviat diagram have the center point in common, and adjacent triangles share the end-point of a common spoke. If there are *n* processors in the system, then for each time step in the animation, *n+1* positions must be specified, followed by a list of connections, which is given by referencing the positions list. In essence, the representation is similar to a "con-nect-the-dots" puzzle.

Conceptually, a Kiviat diagram is easily represented within the Data Explorer data model, and the DX program necessary to render the data is only slightly more complicated than the examples discussed earlier (Figure 3). Thus, in roughly half a day, a fully animated Kiviat diagram prototype was developed from a raw trace file. Figure 10 shows a single frame of the animated visualization.

The ability to go from visualization concept to visualization prototype in just a few hours opens up entirely new possibilities for visualization developers and evaluators. However, implementing a two-dimensional visualization within an advanced visualization environment doesn't offer any additional insight to the performance data. Thus, as was suggested in Chapter VI, the next step is to see how the standard Kiviat diagram can be extended to take advantage of some of the graphical capabilities present in the data visualization software.

One of the potential problems with a standard Kiviat animation is that the viewer sees only one step at a time and can easily lose track of how the performance at a given

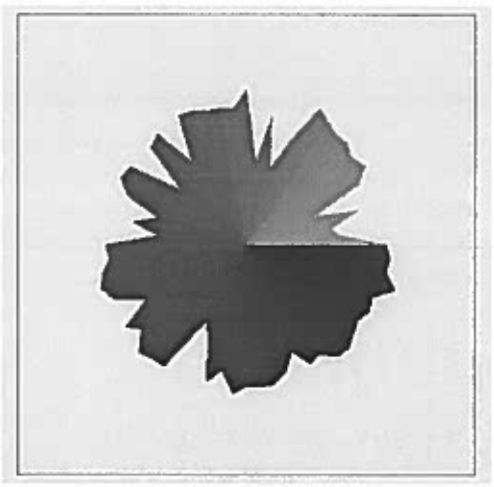


Figure 10. The traditional two-dimensional Kiviat diagram is easily implemented using data visualization software.

step compares to the performance during the rest of the animation. Thus, by removing the animation of the display and letting time run along the newly available third axis, a Kiviat "tube" results. Figure 11 illustrates how this visualization is constructed.

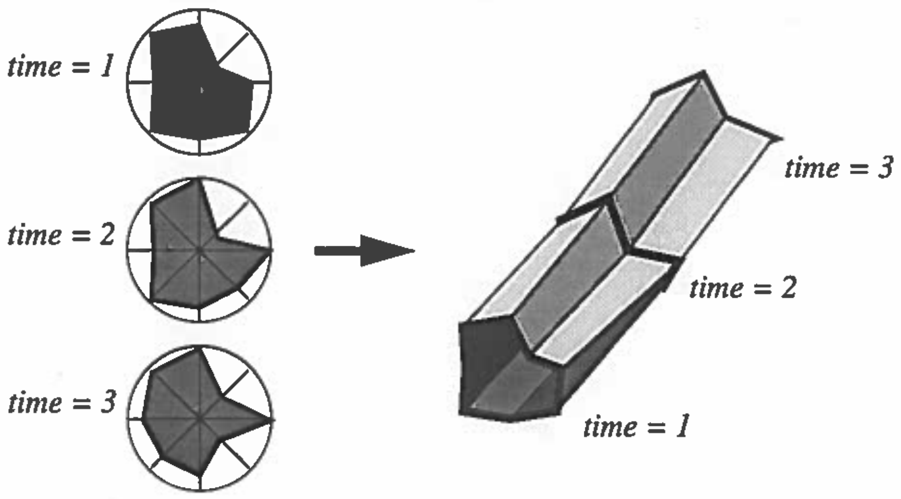


Figure 11. The two-dimensional Kiviat diagram can be extended to three dimensions by allowing time to travel along a third axis.

It is interesting to note that now the representation within the DX data model changes considerably. To render a tube with a solid exterior shell, the quadrilateral surface patches between time steps are rendered instead of the triangular sections emanating from the center of each slice. Still, the transformation is only slightly more complicated than the standard Kiviat transformation. Figure 12 shows a Kiviat tube generated by Data Explorer.

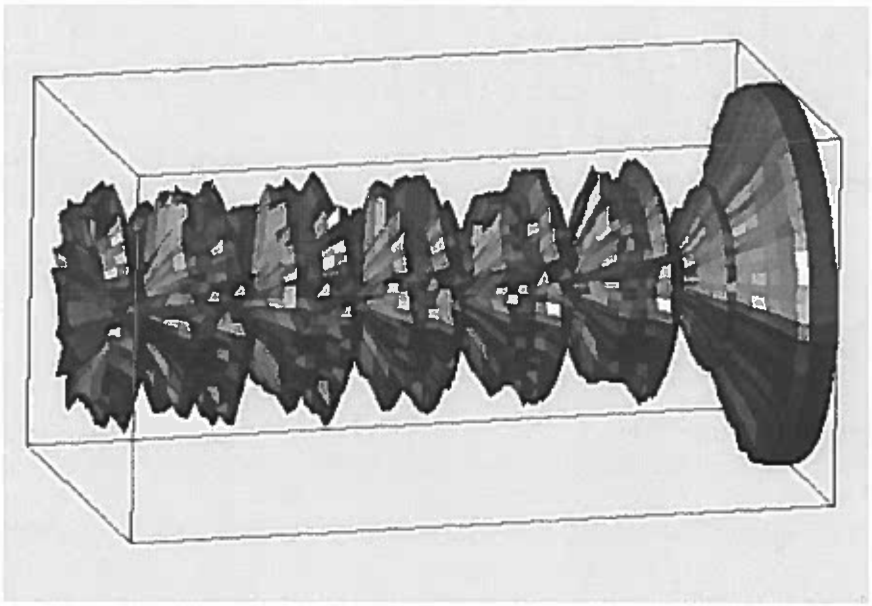


Figure 12. A three-dimensional Kiviat tube reveals global trends in the performance data.

This representation of the original Kiviat diagram is important because it gives the viewer a global view of the performance data, as opposed to the standard two-dimensional version which limits the viewer's ability to compare the performance of the application at different times during the trace. However, the three-dimensional representation tends to obscure more detailed information about individual processors at specific times, whereas the standard Kiviat display shows that information more clearly.

Some of Data Explorer's true power is revealed in the following example. It is possible that individually, neither of the Kiviat displays generated thus far (Figures 10 and 12) totally fulfills the viewer's needs. The two-dimensional display allows the viewer to assess how processors relate to each other during a given time slice, but makes it difficult to see how performance in one time step relates to other parts of the animation. The three-dimensional display tends to do just the reverse; that is, seeing trends over the life of the trace is easier, but it is difficult to see how processors relate to each other during a given time step. It may be that by combining the two displays, both needs could be met. Thus, the idea for a still more enhanced display is to let the two-dimensional Kiviat slice "pass through" a partially transparent Kiviat tube. The slice highlights the interprocessor relationships for a given time step while the rest of the tube still reveals how a particular step relates to the rest of the data. The display is animated by letting the slice slide through the tube. Alternatively, the viewer can directly specify the time step at which to place the slice.

This is a complex, advanced visualization that combines several graphical techniques. However, having previously specified the two pieces of the display individually, Data Explorer allows the developer to combine the two trivially. In what literally took just minutes, the composite visualization in Figure 13 was created. (Please note that this display utilizes transparency and is very difficult to represent in a greyscale manner.)

New Visualizations

New visualizations represent a second category of displays that can utilize some of the graphical capabilities of data visualization software systems. The reader is reminded that the presentation of these new displays is meant to illustrate the usefulness of this particular design method as opposed to that of the visualizations themselves. This category can be further broken down into two methods for developing visualizations. The first approach is analogous to the method presented in the examples above - that is, start with a concept for a visualization, and then translate it into the graphical capabilities of the available software. By definition, this is the only method applicable to prototyping extensions

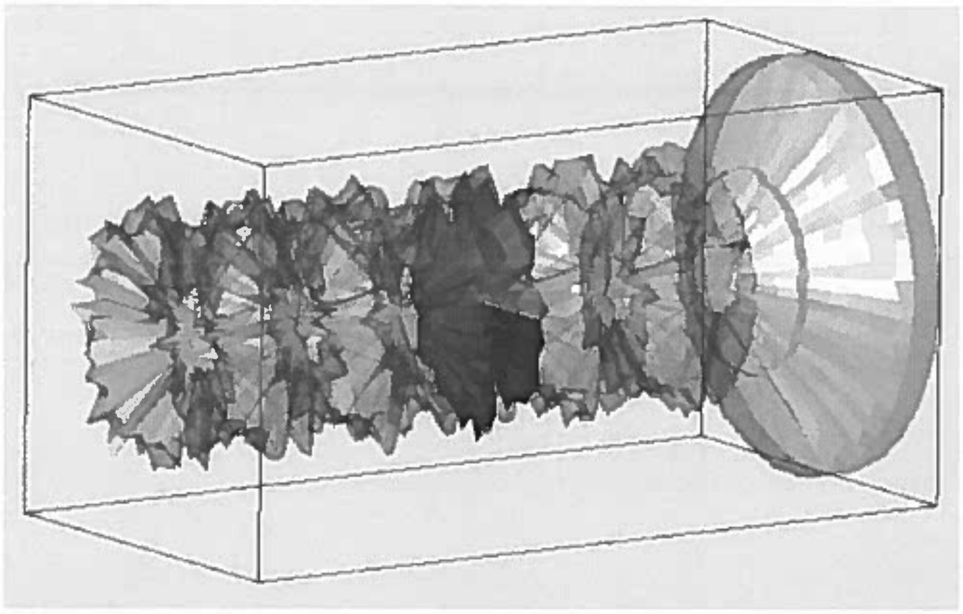


Figure 13. By combining the two-dimensional and three-dimensional Kiviat displays, a potentially more useful visualization results.

to existing displays. However, in prototyping new visualizations, many scientific visualization packages offer another, potentially more powerful, method.

Essentially, the second method works in the opposite direction as the first - start with some feature or graphical technique available in the software, and then develop a concept for a performance visualization that uses that technique. Traditionally, visualizations have been developed out of a dire need to see data presented in a certain way, but the earlier motivation of providing visualization techniques that can better accommodate the rapid generation of *new* displays clearly supports this alternative approach. At first, the thought of letting something other than need motivate a visualization may seem blasphemous or, at least, odd. However, this technique can stimulate creative ideas that might not otherwise be conceived. For the developer looking to create new and novel displays, this technique may be helpful. Of course, the value of any new visualization is unknown until it is thoroughly evaluated, and this is true regardless of how the visualization was created.

Visualization Concept to Graphical Technique

In most cases, visualizations are created by starting with an idea for a display and then figuring out how it could be accomplished using the available graphical resources. This section provides an example of this process.

In the introduction to this thesis, a visualization scenario was posed in which the visualization of molecules interacting within a three-dimensional space was compared to visualization scenarios for the processors in a parallel computer. It was claimed that there was an inherent physical model on which the molecular visualization could be based, but such a concrete model was less obvious for the parallel computer. In particular, it was suggested that molecules could be represented as spheres that moved around a well-defined three-dimensional space. This section explores the use of that same visualization *concept*, but in the context of the parallel architecture.

Three commonly traced metrics of parallel processor performance are the percentages of computation, overhead, and idle times. As percentages, these three metrics create a well-defined space in which the processors of a parallel computer exist. The concept behind the visualization, then, is to represent each processor as a sphere within that space. The location of each sphere is determined by the values of the three metrics corresponding to each processor. Thus, the axes represent computation, overhead, and idle. As time passes, the spheres, like molecules, move around the "performance space" [46].

The raw data represents a time series, and each time step contains values for the three metrics for every processor in the system. In Data Explorer, the visualization can be modelled trivially. As discussed before, Data Explorer works with sets of positions and connections. Consequently, this visualization just degenerates to a set of positions that change over time. From a set of positions, the corresponding spheres are created with the *AutoGlyph* module, as in the example earlier in this paper (Figure 3). So that processors may be distinguished from one another, the spheres are colored, also easily handled by Data Explorer. Figure 14 contains an example of this visualization.

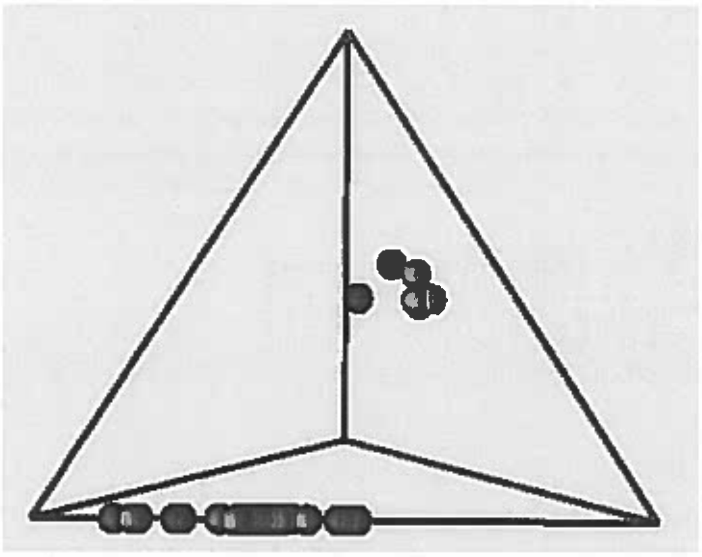


Figure 14. A three-dimensional processor performance metric determines the location of processors within the "performance space."

As with the other examples, it took less than a day to develop the basic prototype for this display. After that, Data Explorer's flexibility allows the user to customize and "tweak" the display to no end. The user has simple control over the size of the glyphs, animation speed and granularity, colors, and other features that are fixed in many performance visualization tools. These types of interactions are available directly from the visualization environment and do not require new transformations of the data.

Graphical Technique to Visualization Concept

Up to this point, Data Explorer's flexibility was impressive, yet it was evident that only the surface of its graphical capabilities had been exposed. Gradually, the visualization development process began to reverse itself as Data Explorer was used not only as a tool to implement a preconceived visualization concept, but as an aid in generating that concept in the first place. This section will offer an example of such a visualization.

Data Explorer has the capability to realize data using a technique called a “rubber sheet.” The concept is simple: a grid of positions and connections is interpolated to form a continuous “sheet”; the data values associated with each position are then used to displace (and color) that position on the sheet a distance proportional to the value in a direction perpendicular to the sheet. The result is a grid that is distorted (and colored) to reflect the data values of the grid positions.

Thus, in examining this graphical realization technique, the idea for a visualization evolved. The visualization’s goal was to provide program and performance visualization information for distributed data structures. In distributed memory multiprocessor computers, processors can read data from either their local memory or from the memory of other processors. Remote data accesses typically involve some form of relatively expensive communication, and can lead to poor performance. For a given algorithm, the distribution of a data structure affects the number of remote accesses that a processor has to make. Using a rubber sheet, it would be possible to graphically represent the difference between local and remote accesses made by processors to the elements of a distributed data structure. Such information is valuable in determining the effectiveness of a particular data distribution. (Chapter VIII contains additional information on the topic of evaluating data distributions with visualization.) Having constructed the visualization’s concept from a graphical technique available in the visualization software, all that remained was to create the trace transformation necessary to realize the visualization. Figure 15 contains several frames of the animation of this visualization.

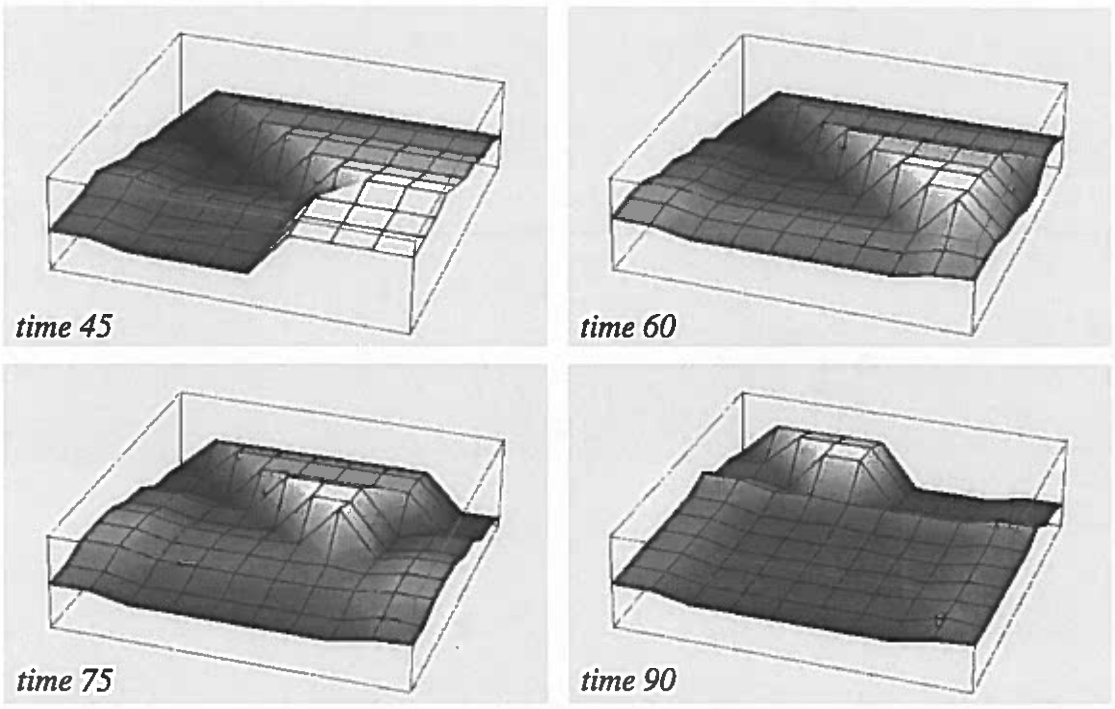


Figure 15. Vertical displacement and coloring reveal remote and local data access patterns to a distributed data structure.

Summary

In this chapter, several visualizations that take advantage of more sophisticated graphical techniques were presented. Data visualization software is capable of implementing current two-dimensional performance visualizations. More significantly, though, it allows such visualizations to be extended into potentially more useful displays. Data visualization software can also inspire the creation of new types of displays. Clearly, performance visualization developers gain access to significant power and flexibility when using general data visualization software.

CHAPTER VIII

CASE STUDY: DATA DISTRIBUTION VISUALIZATIONS

In the preceding case study, the last example demonstrated a visualization that can be used for evaluating the distribution of a parallel data structure across processors. Such displays are important since many new parallel programming languages leave the task of data distribution primarily up to the programmer. This chapter develops a specialized visualization environment, based on the visualization methodology described earlier, that can assist in evaluating data distributions [5].

Introduction

New languages such as High Performance Fortran (HPF) [11] and the parallel C++, pC++ [1] offer cohesive data parallel programming abstractions, strive for efficiency, and have portability as an underlying motivation in their design. While these languages offer levels of abstraction far removed from the physical architecture, they still give the programmer explicit control over such issues as data distributions, data alignments, abstract processor arrangements, and specification of parallel loop constructs. The results are languages whose compilers capitalize on certain recent advances in compiler technology ([12], for example), but presently leave the critical task of distributing data across processors up to the programmer. There are those people who will undoubtedly applaud this step and consider user-specified data distributions a "feature." Equally likely, though, are those who will dread being forced to figure out the best way to partition manually their data structures. Giving this ability (or burden) to the programmer, places the performance of thousands of parallel applications in potential jeopardy. Clearly, the distribution of data across processors is a deciding factor in the performance of the program operating on that data. Thus, programmers using new languages such as HPF and pC++ could benefit from evaluating different data distributions.

The goal of this case study is to demonstrate how this new performance visualization paradigm can assist programmers requiring evaluations of data distributions. The work by Kondapaneni, Pancake, and Ward [20] demonstrates how data distributions and alignments in Fortran D (a language which contributed significantly to the design of HPF) may be specified using graphical representations of program data structures. They claim that the accuracy of data distributions specified by programmers using their visual programming tool were more accurate than those made by programmers who did not use the tool. Furthermore, the graphical environment encouraged experimentation with different mappings. These results strongly suggest that the evaluation of data distributions could benefit from graphical representations as well. To this end, the development of an experimental data distribution visualization (DDV) environment - a framework from which visualization techniques and evaluation criteria can arise - will be explored.

The rest of this case study will proceed as follows. First, a brief overview of the DDV project will be followed by a summary of the relevant aspects of High Performance Fortran. A particular application will be explored, and then some observations on the value of the visualizations which were generated will be discussed. Finally, some possible areas for future work and a summary will be given.

Goals for a DDV Environment

HPF and pC++ offer data parallel extensions to Fortran and C++, respectively, allowing programmers to specify data distributions for the data structures of each language. In HPF the programmer distributes arrays only. In pC++, traditional C++ objects are the targets of distribution specifications. As a starting point, this research focuses on developing visualizations for HPF. High Performance Fortran offers a simple platform from which to begin this work since only arrays need to be considered. With that in mind, two primary goals can be established.

First, metrics that will help the programmer determine the effectiveness of a given data distribution must be identified. Once this occurs, visualizations showing some or all of those metrics for a given distribution can be developed. Second, these visualizations should demonstrate some consistency with the programmer's mental model of the data structure, the distribution, or other aspects of the problem. This is a critical goal which deserves further explanation.

One of the primary features of HPF is that it offers an abstraction far removed from any particular architecture. Thus, the specific architectural knowledge required by the HPF programmer is far less than that required for other parallel environments. The consequence of this is that traditional performance visualizations based on physical processors and low-level message passing, for example, may not be as meaningful to the HPF programmer. Thus, visualizations for HPF must demonstrate levels of abstraction that are consistent with the language's - and hopefully the programmer's - underlying conceptual model.

High Performance Fortran

High Performance Fortran is a parallel Fortran language that offers support for high performance on a wide variety of parallel processing architectures, including massively parallel SIMD machines (*e.g.*, Maspar's MP-1 or Thinking Machine's CM-2), distributed- and shared-memory MIMD architectures (*e.g.*, the Intel Paragon and the Cray Y-MP C90), vector processors, and other architectures. Version 1.0 of the language specification [11] was released in May, 1993, by the HPF Forum, a coalition of industrial and academic groups representing most commercial vendors, several government labs, and many university research groups.

HPF is a set of extensions and modifications to Fortran 90 [15] to offer data parallelism in the form of explicit parallel loop constructs, code tuning for various parallel architectures through a set of extrinsic procedures, and high performance on MIMD and SIMD machines through data distribution and alignment capabilities which the user specifies with compiler directives.

The Programming Model

While there are many features of HPF which make it an interesting topic of discussion, this study is concerned only with its data distribution features. It is well known that modern parallel architectures achieve their best performance when data accesses exhibit high locality of reference. A programmer should, therefore, try to limit the frequency with which a processor must obtain data from other processors. Data distributions hold the key to achieving this goal. HPF allows the programmer to specify aspects of the distribution process. The primary directives and their functions are given in Figure 16.

HPF Directive	*Function*
`PROCESSORS`	Declare an *n*-dimensional mesh of virtual processors
`ALIGN`	Specify relationships between multiple arrays
`DISTRIBUTE`	Specify how arrays (or groups of arrays) are distributed across virtual processors

Figure 16. The primary compiler directives in HPF allow users to create virtual processors, align data structures, and distribute data across virtual processors.

The *PROCESSORS* directive allows the user to declare an abstract set of processors onto which distributed arrays are mapped. The abstract processor set takes the form of an *n*-dimensional mesh with no inherent interconnection network. Next, relationships between different data structures are specified with the *ALIGN* directive. An *ALIGN*ment offers a common structure to which several arrays are oriented. Finally, using the *DISTRIBUTE* directive, *ALIGN*ments (*i.e.*, groups of related arrays) are mapped onto the abstract set of processors. Thus, the HPF programming model can be illustrated as in Figure 17 [11].

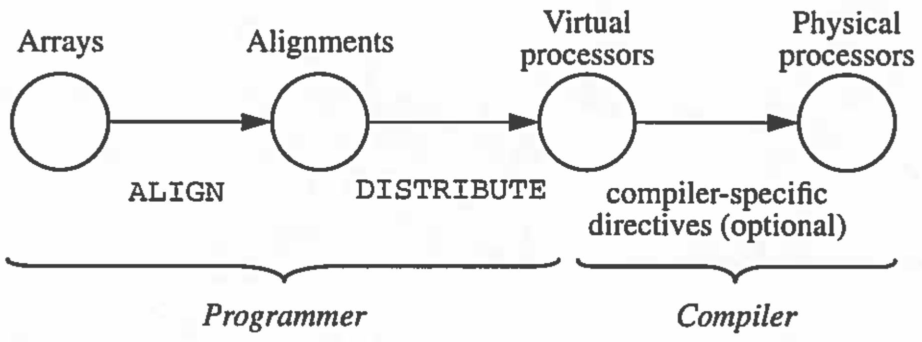


Figure 17. In HPF, programmers specify how data structures are distributed across a set of virtual processors, but the compiler is responsible for mapping virtual processors to the physical processing units of the computer.

From the programmer's perspective, this is a two-level mapping: first, data structures are aligned with one another, and then these groups of data structures are distributed onto a set of abstract processors. The compiler takes responsibility for mapping the abstract processor set onto the physical processors. (The HPF Forum suggests that commercial HPF compilers may wish to offer compiler-specific directives so that the user has some say in how abstract processors are mapped to physical processors.) From this, the underlying programming model becomes evident. Programs are written as if running on an arbitrarily sized, *n*-dimensional mesh of processors. This is consistent with the ideal data-parallel programming paradigm where the amount and type of data determines the processing resources needed.

The HPF model is dependent on two assumptions that are reasonable for current architectures. First, an operation involving multiple data elements will be faster if the elements are all on the same processor. Second, many such operations can be done simultaneously if they can be performed on different processors.

Computations in HPF

The final aspect of HPF which must be addressed is the manner in which computations are carried out. Bozkus *et al.* [2] propose four possible models of computation based on the locations of left-hand and right-hand data operands. Given the generalized form of an assignment statement, *A: lhs = rhs*, Figure 18 illustrates four possible cases.

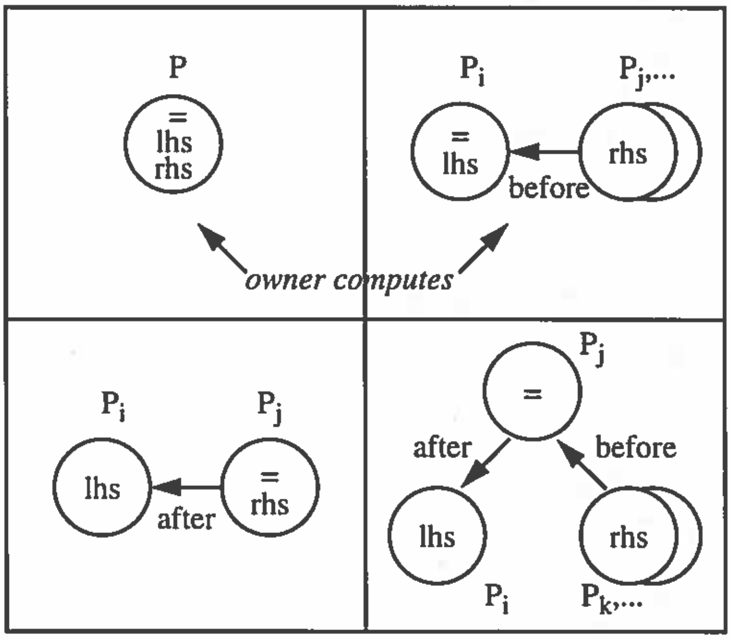


Figure 18. On a distributed-memory parallel computer, an assignment statement can be carried out using four techniques.

P_i denotes processor i. The processor containing "lhs" owns the data element on the left-hand side of the assignment; the processor(s) with "rhs" owns data elements which are required by the computation; the processor containing the "=" is the processor which performs the computation. Annotated arrows indicate the interprocessor communication required with respect to when the computation is performed (*i.e.*, before or after).

Given a data distribution, the processor responsible for performing a computation is not specified by HPF; that is, it is up to the compiler to determine which processor will perform a given computation. As noted in Figure 18, one of the more popular parallel computation models is known as *owner computes*. Under this scheme, the processor which "owns" the left-hand side data element (*i.e.*, the data element to which the assignment is being made) is the one that performs the computation. Thus, before the computation can occur, all data elements which occur on the right-hand side of the assignment and reside on remote processors must be copied into the owner processor. A degenerate case of owner computes occurs when all data elements are locally owned. Thus, the top half of Figure 18 illustrates cases of the owner computes rule.

Since the manner in which assignments and computations are carried out is compiler-dependent, this work will generally assume that the owner computes rule is used. This assumption is evident only in the sample application described later.

Establishing the DDV Problem Domain

As shown in Figure 19, establishing a problem domain for data distribution visualization consists of deciding which HPF features will be addressed by the visualizations, how data distributions are to be evaluated, and what properties the visualizations should have. The following sections will identify and justify the decisions which were made.

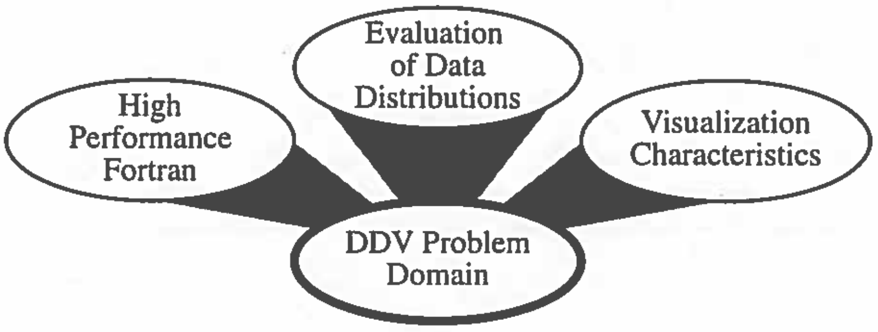

Evaluation of Data Distributions
High Performance Fortran
Visualization Characteristics
DDV Problem Domain

Figure 19. The DDV problem consists of three main issues.

High Performance Fortran

For this case study, it is important that the problem be accessible. For this reason, visualizations are limited to two-dimensional data arrays and two-dimensional sets of processors. This results in simple, obvious mappings from data structures and processor sets to graphical objects. (Similarly, an upper-bound of three dimensions was set in [20].)

In addition, data structure alignments are not considered. The current visualization system considers only individual arrays. Thus, relationships to other arrays, specified with the *ALIGN* directive, are currently irrelevant, though such relationships could undoubtedly play a role in future performance visualizations.

In HPF, programmers can dynamically redistribute arrays in the middle of their program. To maintain simplicity, DDV considers static distributions only.

Finally, as mentioned earlier, the owner computes rule will be assumed as the computational model for the sample application.

Evaluation of Data Distributions

The evaluation of data distributions is essentially a load balancing problem. The programmer wants to make sure that data structures are distributed in such a way that no single processor is excessively burdened with data requests. In accordance with the assumptions underlying HPF, the programmer also wants to maximize local data accesses and minimize the number of remote data operations that a given processor must perform. To evaluate a data distribution for a given application, the programmer needs to formulate answers to two questions:

- What memory reference patterns are exhibited by the application?
- Given the data distribution, how do these reference patterns manifest themselves on the set of processors?

Thus, it stands to reason that the type of accesses being made to data structures (*i.e.*, reads or writes) and the type of communication associated with that operation (*i.e.*, local or remote) will play an important role in determining whether a given distribution is good or bad. If visualizations can be devised that show reference patterns for both the data structure and the processors across which the data is distributed, then the HPF programmer can answer the questions above. To this end, DDV focuses on collecting information about data reference patterns, and then determines which processor has a given data element under the specified data distribution. Note that "processor," unless otherwise stated, refers to "virtual" processor during the rest of this chapter. Since HPF will potentially run on many different architectures, visualizations based on virtual processors are not architecture-specific.

Visualization Characteristics

One of the primary goals set out earlier in this thesis was that visualizations should incorporate the programmer's mental model. This applies to both data structures and processors. The limitations applied to HPF described above make this task much easier. By placing an upper-bound of two dimensions on arrays and processors, visualizations should be able to include easily the traditional representation of arrays. The creators of the visual programming tool for Fortran D [20] point out the importance that should be placed on maintaining consistency between graphical portrayals of arrays and the programmer's mental model of arrays. They indicate that a grid of cells arranged in rows and columns is the most obvious and intuitive representation of such structures.

In order to answer fully the questions posed above, visualizations should incorporate the passage of time. In general, this means that there should be some component of animation in the visualizations. (Although, time can be incorporated into visualizations in other ways, as was done in Chapter VII.)

Finally, the ability to show multiple dimensions of the problem in a single visualization may be useful in some circumstances. At the very least, this should be an option for the programmer.

Given the establishment of the DDV problem domain, the problem at hand is well-defined and approachable. We now turn to a discussion of the experimental environment which was developed within the parallel program and performance visualization methodology that is the topic of this thesis.

An Experimental DDV Environment

In the following sections, a detailed examination of the DDV environment will be presented. Figure 20 offers an overview of the steps from application to visualization.

Creating Memory Reference Profiles

As with the visual programming tool for Fortran D [20], DDV encourages the programmer to experiment with different distributions for a given application. As shown in Figure 20, the first step in this process is to create a memory reference profile for an application. The current environment does not actually handle parallel programs since no HPF compiler was readily available at the time of this work; rather, a sequential application is

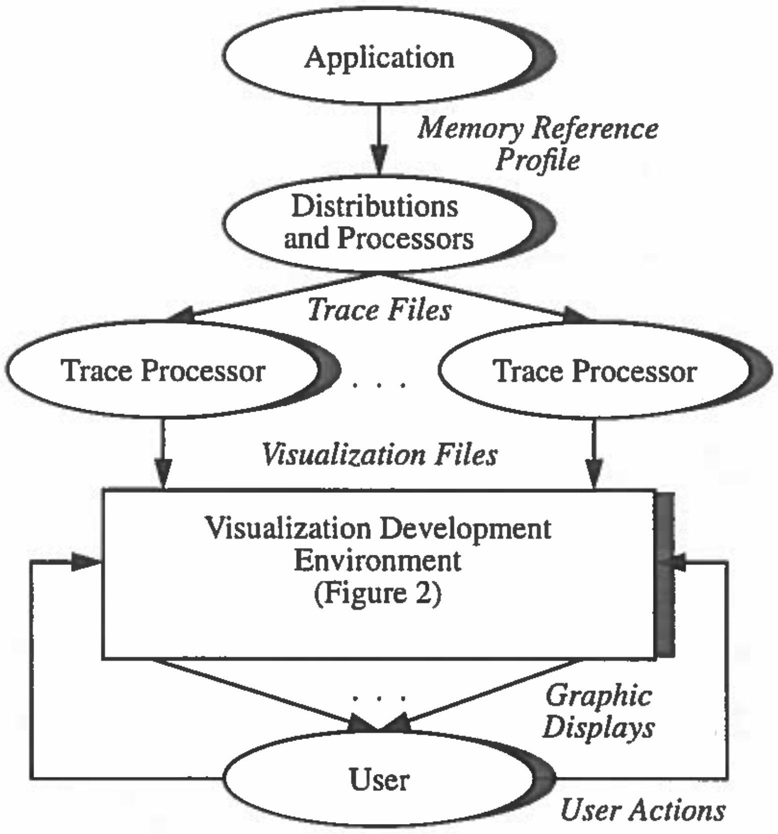


Figure 20. The DDV environment incorporates the implementation environment of Figure 2 by adding capabilities for collecting and processing trace and performance data.

used to mimic a parallel one, and the owner computes rule is imitated by calls to a *TRACE* function (see below). At desired points in the code, function calls are inserted to generate profiling information about the particular memory operation(s) being performed and the data elements being referenced. To mimic owner computes, the profiling information is made relative to some other data element - in particular, the element appearing on the right hand side of the assignment. This process can be accomplished automatically with current program transformation technology.

While the profile may not actually represent the parallel execution, it identifies the reference patterns that an application exhibits. Better yet, the reference profile is independent of any particular data distribution. Thus, a single profile can lead to the creation of any number of trace files which are distribution-specific. If a real parallel application was to be used, then the *TRACE* function could be easily modified to generate distribution-specific information, and the memory reference profile would be unnecessary.

The *TRACE* function accepts a time stamp, the location of the "owner" element, a range of referenced data elements, and the operation (read or write) performed. (Note that whether the operation required local or remote communication is unknown until a distribution is specified in later steps.)

Building Trace Files

Once a memory reference profile has been established, any number of trace files can be generated. Each trace file is specific to a single data distribution and processor arrangement. Thus, the process of building a trace file requires a filter program that understands HPF's data distribution functions. The HPF language specification [11] offers formulaic definitions of the distribution options discussed earlier in this paper. These formulas provide the mapping from data elements to processors in a single dimension.

First, define *c(j,k)*, the ceiling division function, by $c(j, k) = \left\lfloor \frac{j+k-1}{k} \right\rfloor$. Let d represent the size of the data array in a certain dimension, r, and let p be the size of the corresponding dimension in the processor array. (Assume all dimensions have a lower bound of 1.) Then, the specification *BLOCK(m)* means that data element j of dimension r will be mapped to the abstract processor at location $c(j,m)$, provided that $mp \geq d$. Similarly, *CYCLIC(m)* means that the data element in location j of dimension r will be owned by processor $1+((c(j,m)-1) \bmod p)$. These relationships are used to transform a memory reference profile into a trace file once a data distribution has been specified. The trace file allows for multiple operations in a given time step. In this fashion, the annotation of a sequential application can be used to mimic the operation of the parallel application.

Creating Visualization Files

The next step to data distribution visualizations is to transform the trace file into a visualization file. From the information in the trace files discussed above, this step keeps track of running totals and averages of local reads, remote reads, local writes, and remote writes for each element in the data array and the processor set. This information is present for each time step in the trace file. Thus, the totals and averages are computed over the time that has expired up to that point in the trace. In this way, an animation of the application's memory references may be constructed in the visualization environment.

Building a Visualization Framework

The final step in the visualization process is to use the data visualization software to create an environment for exploring the data distribution visualization. As mentioned in Chapter VI, Data Explorer allows the user to create control panels. The one in Figure 5 came from the DDV environment.

The visualizations display two views (cumulative and average) of a given trace quantity. One statistic is mapped to a colored background on the grid, and the other is mapped to "glyphs" which float in front of the grid. (The graphic displays will be discussed in more detail later.) The user can control several aspects of the display:

- The trace type to view (data or processor)
- The dataset to view (the user can easily add additional traces to the control panel)
- The quantity to be displayed display (*e.g.*, local reads, all remote accesses, all reads and writes, etc.)
- The statistics to be displayed (cumulative and/or average)

The user can also control the animation with the "Sequence Control" box. The controls are simple and intuitive, similar to those used in audio cassette and compact disc players.

In the actual visualizations, quantities are represented by the size and color of the glyphs, and by the color of the background grid coloring. (Note that glyphs contain redundant information. That is, the size and color of the glyph represent the same quantity.)

Application: Gaussian Elimination

To demonstrate the effectiveness of these visualizations, a sample application will be presented here. As was mentioned earlier, the current DDV environment is set up to operate with sequential programs. By carefully annotating this code, a pseudo-parallel trace file can be generated.

The Data Distribution

In the following, visualizations built from a Gaussian elimination algorithm will be presented. The algorithm operates on a 10x10 data array which is distributed on a 2x8 grid of processors in a cyclic fashion (for both rows and columns). The quantity being displayed in the images consists of all memory accesses (that is, local and remote, reads and writes).

Let us consider the distribution of data which results from the distribution and processor arrangement given above. Figure 21 specifies the processor which owns each element of the 10x10 array by assigning each processor a unique number. It is important to note that processors 0, 1, 8, and 9 own many more elements than the other processors because columns are distributed in a cyclic manner.

Source Code Instrumentation

A fragment of code from the Gaussian elimination algorithm used to generate the visualization presented in the following sections appears in Figure 22.

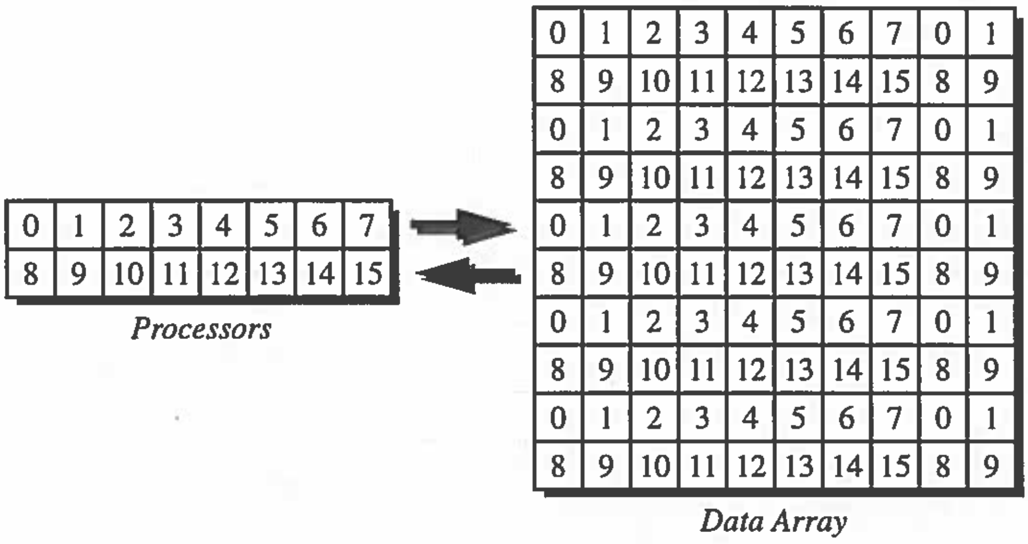


Figure 21. The cyclic distribution of a 10x10 array onto a 2x8 grid of processors leaves some processors more heavily loaded with data elements.

```
c = 1.0/B[j][j];
   for (col=0; col<MAX; col++) {
      /* scalar mult. the elements of row */
      B[j][col] = c*B[j][col];

      /* each owner read a copy of B[j,j] */
      TRACE(Time, j,col, j,j, j,j, 0);
      /* read own copy of B[j,col] */
      TRACE(Time, j,col, j,j, col,col, 0);
      /* write B[j,col] */
      TRACE(Time, j,col, j,j, col,col, 1);

      /* do the same to the augmentation */
      M[j][col] = c*M[j][col];
}
Time++;
```

Figure 22. An instrumented code fragment shows how the sequential algorithm is used to imitate a possible parallel implementation.

The piece of code illustrates the transformation that takes place during the instrumentation process. The loop is one part of the Gaussian elimination algorithm. In this example, each row is being scaled by the reciprocal of the element on the main diagonal.

While the sequential algorithm makes an assignment to the variable *c*, the calls to *TRACE* mimic what might actually happen in a parallel version of the algorithm by having each of the "owners" - referenced by the data element at *(j,col)* - request copies of that value. Note also that the *Time* variable is not incremented until after the loop, suggesting that the operations in the for-loop could be done simultaneously in a parallel implementation.

Display Generation

The instrumentation is compiled along with the source code, and a file containing a stream of memory references augmented with some additional information results after the application is run. This memory reference profile is independent of any data distribution, and it represents the pattern of memory accesses that this implementation of Gaussian elimination displays under a loose interpretation of the owner computes rule. The profile is used to create several different traces - one for each distribution and processor set that the user wants to evaluate. The trace files are then transformed into visualization files and imported into Data Explorer where the user can control many aspects of the resulting animation, as explained in the earlier parts of this thesis.

Visualizations

At this point, we are finally prepared to present a sequence of actual visualizations which resulted from the Gaussian elimination program.

Figure 23 illustrates the data array early in the execution of the Gaussian elimination algorithm (time 20). The background represents the average number of memory references, while the glyphs (size and color) keep track of the cumulative number of references. With element (0,0) in the upper left corner, the access patterns of this phase of the algorithm are already evident. As the algorithm proceeds, the glyphs in each column below the main diagonal grow larger (and change color) as the column elements below the diagonal are zeroed-out, and the row elements are adjusted accordingly.

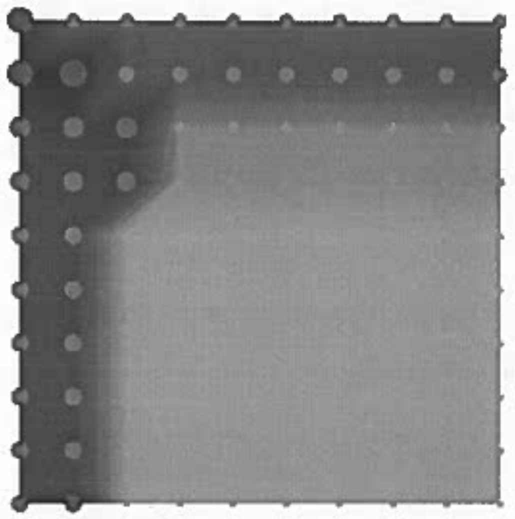


Figure 23. (Time 20) Data access patterns are already evident early in the algorithm.

Figure 24 displays the state of the processors at the same time as Figure 23. It can be seen that the size of the glyphs on the left side of the grid are slightly larger than the others, however, it is too early to tell if this is due to a poor distribution or the particular phase of the algorithm.

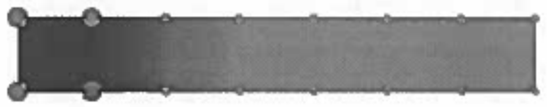


Figure 24. (Time 20) The processors do not yet appear too unbalanced.

Figure 25 shows the data array at time 60 of the algorithm. The first phase of the Gaussian elimination has been completed, and the second phase is just beginning. Thus, all of the elements on the main diagonal are now 1, and all elements below the diagonal are 0. The algorithm is proceeding to zero-out all matrix elements above the main diagonal by working its way back through the columns in reverse order, toward the origin of the array. Figure 26 shows a different quantity being displayed on the data array. Rather than showing all memory references, Figure 26 shows only remote writes, the pattern of which is considerably different than the overall memory reference pattern illustrated by Figure 25.

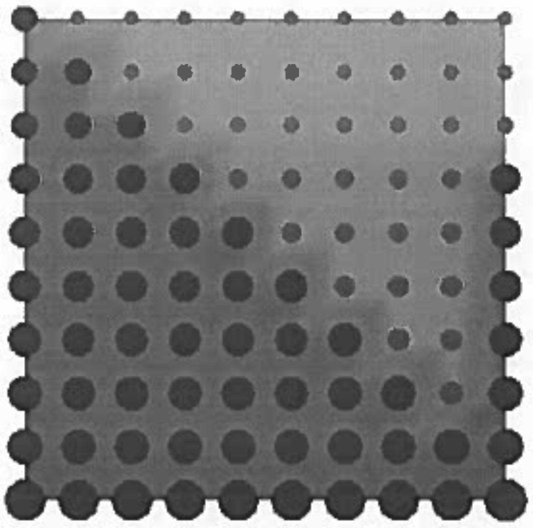


Figure 25. (Time 60) The visualization reflects the beginning of the second phase of the Gaussian Elimination algorithm.

Figure 27 again shows the status of the processor grid. The imbalance generated by the cyclic distribution is becoming more and more evident at this point in the animation. The other processors appear to be fairly well-balanced, though.

By time 89, as seen in Figure 28, the algorithm is nearing completion with only two columns remaining to be processed. It is evident from this display that overall, the Gaussian elimination algorithm accesses the array in a fairly uniform manner. That is, there are no elements that receive excessive attention in the long run. However, the earlier frames of the animation clearly show that the algorithm progressed in certain phases.

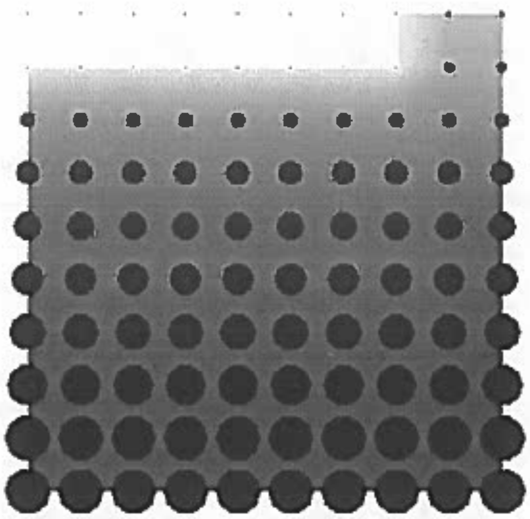


Figure 26. (Time 60) The distribution of remote writes is considerably different than the overall distribution of all memory accesses.

Finally, Figure 29 illustrates the sorry state of the processor grid by the end of the algorithm. The load imbalance introduced by the cyclic distribution has clearly overtaxed the processors in the two left columns of the grid.

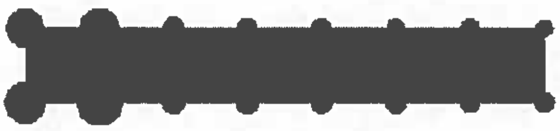


Figure 27. (Time 60) A growing imbalance in processor load becomes evident.

Some conclusions regarding the effectiveness of a cyclic distribution in this case have already been made. Using what has been learned and the DDV tools discussed earlier, the programmer could now go back and create a new trace file using a different distribution or a different processor arrangement. In this way, experimenting with different distributions is simplified and the programmer undoubtedly gains insight not only to the algorithm, but to which distributions are most effective for it.

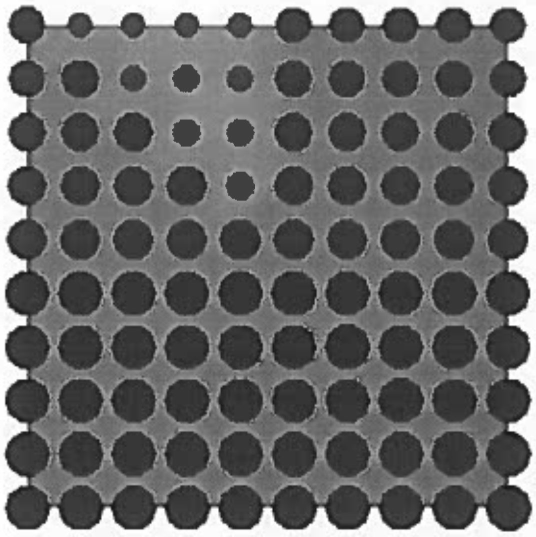


Figure 28. (Time 89) The algorithm nears completion, and the display seems to suggest a fairly uniform access pattern over the life of the algorithm.

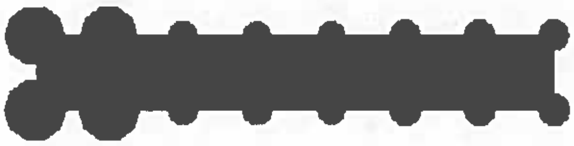


Figure 29. (Time 89) Processors in the left two columns of the grid have experienced significantly more memory accesses than the others.

Observations

In this section, some informal observations on the effectiveness and utility of the DDV environment, as implemented within the more general parallel program and performance visualization methods discussed earlier, will be made. This will take place in three categories: usefulness, understandability, and scalability.

Usefulness

As was shown in the previous section, the visualizations generated from the DDV environment provided a good overview of memory reference patterns exhibited by the application. Depending on the quantity that the programmer is viewing, a sense of the interprocessor communication required is easily achieved. By offering the programmer visualizations of both the data structure and the processor set holding the data, one can effectively assess many characteristics of a given data distribution.

While reference patterns can be extracted directly from the animated displays, the size of messages being passed between processors remains obscured. Similarly, contention for data elements can not be seen. While this study has not focused on these metrics, they are still of potential interest to programmers.

Understandability

Since the problem domain limited the data structures and processor arrangements to two dimensions, these displays were able to take advantage of the obvious way to visualize such structures (*i.e.*, by using a grid). For this reason, the displays are fairly simple to understand. The images shown here clearly lack labelling that would be present in a more complete environment, though. For example, the actual DDV displays contain captions and color bars for each quantity being measured.

It was mentioned earlier that the glyphs contain redundant information by having the size and color controlled by the same quantity. This could have both positive and negative effects. To novice users, it may reinforce the magnitude of the particular quantity. This technique is called "double cueing." More advanced users might find double cueing redundant and would prefer having color and size each represent a unique metric. The visualization environment is flexible enough to handle both cases.

Scalability

Finally, a very important topic in performance visualization is the scalability of the displays, especially in the context of data-parallel programming. Certain aspects of these visualizations are scalable while others are not. The glyphs, for example, do not offer a high degree of scalability. While very effective for small grids, the information is lost when grid size goes much above twenty in one dimension. In particular, the variance in the size of the glyph becomes so minute that little or no information is conveyed and visual complexity becomes excessive. Thus, glyphs may be more appropriate for processor sets since they tend to be smaller than data arrays.

The background grid coloring represents a more scalable visualization technique, however. By letting the color of the grid at each data element correspond to the quantity being measured, a global picture of memory references can be achieved for both small and large grids. In fact, the display actually seems to improve as the grid grows larger, up to a certain point. By letting the appropriate quantities color the grid background, the "hot" and "cool" spots of the data structure emerge. Also, data visualization software offers zooming capabilities which allow the user to focus on particular regions of the data or processor structure. The issue of scalable visualizations will be explored in detail in the final case study in Chapter IX.

Summary

The work presented here can be extended in several ways, especially since a very limited problem domain was pursued for this case study. The visualizations presented in this case study have as a primary goal the maintenance of consistency with the programmer's conceptual model. While these visualizations do not make full use of the data visualization software's graphical capabilities, the DDV environment, as a whole, does take advantage of the customization and interactive features of the software. Other visualization techniques that preserve consistency and make use of more sophisticated visualization techniques are also possible. Figure 15 presented in Chapter VII, for example, uses a rubber sheet in which grid points are displaced a distance proportional to some performance metric.

Still other visualizations may choose to deviate from the consistency goal to reveal relationships that are not evident in a simple two-dimensional grid. It is still important that the programmer have access to displays like the ones presented here, but there could be much to gain from advanced displays which are not necessarily formulated from a programmer's conceptual view. Such extensions to data distribution displays are pursued in Chapter IX which deals with scalable performance visualizations for data-parallel programs in general.

The important role played by data distributions in determining the efficiency of parallel applications demands that tools which assist in evaluating data distributions be developed. This case study has developed an experimental environment that explores the use of visualization for the evaluation of data distributions. It has been demonstrated that visualization techniques can be developed that will benefit the programmer responsible for writing efficient HPF programs.

CHAPTER IX

CASE STUDY: SCALABLE PERFORMANCE VISUALIZATIONS

The case study in Chapter VII explored the use of sophisticated graphical techniques in the general context of parallel programs and performance visualization. Chapter VIII explored the development of visualizations for the sole purpose of evaluating data distributions within the context of the data-parallel High Performance Fortran language, but the displays were not particularly "advanced" in their graphical representation. Rather, the environment took advantage of the customization and interaction features of the data visualization software environment.

In many ways, the case study in this chapter is an integration of the two previous studies. At the end of Chapter VIII, the notion of visualization scalability was briefly discussed in the context of the DDV displays. For this study, scalability is a primary concern in that it explores how advanced graphical techniques can be used to create scalable performance visualizations for data-parallel programs. In many cases, the same type of performance data and visualization concepts used in Chapter VII will be used here.

Introduction

In this study, attention is directed toward the problem of designing visualizations of parallel performance information for scalable, data-parallel programs. To this end, this study builds upon Couch's work in scalable execution visualization [4], and Rover's work in performance visualization of SPMD and data-parallel programs [40], to propose techniques that may be effectively applied in future parallel program analysis tools. These techniques are demonstrated through visualizations that have been produced for the Data-parallel C [8] and pC++ [26] language systems using the visualization environment presented in the earlier chapters of this thesis. In this study, a categorization of four scalable visualization techniques for data-parallel programs is presented. These techniques are then demonstrate with several examples.

Scalable Visualization Methods

Data-parallel programming is a well-accepted approach to develop programs that can scale in their performance. However, as was seen in the DDV environment of Chapter VIII, to achieve scalable performance, users must be concerned about how the data distribution across the processors of a system affects the load balance of the computation and the overhead of processor interactions. The data-parallel programming model and its execution model instantiation on parallel machines provides a rich semantic context for program analysis tools. Visualizations can be an effective complement to program analysis if the problems of display scalability can be overcome.

Through this study, four general methods that can be used to achieve visual scalability have been observed. Some have been introduced by other researchers, while others are new visualization techniques. In either case, the integration of these methods with sophisticated graphics represents a significant advance in parallel performance visualization, in particular as it applies to data-parallel program analysis. The following four categories represent a diverse set of visualization techniques that have been used to create scalable visualizations:

- adaptive graphical representation,
- reduction and filtering,
- spatial organization, and
- generalized scrolling.

This classification is certainly not exhaustive. Rather, this list represents a set of generalizations that have been deduced from several displays developed during the course of this research effort. These methods are best illustrated by the examples in the following sections, however, general descriptions of each will be given here.

Adaptive Graphical Representation

A series of visualizations that achieves scalability by adaptive graphical representation accomplishes this by changing the graphical characteristics of the display in response to the size of the dataset. This transformation of the graphical mechanism is performed within the bounds of the following premise:

> Given a fixed level of detail that is to be portrayed in a visualization, a view of a visualization should (1) reveal as large a quantity of the detail to be represented in the display as possible, and (2) prevent visual complexity from interfering with the perception of that detail.

Thus, a visualization is developed around a concept that can be represented by several different graphical representations. An appropriate technique is chosen that is consistent with the visualization concept and the two properties above. It is important to note that these criteria oppose one another since level of detail and visual complexity are directly related in many displays. That is to say that the revelation of more detail generally leads to a higher degree of visual complexity, and less visual complexity usually implies that less detail can be shown. The balance between the two is subjective and dependent on many factors such as the visualization concept being used and the viewer's preference.

Reduction and Filtering

Reduction and filtering are established techniques for achieving scalability. Traditional reduction methods perform statistical operations at the raw data level, such as computing the sum, mean, standard deviation, and frequency distribution. Using these methods, a smaller dataset containing some essential, summarized, or less detailed information is presented in place of the original data. This work extends this notion to include *graphical* reduction, which includes operations that help to reduce the complexity of the visualization at the level of graphical representation, rather than at the level of the raw data.

Spatial Arrangement

Spatial arrangement can be used to produce scalable visualizations by arranging graphical elements so that as a dataset scales, the display size and/or complexity increase at a much slower rate. In many instances, the spatial arrangement of data in a visualization simply provides a framework in which other graphical techniques (*e.g.*, coloring, annotation) attempt to portray the characteristics of the performance data. A special type of spatial arrangement, *shape construction*, is a technique in which one defines the properties of a three-dimensional structure by the characteristics of the performance data to be visualized. The shape of the resulting object captures the combined characteristics of the performance data. In general, the spatial arrangement of a dataset is not the component of the display that conveys the characteristics of that dataset. Rather, as mentioned above, some additional graphical technique is employed within that organization to accomplish that task. In a visualization using shape construction, the spatial arrangement of the data does play a key role in conveying the characteristics of the performance data. Other features of the "shape" may assist in this function as well (*e.g.*, color, surface properties).

Generalized Scrolling

Finally, generalized scrolling refers to the use of a variety of methods to present a continuous, localized view of a much larger mass of information. Localization allows a greater level of detail to be presented, while continuity allows relationships between nearby representations to be observed. As an extension to the traditional notion of scrolling which depends on *spatial* continuity, this work considers how *temporal* continuity can be used to perform scrolling. In this way, display animation is classified as a scrolling technique that utilizes temporal continuity.

Applications

This section offers several examples of data-parallel program performance analysis that demonstrate the types of visualization scalability mentioned in the previous section. Some visualizations illustrate more than one technique. All displays were created using the visualization framework described earlier in this thesis and use data distribution performance data similar to that discussed in the previous case study.

Adaptive Graphical Representation

To illustrate how visual scalability is achieved by adapting the graphical techniques used, consider the visualizations in Figures 30, 31, and 32. The data for this sequence was generated by a Dataparallel C implementation of a Gaussian elimination algorithm running on a Sequent Symmetry multiprocessor. As in the DDV environment, these displays are concerned with providing information for evaluating the interleaved data distribution as problem size scaled. To do this, the visualization concept that was initially presented in the rubber sheet example of Figure 15 was adopted. The concept uses vertical displacement from a reference plane, double-cued with color, to portray the differences in local and remote accesses to elements of the data structure. A processor performs a local access if it reads or writes data that resides in the memory of another processor, and it performs a remote memory access if it reads or writes data that resides in the memory of another processor. (As was done earlier, we view data accesses at the level of the virtual machine maintained by Dataparallel C.) In all visualizations, the structures above the reference plane represent the difference between local and remote read accesses, while the objects below the plane represent the difference between local and remote write accesses.

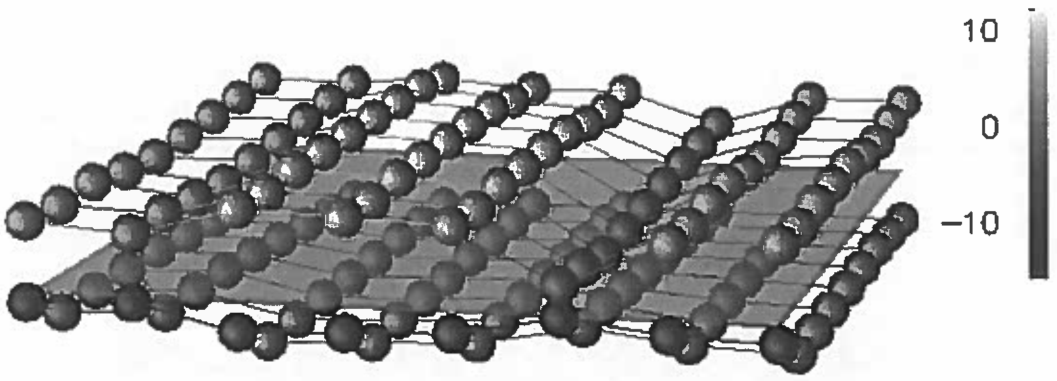


Figure 30. A small data structure (8x9) is effectively portrayed by using discrete spherical glyphs.

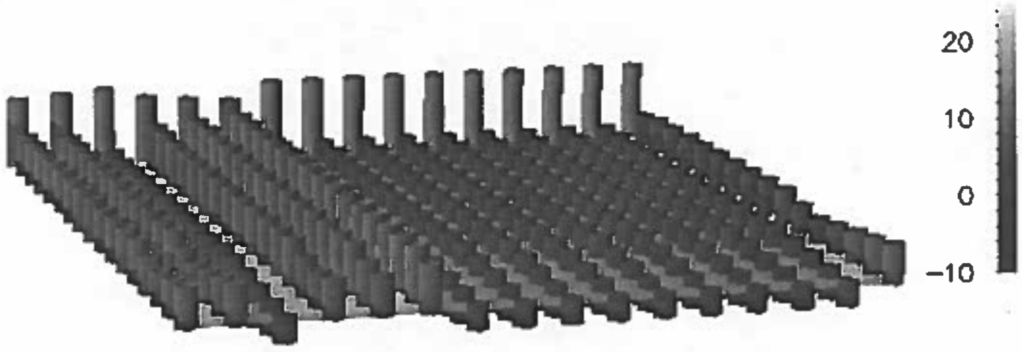


Figure 31. A medium-sized data structure (16x17) requires the vertically-displaced tops of the cylinders be connected to the plane to provide reference information.

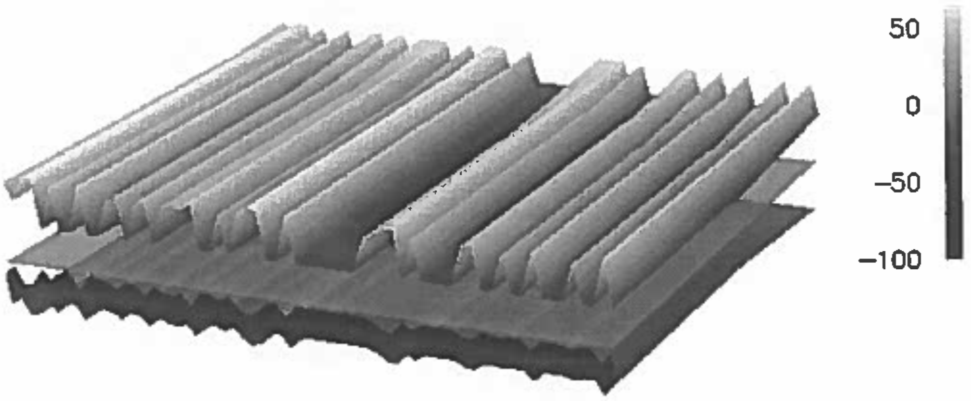


Figure 32. A continuous displacement grid minimizes visual complexity for a large data structure (64x65).

For the small 8x9 dataset in Figure 30, discrete, colored spheres floating above the reference plane minimize the obstruction of other objects in the view, while effectively representing the performance information because of the small size of the dataset. For example, elements in row 2 have encountered more remote read accesses than those in other rows. Figure 31 shows a 16x17 dataset. Spherical glyphs lack "connection" to the reference plane, and when the number of elements increases sufficiently it becomes difficult to determine which glyph corresponds to which element. The towers used in Figure 31 provide such reference information by linking the vertically displaced top of the tower with the reference plane. Finally, for the larger 64x65 dataset (Figure 32), a continuous displacement grid minimizes the visual complexity that would be caused by 4,096 discrete glyphs or towers, but it does so at the expense of the quantity of detail that is visible from a given view. (That is, with the small dataset you could simultaneously see both sets of glyphs, whereas with the large dataset this isn't possible.)

From these visualizations, the user can interpret the access frequency among local and remote reads and writes. However, detailed views are obscured as the dataset increases in size. The sequence illustrates one method of achieving scalability by adapting the graphical technique in relation to the size of the dataset. Such adaptation often manifests itself as a transition from a discrete, detail-revealing method to a continuous, complexity-reducing technique, echoing the two criteria set out in the premise for this technique. With respect to the data-parallel program, the discrete-to-continuous transition approximates the increasing detail of the parallel data structure.

Reduction and Filtering

In many instances, one need not visualize all the detail present in a dataset to gain insight into what the dataset contains. To this end, filtration and reduction can be used to develop scalable visualizations.

The visualizations in Figures 33 and 34 illustrate this technique by using isosurfaces within a three-dimensional structure. Isosurfaces, the three-dimensional analog of (two-dimensional) contour lines, represent surfaces of constant value (called the "isovalue") within the structure. In these visualizations, local and remote access information from a pC++ implementation of the random sparse conjugate gradient computation in the NAS benchmark suite is portrayed. The elements of a *BLOCK*-distributed data structure have been arranged in a three-dimensional cube. Each element has an associated number of local and remote data accesses made to it during the last time interval. The isosurfaces within the structure reveal areas of the data structure experiencing similar levels of remote (or local) accesses. By animating the visualization, the evolution of data access patterns during the program's execution is effectively revealed, allowing regions of more intense access to be identified. Figure 33 shows two time slices from a 4x4x4 cube showing remote accesses. Figure 34 is a scaled version of this display, using a 16x16x16 structure showing local accesses.

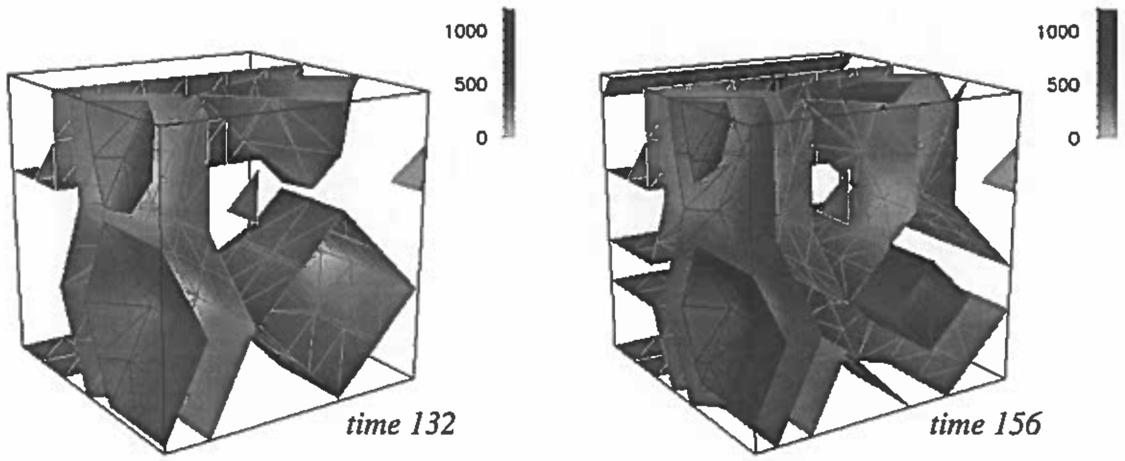


Figure 33. Isosurfaces are used to portray remote data accesses to 64 data elements arranged in a 4x4x4 grid at two different times during the application.

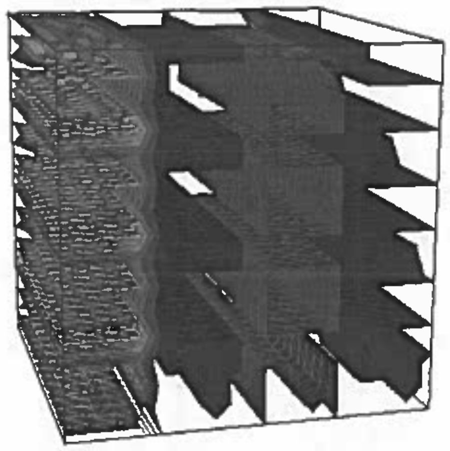


Figure 34. A scaled visualization shows local data accesses to 4,096 elements arranged in a 16x16x16 grid.

Scalability is achieved in these displays by filtering and reducing the displayed data. Isosurfaces perform an effective graphical reduction because several isovalues can be used (each figure contains five) to create multiple surfaces that span the range of the performance metric of interest and represent all elements of the structure, yet do not cause uninformative visual complexity.

Spatial Arrangement

This study has found that certain spatial arrangements of data elements can produce highly scalable visualizations. To illustrate this, three sets of figures are presented, beginning with the isosurface displays in Figures 33 and 34. It was discussed previously that the isosurfaces provide scalability by reducing and filtering the amount of data to be represented in the visualization. However, the spatial arrangement of the data structure provides another aspect of scalability. The structures in Figure 33 each represent 64 data elements organized in a 4x4x4 cube; Figure 34 represents 4,096 elements organized as a 16x16x16 cube. Whereas the problem size has scaled by a factor of 64, the axes of the cube have only scaled by a factor of 4. Clearly, the scalability of spatial dimensions is critical to the scalability of visualizations in general.

The next set of displays that demonstrates the use of spatial arrangement as a method of achieving scalability are in Figures 35 and 36. These displays implement a type of three-dimensional scatter plot that relies on the perceptive abilities of the human visual system to detect clustering and distribution patterns [4,16,48]. For this reason, this visualization gains effectiveness for larger datasets. Thus, the spatial arrangement (*i.e.*, distribution) of the data yields a technique that scales well. Figure 35 portrays the quantity and location of accesses by each processor. Processors are identified by a glyph's color, with the vertical displacement from the reference grid indicating the number of accesses made to each of the 64 data elements (arranged in an 8x8 grid) during the last time interval of the pC++ application. Figure 36 offers a prototype of a scaled version of this display on a 32x32-element structure.

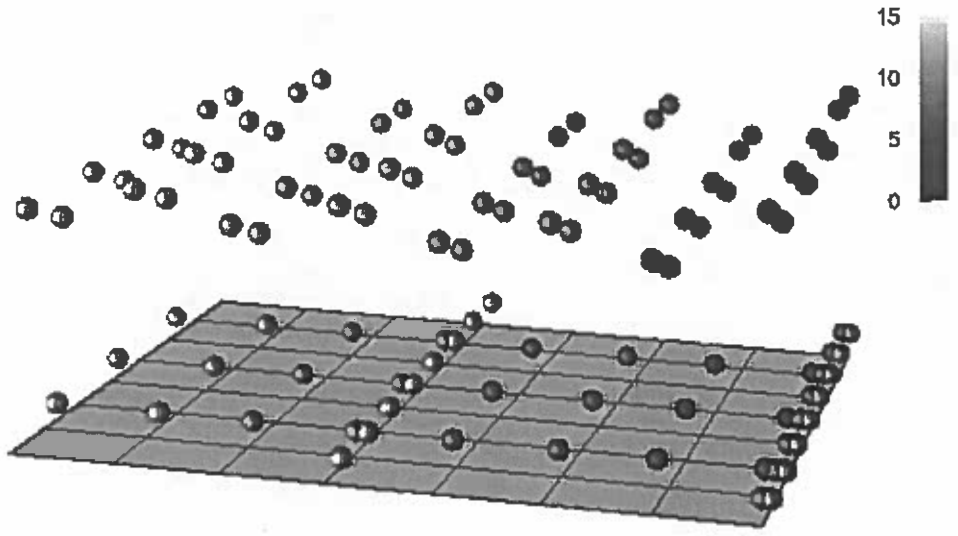


Figure 35. A three-dimensional scatter plot shows which and how often processors are accessing the elements of the data structure.

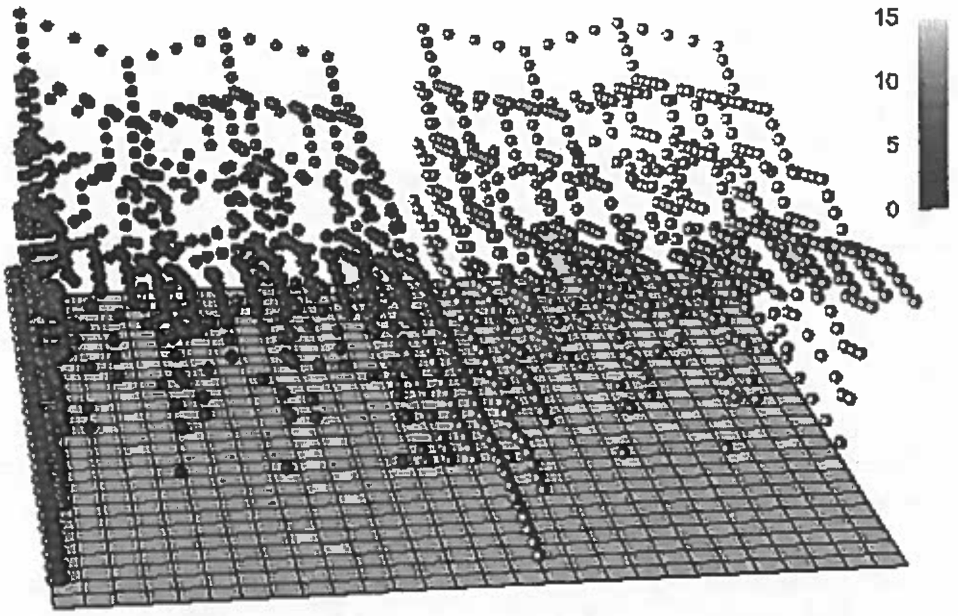


Figure 36. A prototype of a scaled scatter plot exposes global data access patterns.

From these displays, the analyst can derive several observations that are helpful in evaluating the memory access patterns of an application as well as the data distribution currently being used. For example, by using the vertical displacement visualizations (Figures 30-32) for this pC++ application, an analyst could easily learn about the distribution of local and remote data accesses. This is also seen in Figures 35 and 36 by noticing that the glyphs generally appear in two vertically separated clusters (particularly when viewed from the appropriate orientation). The display reveals how the data was distributed among the 16 processors by the color distribution. However, a differentiation between local and remote references is not made. The analyst might notice such a division first, though, and then be motivated to determine local and remote distributions from the other displays. Alternatively, within the Data Explorer environment, the user could change the color mapping so that local and remote accesses are distinguished. This example shows how user interactions can play a significant role in scalable visualizations.

That such references are actually remote is confirmed not only by the small degree of vertical displacement (indicating the lower frequency associated with remote accesses in this application), but by the presence of glyphs having colors different than the corresponding glyphs above representing local accesses.

The last set of visualizations that illustrate scalability by spatial arrangement are in Figures 37, 38, and 39. These displays are reincarnations of the popular Kiviat displays, as used in ParaGraph [10] and introduced in Chapter VII to show processor utilization. They use the same pC++ application described earlier. The construction of a single time slice of these displays is achieved by arranging the 64 data elements (as opposed to processors which was done before) in a circle. The distance a given element is from the center of the circle is directly proportional to the accumulated number of local (Figures 37 and 38) or remote (Figure 39) data accesses made to that element during the previous time interval. To construct the solid shown in Figure 37, adjacent elements in the same time slice and corresponding elements in successive time steps are connected to form quadrilateral surface patches. A Kiviat tube showing data distribution and access patterns results.

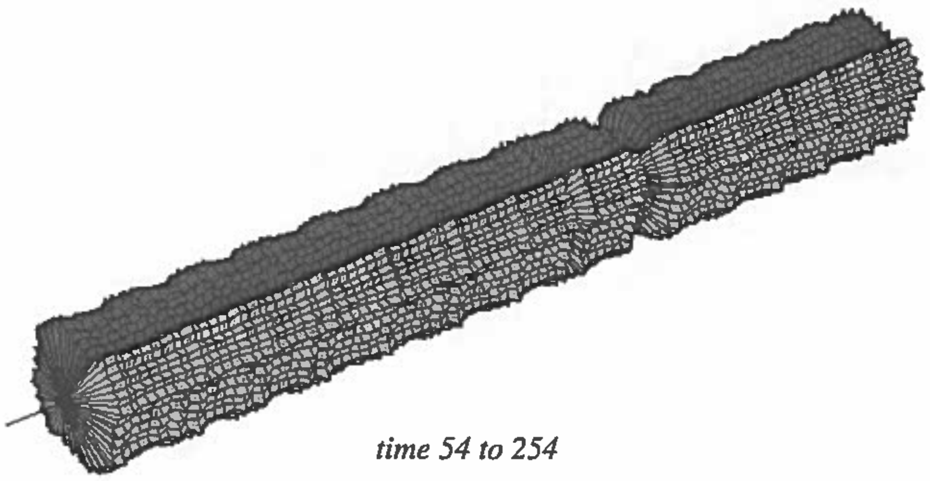
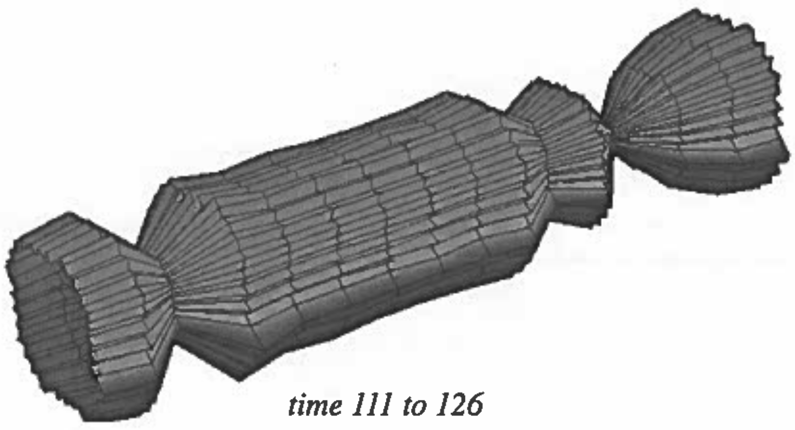


Figure 37. A Kiviat tube can be used to portray data element accesses instead of processor utilization.

The arrangement of data elements in a circle provides a moderately scalable (two-dimensional) spatial arrangement. Some additional scalability issues for this series of displays will be presented in the next section, but first, the construction of the Kiviat tube, an example of shape construction, will be discussed in more detail. The unique feature of shape constructive spatial arrangement over the spatial arrangement seen in the isosurface visualizations (Figures 33 and 34) is that the former uses the shape itself to capture the characteristics of the performance data, while the latter simply provides a framework within which some other graphical technique (*e.g.*, a set of isosurfaces) is employed to relate the performance data. Figure 37 depicts a single graphical object that captures the characteristics of an entire trace file, with time traveling along the length of the cylinder. (The initial 53 time slices have been removed because no memory accesses to the chosen data structure occurred during that time.) Such three-dimensional representations can play an important role in providing global performance information. For example, in Figure 37 one may notice a very regular access pattern for the first part of the trace, as indicated by the symmetry and constant diameter of the tube. However, approximately two-thirds of the way through the displayed trace data, a significant decrease in the number of local memory accesses occurs. Such global observations guide the program analyst to potential trouble spots in the execution of the algorithm and tend to be more perceptual rather than cognitive tasks [47]. As will be seen in the next section, the analyst can then examine that portion of the trace in more detail.

Generalized Scrolling

Scrolling is an established technique used to create scalable visualizations by representing smaller localized views of the visualization as the quantity of data to be displayed increases. Traditional scrolling provides a spatially continuous view of the display by allowing the user to “move” around the structure. This work has generalized this notion



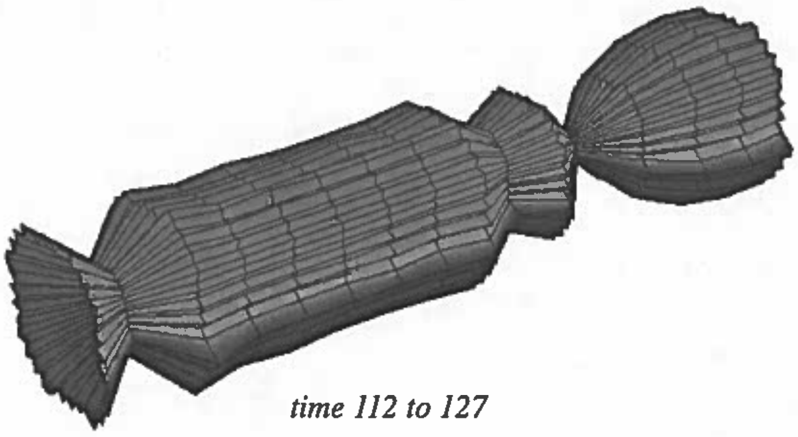


Figure 38. A blown-up region of the Kiviat tube reveals three significant decreases in local data accesses.

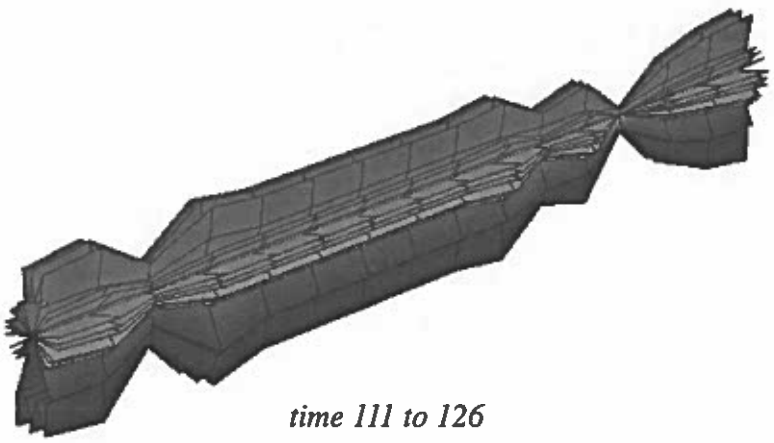


Figure 39. The corresponding Kiviat tube section showing remote accesses indicates similar decreases.

to include temporal continuity and demonstrates animation as a generalized scrolling technique.

The Kiviat tubes of Figures 37-39 illustrate both scrolling techniques. It has already been discussed how Figure 37 could guide the analyst to a particular region of the trace file. Such a region is expanded in Figures 38 and 39. The Kiviat tube visualizations allow the user to specify an animation width to display a smaller portion of the structure. Figures 38 and 39 each have animation widths of 15 time steps. The ability to zoom in on local regions of the larger structure is an example of spatial scrolling. The display's use of scrolling is, in fact, somewhat more general than most uses of scrolling since it allows the user to view just the desired section of the tube. In addition, the displayed portion may be stretched or compressed to the viewer's preference. Again, these are examples of the significant customizability attained by using the data visualization software.

To extend the notion of scrolling further, animating the structures in Figures 38 and 39 provides additional insight into the performance data's characteristics. This implementation of the visualization allows the viewer to "slide" down the length of the Kiviat cylinder at a given animation width. Figure 38 shows two successive time slices of the Kiviat tube section. This section of the Kiviat tube reveals three intervals during which the number of local memory accesses decreased significantly. Upon noticing this, the analyst may subsequently wish to examine the corresponding structure for remote memory accesses. Figure 39 shows such an alternate view that corresponds to the upper display in Figure 38. One immediately notices the significant difference in the distribution of remote memory accesses made to the elements of the data structure. In particular, the elements located on the top and bottom sides of the tube experience larger numbers of remote data accesses than the other elements. One also notices that remote accesses experience a decline similar to the local accesses during the same three intervals. All of this is potentially meaningful information for the analyst seeking to understand data distribution and performance characteristics of an application.

As has been demonstrated, generalized scrolling provides scalability by presenting a local view of the represented data, but allows global relationships to be observed by providing spatially or temporally continuous transitions from one representation of the data to another.

Summary

This case study has utilized an advanced visualization development methodology to explore the issue of display scalability, particularly as it applies to data-parallel programs. Easy access to sophisticated graphics has allowed the recognition of four categories of scalable visualization techniques that can be successfully applied to the Dataparallel C and pC++ languages. This study demonstrates how new visualization techniques can be applied to specific application contexts.

CHAPTER X

EVALUATION

In previous chapters, three in-depth case studies have demonstrated how this new visualization design process can be applied in practice. Having documented several examples, a discussion of some possible benefits and detriments of the proposed design process will be presented.

The most obvious benefits come from the minimal overhead that developers incur by using existing software products. The primary overheads involved in the process are limited to the initial cost of the software and learning the visualization environment and underlying data model. Traditional development processes have a single, overwhelming overhead: programming. But evaluating this new process must go beyond just the overheads that are encountered. In fact, if this environment is used for prototyping, but the final implementation is to be built from scratch, then both overheads are incurred. Even in this worst case scenario, there are advantages to be gained by using the proposed methods.

It was mentioned earlier that this design technology fosters iterative design and evaluation by minimizing the impact that modifications to either the data or graphical aspects of the visualization have on other parts of the visualization process. This is perhaps the most important and relevant aspect of this research to the field of performance visualization in general. Studies of performance visualizations have been few because of the effort required just to create the displays. Developers are forced to devote a majority of their time to building the visualization tool rather than testing visualization usability [28,42]. Formal evaluation is apparently sacrificed for two primary reasons. First, once a tool is completed, so much time has already gone into the project that developers can not afford to extend their work to include evaluation. Second, even if evaluations were done, modifications could impose additional significant programming costs on the project. Even those tools which market themselves with buzzwords like "modular" and "extensible" usually require considerable programming expertise [10]. Using existing data visualization software stands to refine the field of performance visualization by enabling researchers to more easily conduct usability studies and perform formal evaluations of visualizations - to determine what displays are indeed useful.

Scientific visualization packages by nature offer a high degree of user control. Thus, the ability to customize displays is built into the package. Other features like display interaction, animation, and modularity are also present. In particular, the ability to interact with what is shown on the screen will become an absolutely necessary component of next-generation visualization tools as displays become more complex and take advantage of multiple dimensions. These packages have many of the more difficult-to-program features already established and again allow visualization developers to focus on the quality of their displays rather than the code needed to generate them.

On the other hand, it is conceivable that generalized data visualization packages contain too much functionality, resulting in "environment overkill." In other words, the software may be too general, resulting in slower performance and unnecessary complexity. Experience supports this to a limited degree. The work with Data Explorer was done on an IBM RS/6000 Model 730 which required additional graphics hardware assistance and greater than 64 megabytes of memory to be acceptable. Tools built specifically for performance visualization generally do not have such stressing requirements on the computing system.

Of particular concern to performance visualization developers is the capability of a tool to handle animation (real-time or post-mortem) and dynamic display interaction.

Experience suggests that without costly hardware, such usability requirements are not adequately met by simply applying scientific visualization software. It is likely that certain limitations of this approach will be overcome by technology, but for the time being, these limitations make practical use of this technique more difficult. The hardware requirements made by scientific visualization packages are not without reason, though. These systems offer extremely flexible and powerful graphical techniques. For example, performance visualization developers would undoubtedly find the ability to compose a new display from two or more simpler displays advantageous, as was done in the Kiviat diagram examples discussed in Chapter VII. Furthermore, these products offer flexible and general data models that can be used to generate a wide range of visualizations, are designed to handle large multi-dimensional data sets, and are usually portable across many architectures.

CHAPTER XI

CONCLUSION

This thesis has presented a design process by which performance visualization developers can drastically reduce the costly and time-consuming overhead of programming, yet gain significant power and flexibility in the displays they can generate. The application of a high-level visualization methodology to general data visualization software, like IBM's Data Explorer, leads to a design technology that improves the visualization development process by increasing both the quality of the displays and the speed with which they are created. As a whole, the process lends itself to iterative design and evaluation which is required to validate a display's usability - techniques that have not been widely applied to the arena of performance visualization. Furthermore, the work being done at the University of Oregon and other research sites strongly suggests that users will demand interaction with and customization of next-generation performance displays. Existing data visualization software products are designed with these very capabilities in mind.

The work presented in this thesis can be extended in several ways. In terms of the high-level methodology developed in Chapter V, techniques that enable the specification of performance and view abstractions and their subsequent instantiations to performance and view objects are needed. This would entail automating the creation of trace and graphical transformations (*i.e.*, the creation of data object files and visual programs) from more abstract specifications. The development of performance visualization-specific modules within Data Explorer, dedicated to certain performance visualization techniques (*e.g.*, Kiviat diagrams and tubes, interprocessor communication displays, etc.), would make the trace and graphical transformations more independent, and consequently improve the implementation's consistency with the high-level methodology. Another area of interest is to exploit more of the graphical capabilities of the data visualization software and to see what other types of visualizations are possible. Finally, additional work is required to refine a methodology for evaluating visualizations in this environment.

The increasing sophistication and complexity of parallel computing environments will continue to present new challenges to understanding the operation and performance behavior of parallel programs. Visualization has the inherent capacity to facilitate parallel program evaluation only if its graphical data presentation capabilities, routinely used for scientific data analysis, can be effectively applied to reveal important properties of parallel program execution data. The most productive approach to developing useful visualizations for inclusion in parallel programming tools is to experiment with different visualization techniques in specific application contexts, such as those described in the three case studies documented earlier. The visualization methodology, model, and techniques proposed in this thesis are a significant step toward the integration of sophisticated graphical resources into the maturing field of parallel program and performance visualization.

BIBLIOGRAPHY

[1] F. Bodin, P. Beckman, D. Gannon, S. Narayana, S. Yang. *Distributed pC++: Basic Ideas for an Object Parallel Language*. University of Rennes.

[2] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M. Wu. *Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers*. Northeast Parallel Architecture Center, Syracuse University, Technical Report SCCS-444, March, 1993.

[3] M. Brown. *ICARE: Interactive Computer-Aided RGB Editor*, (A report on the work of Donna Cox, NCSA). Access newsletter, University of Illinois at Urbana-Champaign, National Center for Supercomputing Applications, 1.3.10, September-October, 1987, p. 10-12.

[4] A. Couch. *Categories and Context in Scalable Execution Visualization*. Journal of Parallel and Distributed Computing, 18, 1, June, 1993, pages 195-204.

[5] S. Hackstadt and A. Malony. *Data Distribution Visualization for Performance Evaluation*. Dept. of Computer and Information Science, Univ. of Oregon, Technical Report CIS-TR-93-21, October, 1993.

[6] S. Hackstadt and A. Malony. *Next-Generation Parallel Performance Visualization: A Prototyping Environment for Visualization Development*. To appear Proc. Parallel Architectures and Languages Europe (PARLE), Athens, Greece, July, 1994; Also, Dept. of Computer and Information Science, Univ. of Oregon, Technical Report CIS-TR-93-23, October, 1993.

[7] S. Hackstadt, A. Malony, and B. Mohr. *Scalable Performance Visualization for Data-Parallel Programs*. Proc. Scalable High Performance Computing Conference (SHPCC), Knoxville, TN, May, 1994; Also, Dept. of Computer and Information Science, Univ. of Oregon, Technical Report CIS-TR-94-09, October, 1993.

[8] P. Hatcher and M. Quinn. *Data-Parallel Programming on MIMD Computers*. The MIT Press, Cambridge, MA, 1991.

[9] M. Heath and J. Etheridge. *Recent Developments and Case Studies in Performance Visualization using ParaGraph*. Proceedings from the Workshop on Performance Measurement and Visualization of Parallel Systems, Moravany, Czechoslovakia, October, 1992.

[10] M. Heath and J. Etheridge. *Visualizing the Performance of Parallel Programs*. IEEE Software, September, 1991, pp. 29-39.

[11] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 1.0.* Rice University, May, 1993.

[12] S. Hiranandani, K. Kennedy, and C. Tseng. *Compiling Fortran D for MIMD Distributed-Memory Machine.* Communications of the ACM, Vol. 35, No. 8, August, 1992, pp. 66-80.

[13] A. Hough and J. Cuny. *Perspective Views: A Technique for Enhancing Parallel Program Visualization.* Proc. of 1991 Int'l. Conf. on Parallel Processing, August, 1991, pp. 124-132.

[14] International Business Machines Corporation. *IBM Visualization Data Explorer, User's Guide*, 2nd ed. August, 1992.

[15] ISO. *Fortran 90.* May, 1991. [ISO/IEC 1539: 1991 (E) and now ANSI X3.198-1992].

[16] P. Keller and M. Keller. *Visual Cues: Practical Data Visualization.* IEEE Press, Piscataway, NJ, 1993.

[17] D. Kimelman and G. Sang'udi. *Program Visualization by Integration of Advanced Compiler Technology with Configurable Views.* Technical report, IBM T. J. Watson Research Center, September, 1992.

[18] J. Kohn and T. Casavant. *Use of PARADISE: A Meta-Tool for Visualizing Parallel Systems.* Proc. 5th Int'l. Parallel Processing Symposium, May, 1991, pp. 561-567.

[19] K. Kolence and P. Kiviat. *Software Unit Profiles and Kiviat Figures.* ACM SIGMETRICS, Performance Evaluation Review, September, 1973, pp. 2-12.

[20] P. Kondapaneni, C. Pancake, and C. Ward. *A Visual Programming Tool for Fortran D.* December, 1992.

[21] E. Kraemer and J. Stasko. *The Visualization of Parallel Systems: An Overview.* Journal of Parallel and Distributed Computing, 18, 1, June, 1993, pp. 105-117.

[22] M. LaPolla, J. Sharnowski, and B. Cheng. *Data Parallel Program Visualizations from Formal Specifications.* Jour. of Parallel and Distributed Computing, 18, 2, June, 1993, pp. 252-257

[23] T. LeBlanc, J. Mellor-Crummey, and R. Fowler. *Analyzing Parallel Program Executions Using Multiple Views.* Jour. of Parallel and Distributed Computing, 9, 2, June, 1990, pp. 203-217.

[24] B. Lucas, G. Abram, N. Collins, D. Epstein, D. Gresh, and K. McAuliffe. *An Architecture for a Scientific Visualization System.* Proceedings from Visualization '92, Boston, MA, October, 1992, pp. 107-114.

[25] A. Malony, D. Hammerslag, and D. Jablonowski. *Traceview: A Trace Visualization Tool.* IEEE Software, September, 1991, pp. 29-38

[26] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, and F. Bodin. *Performance Analysis of pC++: A Portable Data-Parallel Programming System for Scalable Parallel Computers*. Proc. Int'l. Parallel Processing Symposium, April, 1994, pp. 75-85.

[27] A. Malony and D. Reed. *Visualizing Parallel Computer System Performance*. In Instrumentation for Future Parallel Computer Systems, M. Simmons, R. Koskela, and I. Bucher (Eds.), ACM Press, New York, NY, 1989, pp. 59-90.

[28] A. Malony and E. Tick, *Parallel Performance Visualization*. Proposal to the National Science Foundation, CISE/ASC, Grant No. ASC 9213500, February, 1992.

[29] B. Miller. *What to Draw? When to Draw? An Essay on Parallel Program Visualization*. Journal of Parallel and Distributed Computing, 18, 1, June, 1993, pp. 265-269.

[30] NCSA. *NCSA HDF, Version 2.0*. University of Illinois at Urbana-Champaign, National Center for Supercomputing Applications, February, 1989.

[31] C. Pancake. *Customizable Portrayals of Program Structure*. Proceedings from the ACM/ONR Workshop on Parallel and Distributed Debugging, San Diego, CA, May, 1993, pp. 64-74.

[32] D. Reed, R. Olson, R. Aydt, T. Madhyastha, T. Birkett, D. Jensen, B. Nazief, and B. Totty. *Scalable Performance Environments for Parallel Systems*. Proc. 6th Distributed Memory Computing Conf., April, 1991, pp. 562-569.

[33] D. Reed, R. Aydt, T. Madhyastha, R. Noe, K. Shields, and B. Schwartz. *An Overview of the Pablo Performance Analysis Environment*. University of Illinois Board of Trustees, November, 1992.

[34] S. Reiss and M. Sarkar. *Generating Abstractions for Visualization*. Brown University, Department of Computer Science, Technical Report CS-92-35, August, 1992.

[35] P. Rheingans. *Color, Change, and Control for Quantitative Data Display*. Proc. Visualization, IEEE, September, 1992, pp. 252-259.

[36] G.-C. Roman and K. Cox. *Program Visualization: The Art of Mapping Programs to Pictures*. Proc. 14th International Conf. on Software Engineering, May, 1992, pp. 412-420. Also, Washington University, Department of Computer Science, Technical Report WUCS-92-06, February, 1992.

[37] J. Roschelle. *Designing for Conversations*. Paper presented at the AAAI Symposium on Knowledge-Based Environments for Learning and Teaching, Stanford, CA, 1990.

[38] D. Rover. *A Performance Visualization Paradigm for Data Parallel Computing*. Proc. 25th Hawaii International Conference on System Sciences, mini-conference on Parallel Programming Technology, Software Technology Track, 1992.

[39] D. Rover and A. Waheed. *Multiple-Domain Analysis Methods*. Proceedings from the ACM/ONR Workshop on Parallel and Distributed Debugging, San Diego, CA, May, 1993, pages 53-63.

[40] D. Rover and C. Wright. *Visualizing the Performance of SPMD and Data-Parallel Programs*. Jour. of Parallel and Distributed Computing, 18, 2, June, 1993, pp. 129-146.

[41] S. Sarukkai and D. Gannon. *SIEVE: A Performance Debugging Environment for Parallel Programs*. Jour. of Parallel and Distributed Computing, 18, 2, June, 1993, pp. 147-168.

[42] S. Sarukkai and D. Gannon. *Performance Visualization of Parallel Programs Using SIEVE.1*. Proceedings of the 1992 ACM International Conference on Supercomputing, Washington, D.C., July, 1992, pp. 157-166.

[43] S. Sarukkai, D. Kimelman, and L. Rudolph. A *Performance Visualization Methodology for Loosely Synchronous Programs*. Jour. of Parallel and Distributed Computing, 18, 2, June, 1993, pp. 242-251.

[44] D. Socha, M. Bailey, and D. Notkin. *Voyeur: Graphical Views of Parallel Programs*. SIGPLAN Notices 24, 1, January, 1989. Also, Proc. Workshop on Parallel and Distributed Debugging, Madison, WI, May, 1988, pp. 206-215.

[45] S. Srinivas and D. Gannon. *Interactive Visualization and Animation of Parallel Programs*. Technical report (abstract), Indiana University.

[46] J. Stasko and E. Kraemer. *A Methodology for Building Application-specific Visualizations of Parallel Programs*. Jour. of Parallel and Distributed Computing, 18, 1, June, 1993, pp. 258-264

[47] J. Stasko. *Three-Dimensional Computation Visualization*. Technical Report GIT-GVU-92-20, College of Computing, Georgia Inst. of Technology, 1992.

[48] E. Tufte. *Envisioning Information*. Graphics Press, Chesire, CT, April, 1991.