Chapter 5

WWVM Data and Performance Visualization Systems

Visualization has been the cornerstone of scientific progress throughout history. Virtually all comprehension of science and technology calls on our ability to visualize. Visualization is a proven, standard technique for facilitating human comprehension of complex phenomena and large volumes of data. It can be thought of as the last step of solving computational problems or a form of assessment of the results. In fact, the ability to visualize is almost synonymous with understanding. This chapter presents the capabilities of the WWVM's Java-based implementation of the visualization engine. This engine has a data visualization component for plotting the results produced by a program and a performance visualization component for a post-mortem analysis of parallel message-passing programs. Compared to many other platform-specific data and performance visualization tools, WWVM visualization displays can be observed using personal computers in addition to Unix workstations. The only requirement is to have a

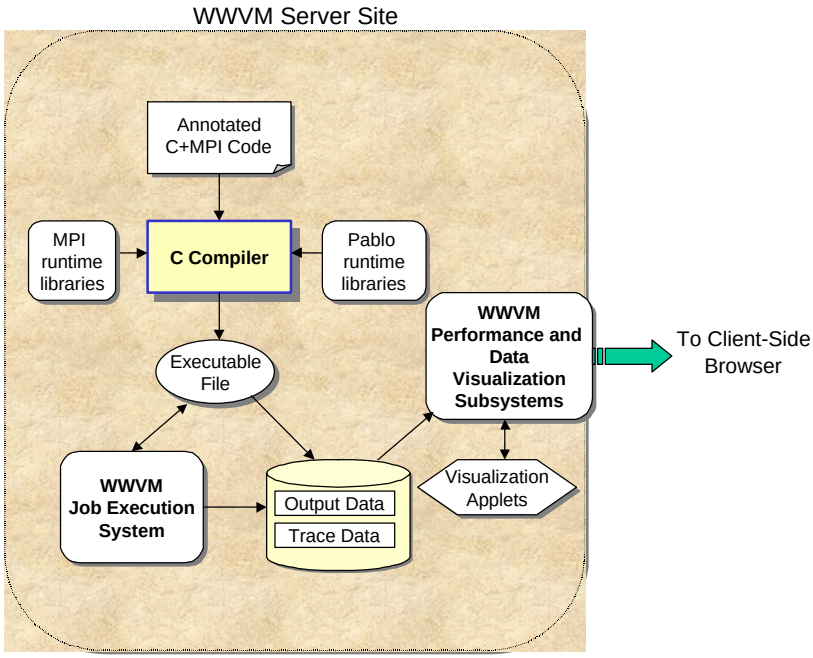

Figure 5 1. Preparing visualization traces for an MPI-based message-passing program written in C.

high-bandwidth network that is able to move large amounts of data as fast as needed. Like other parts of the WWVM user-interface, the visualization engine components are embedded into the Web-browser interface.

In addition to the specific display types, a mechanism called “data wrappers” was provided that offers the means for users’ programs written in traditional languages such as C, Fortran, or HPF to interact with Java visualization applets. This mechanism allows users to define their own visualization displays and to do interactive computation steering. Data wrappers provide a convenient way to display the data structures of users’ programs.

The aim of both performance and scientific data visualization is to gain insight into underlying phenomena by graphically depicting data. The behavior of parallel programs on

advanced computer architectures is often extremely complex, and performance monitoring of such programs can generate vast quantities of data. It therefore seems natural to use visualization techniques to gain insight into the behavior of parallel programs so that their performance can be understood and improved. On the other hand, scientific visualization is concerned with exploring data and information graphically in order to understand it. Through a combination of tools and techniques this work seeks to bring new dimensions of insight into problem solving by using current Web technology.

5.1 Performance Visualization System

Performance visualization provides insight into the factors affecting performance and provides some indication of how to improve it. The substantial effort of parallel programming is justified only if the resulting codes are adequately efficient. In this sense, all types of performance tuning are extremely important to the development of parallel software. Performance improvements are much more difficult to achieve with parallel programs than with sequential programs. One way to overcome this inherent difficulty is to bring in graphical tools. COMET [Kumar 88], IPS [MilY 87], PARAGRAPH [HeaE 91], PAWS [PGA+ 91], and TRACEVIEW [MHJ 91] are among these tools.

WWVM employs a Java-based performance visualization component that provides a detailed, dynamic, graphical animation of the behavior of message-passing parallel programs, as well as graphical summaries of their performance. This component helps to visualize execution traces (in Self-Defining Data Format - SDDF [Aydt 96]) generated from Fortran or C codes with MPI message-passing calls instrumented with Pablo trace collection (instrumentation) library calls (Figure 5-1). Pablo [RAM+ 92, Noe 96] of the University of Illinois at Urbana-Champaign is a

well-recognized performance instrumentation and analysis environment designed to organize and visualize information collected from programs executing on parallel machines.

The performance visualization system provides many different visual perspectives from which to view the same performance data. Any single visualization or perspective can only display a portion of the relevant behavior. Viewing the same underlying phenomenon from diverse perspectives gives a better-rounded impression and is more likely to yield useful sights [LCF 90].

5.1.1 Postmortem Analysis

The WWVM performance visualization component is currently used only for postmortem visualization. A majority of the performance visualization tools for distributed-memory and network computing platforms, such as Pablo, and IPS, are trace-based. These systems require an application to be recompiled, then run while the trace data is gathered for the entire application, and finally analyzed postmortem using the trace data. Following the same lines, the WWVM performance visualization component uses an SDDF trace file created during the execution of the parallel program and saved for later study.

Real-time performance visualization is generally not desirable because of three major impediments. First, it is difficult to extract performance data from the distributed-memory and networked computing platforms and to send it to the outside world during execution without significantly perturbing the application program being monitored. It should also be remembered that computation time on many state-of-the-art parallel platforms has a high cost. Second, the network bandwidth between the parallel platform and the graphical workstation, as well as the drawing speed of the workstation, is usually inadequate to handle the extremely high data transmission rates required for real-time display. Finally, even if these other limitations were not

a factor, human visual perception would be hard pressed to digest a detailed graphical depiction as it flies by in real time. Real-time visualization requires higher bandwidth, while postmortem display requires greater storage volume.

In designing the performance visualization system, the principal goals were to build a system that is easy to understand, easy to use, and portable from platform to platform. The system has an easy-to-use, interactive, mouse- and menu-oriented user interface so that the various features of the package are easily invoked and customized. Another important factor in ease of use is that the user's parallel program need not be extensively modified to obtain the data on which the visualization is based. The performance visualization system currently takes its input data from execution trace files in the SDDF format produced by Pablo, which enables the user to produce such trace data automatically. We have tried to keep the user's learning curve for JPVS very short, even at the expense of limiting the flexibility of its data processing and graphical display capabilities.

One of the weaknesses in previously built performance visualization systems is that they are dependent on a high-powered graphical UNIX workstation at the client end. On the other hand, the performance visualization component is based on the Java AWT and thus runs on a wide variety of scientific workstations and personal computers from many different vendors. It also inherits a high degree of such portability from Pablo, which runs on parallel architectures from a number of different vendors (e.g., Intel, Meiko, Ncube, and Thinking Machines). Therefore, the package is capable of displaying execution behavior from different parallel architectures and parallel programming paradigms.

The visualization component provides 16 different visual perspectives, since no single view is likely to provide full insight into the complex behavior and large volume of data associated with the execution of parallel programs. It includes modules for visualizing processor utilization, inter-

processor communication overhead, input/output behavior, and overall task performance. The information conveyed by the displays and charts are as self-evident as possible, and they facilitate understanding. The type of information conveyed by a diagram is obvious, or at least easily remembered once learned. The choice of colors used takes advantage of existing conventions to reinforce the meaning of graphical objects, and are consistent across views.

5.1.2 Displays

In this section we describe the individual displays provided by the WWVM performance visualization component. The provided displays fall into one of four basic categories: utilization, communication, input/output, and task information.

Utilization displays are concerned primarily with processor utilization. Good parallel performance requires, among other things, that the computational work be spread evenly across the processors, that each processor do its share concurrently, and that additional work beyond that required by a serial algorithm be minimized [HMR 95]. Utilization displays are helpful in determining the effectiveness with which the processors are used and how evenly the computational work is distributed across the processors. Well-known utilization displays such as the utilization summary, concurrency profile, utilization count, utilization Gantt Chart, and Kiviat diagram were implemented. *Communication displays* depict interprocessor communication, and they are particularly helpful in determining the frequency, volume, and overall pattern of communication. WWVM supports communication animation, communication matrix, and space-time diagrams. *Input/output displays* show the input/output events, which are the events that involve reading from or writing to the disk. *Task displays*, such as depicted in the Gantt chart use information provided by the user. With the help of the Pablo instrumentation system they depict the portion of the user's parallel program that is executing at any given time.

Specifically, the user defines "tasks" within the program by using special Pablo routines to mark the beginning and end of each task and assign it a user-selected task name. The scope of what is meant by a task is left entirely to the user: a task can be a single line of code, a loop, an entire subroutine, or any other unit of work that is meaningful in a given application.

Here we will first describe the displays common to all or several of the basic categories. The processor states and operations may change, but the basic structure of the display and representation stays the same. Then, we will describe a few special displays. The current limit for most of the displays is 32 processors, which was adequate for the platforms on which this software was tested.

5.1.2.1 Common Displays

Gantt Chart

The Gantt chart depicts the operations performed by individual processors by a horizontal bar chart in which the color of each bar indicates the status of the corresponding processor as a function of time. The Gantt chart provides the same basic information as the Count display, but on an individual processor, rather than aggregate, basis.

Event Count Display

This display shows the aggregate number of processors in each separate stage as a function of time. Since the categories are mutually exclusive and exhaustive, the total height of the composite is always equal to the total number of processors.

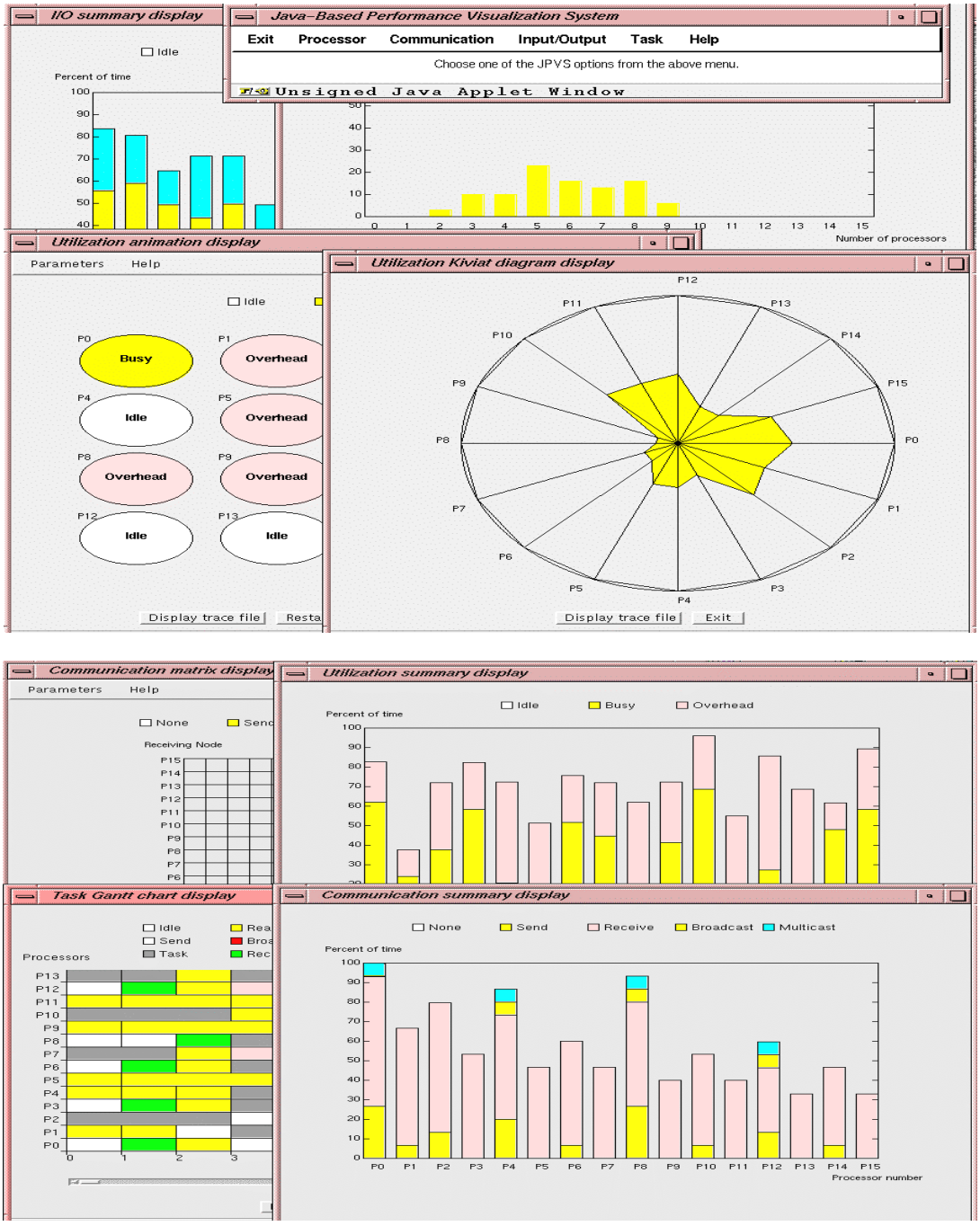

Figure 5 2. A snapshot of a sample performance visualization session.

Animation

In this display, the parallel system is represented by a graph whose nodes (depicted by numbered ellipses) represent processors. The status of each node is indicated by a different color, so that the ellipses can be thought of as the "front-panel lights" of the parallel computer. When the event traces involve communication events, the graph is further extended with arcs (depicted by lines between the ellipses) representing communication between processors. A line is drawn between the source and destination processors when each message is sent, and erased when the message is received. Thus, both the colors of the nodes and the connectivity of the graph change dynamically as the simulation proceeds. The lines represent the logical communication structure of the parallel program and do not necessarily reflect the actual interconnectivity of the underlying physical network.

Concurrency Profile

For each possible number of processors, this display shows the percentage of execution time during the run that *exactly N* processors were in a given state. The percentage of time is shown on the vertical axis, and the number of processors is shown on the horizontal axis.

Summary Display

This shows the cumulative percentage of execution time that each processor spent in each stage over the entire run. For example, when this display is used in the context of the processor utilization summary, it provides feedback on the overall efficiency of the program and load balance across processors.

Trace Display

This is a non-graphical display that prints an annotated version of each trace event as it is read from the SDDF trace file. It is primarily useful in the single-step mode for debugging or other detailed study of the parallel program on an event-by-event basis.

Clock Display

This display provides digital clock readings during the graphical simulation of the parallel program. The current simulation time is shown as a numerical reading, and the proportion of the full trace file that has been completed thus far is shown by a colored horizontal bar.

Statistical Summary

This is a non-graphical display that gives numerical values for various statistics summarizing processor utilization and communication, both for individual processors and aggregated over all processors. Representing raw data by statistical summaries, such as standard deviations, means, the percentage of busy, overhead, and idle time; total count and volume of messages sent and received; maximum queue size; and maxima, minima, and averages for the size and overhead incurred for both incoming and outgoing messages conveys the general trends rather than the detailed behavior.

5.1.2.2 Specific Displays

Kiviat Diagram

This display gives a geometric depiction of the utilization of individual processors and the overall load balance across processors. A spoke of a wheel represents each processor. The recent average fractional utilization of each processor determines a point on its spoke, with the hub of

the wheel representing zero (completely idle) and the outer rim representing one (completely busy). The distance from the hub corresponds to the percentage of use. Poor load balance across processors causes the polygon to be strongly skewed or asymmetric.

Communication Spacetime Diagram

In the Spacetime Diagram, the processor number is on the vertical axis and time is on the horizontal axis, which scrolls as necessary as time proceeds. Processor activity (busy/idle) is indicated by horizontal lines, one for each processor, with the line drawn solid if the corresponding processor is busy (or doing overhead), and blank if the processor is idle. Messages between processors are depicted by slanted lines between the sending and receiving processor activity lines indicating the times at which each message was sent and received.

Communication Matrix

This display shows the communication pattern among processors by using a square array, with sending and receiving processors along the two dimensions, respectively, for each message. At the end of the simulation, the Communication Matrix display shows the cumulative statistics (e.g., communication volume) for the entire run between each pair of processors, depending on the particular choice of color code.

5.1.2.3 Parameters

The execution behavior and visual appearance of the visualization displays can be customized in a number of ways to suit each user's taste or needs. The individual items in the *parameters* menu are described in this section.

Time Unit: The time unit chosen determines the relationship between simulation time and the timestamps of the trace events. By convention, Pablo provides event timestamps with a

resolution of microseconds. Consequently, a value of 100 for the time unit in JPVS, for example, means that each "tick" of the simulation clock corresponds to 100 microseconds in the original execution of the parallel program.

Start Time and Stop Time: By default, the system starts the simulation at the beginning of the trace file and continues to the end of the trace file. By choosing other starting and stopping times, however, the user can isolate any particular period of interest for visual scrutiny without having to view a possibly long simulation in its entirety.

Trace Node and Trace Type: These parameters determine which trace events are printed in the Trace display window. This feature allows the user to focus on events for a specific node and/or of a specific type, since looking at every event for every processor can be tedious and time consuming. The default value for both parameters is *all*.

5.1.3 Interaction with Pablo

Pablo is a performance analysis environment designed to provide performance data capture, analysis, and presentation across a wide variety of scalable parallel systems. Pablo helps to predict application or system behavior on massively parallel systems by means of post-execution analysis. By recording dynamic activity at the application level, one can identify and remove performance bottlenecks. To gain insight from this data and to tune both application and system software, the data is processed and presented in ways that not only show trends but also allow detailed exploration of small-scale behavior.

The Pablo environment consists of three primary system components: portable software instrumentation, portable performance data analysis (with a trace data meta-format coupling the instrumentation with the data analysis), and support for mapping performance data to both graphics and sound. From these three components, we adopted only the first one to use in the

WWVM. The performance visualization component replaces the functions of the other two components. The Pablo instrumentation component [Noe 96] can be further subdivided into three subcomponents:

- a graphical interface for interactively specifying source code instrumentation points;
- modified C and Fortran parsers that receive the instrumentation specifications from the graphical interface and emit instrumented source code (i.e., source code with embedded calls to a trace capture library);
- and a trace capture library that can record performance data generated by the instrumented source code when it is executed on distributed-memory parallel systems. All the idiosyncrasies of extracting data from a particular parallel machine generating event timestamps, as well as buffering data, are isolated in the Pablo trace capture library.

The Pablo graphical interface and the parsers cooperate to enable insertion of trace library calls at the selected instrumentation points in the user's code. The WWVM environment instead lets the users instrument an application source code by manually inserting calls to the Pablo performance data capture library. This minimizes the amount of software that needs to be ported into Java.

Pablo's only modification to the source code is the insertion of calls to the trace capture library. At execution time, the inserted instrumentation code invokes tracing routines supplied by the trace capture library, producing performance data in a standard trace format. It is possible to move an instrumented program to another parallel system that allows the same application data to be captured there, thus permitting cross-architecture performance comparisons. The Pablo trace capture library is scalable with the size of the system being studied and is also extensible, allowing users to add environment functionality as needed.

Although performance analysis occasionally requires a knowledge of architecture-specific data semantics, the Pablo design philosophy presumes that embedding this information in either the trace data format or the analysis software modules will preclude cross-platform portability and extensibility. For this reason, the performance data format is semantics-free (i.e., there are no predefined event types or data sizes).

5.1.3.1 Pablo Self-Describing Trace Data Format

The Pablo Self-Describing Data Format (SDDF) is a trace description language or data meta-format that specifies both the structure of data records and data record instances. SDDF does not restrict the user to a predefined record set, but allows description of general data records.

In SDDF, a header that can be interpreted by a simple parser determines the structure and semantics of the data in a trace file. This approach brings greater flexibility for filtering out unnecessary data.

Self-describing data files include a group of record definitions (i.e., the header) and a subsequent sequence of tagged data records. The tag identifies the type of the record, allowing the data record byte stream to be interpreted by using a particular record definition. The SDDF format supports the definition of records containing scalars and arrays of the base types found in most programming languages (i.e., byte/character, integer, and single and double-precision floating point) and multi-dimensional arrays whose sizes, but not number of dimensions, can differ in each record instance.

The Pablo portable trace data format links the Pablo instrumentation software, which captures dynamic performance data, and the JPVS, which analyzes and visualizes the performance data.

On a distributed-memory parallel system with hundreds or thousands of processors, the size of an event trace file can quickly reach many gigabytes. For the sake of compactness and efficient

processing, a binary version of SDDF also exists. On the other hand, the necessity of portability (even across machines with different byte ordering, floating point formats, or word lengths) and human-readability dictates an ASCII version of SDDF. Simple tools are provided for quick conversions from one representation to the other.

The ASCII and binary versions of the SDDF meta-format describe three classes of records:

- *Stream attribute records* contain information pertinent to the entire trace file such as the machine platform, or generation date of the trace file. Each stream attribute consists of a key and an attribute, both of which are arbitrary strings of characters.
- *Descriptor records* describe record layouts or structures. Each descriptor record associates a record name with a description of the fields that will appear in all data records having that name. In addition, descriptor records can contain both record and field attributes that provide descriptive information about records and fields.
- *Data records* contain actual event trace information. In the ASCII version of SDDF, a data record is interpreted by matching the record name in the data record with the name of a previously defined descriptor record. In the binary version of SDDF, records are matched to definitions via integer tags.

Figure 53 shows a sample SDDF file in the ASCII format. This file contains a stream attribute (the trace file generation date), two record descriptors (message send and message receive), and four data records. The integers “1” and “2” near the message send and receive record descriptors are the record tags used to match data records to definitions in the binary version of SDDF. The message send field “Source” is a one-dimensional array whose actual size will be specified in each instance of the message send data records. Using the record descriptors, the first data record shows that processor 0 sent 512 bytes to processors 1 and 3 at time 100.10.

```
SDDFA
    /*
     * "run date"  "January 1, 1997"
     */ ;;
    #1:
    // "event" "message sent to other processors"
    "message send" {
       double "timestamp";
       // "Source" "Sending processor"
       int "source";
       // "Destination" "Destination processor(s)"
       int "dest"[];
       // "Length" "Message length in bytes"
       int "length";
    };;
    #2:
    // "event" "message received from other processors"
    "message receive" {
       double "timestamp";
       // "Me" "my processor id"
       int "myid";
       // "Source" "Sending processor"
       int "source";
       // "Length" "Message length in bytes"
       int "length";
    };;

    "message send" {100.100000, 0, [2]{1, 3}, 512};;
    "message send" {100.100100, 1, [2]{0, 2}, 512};;
    "message receive" {110.102000, 1, 0, 256};;
    "message receive" {110.110000, 2, 1, 512};;
```

Figure 5 3. A sample SDDF file in ASCII format.

5.2 Data Visualization System

Data visualization has proved effective in exploring many types of science and engineering data and facilitating human comprehension of large amounts of complex data. The data visualization component of the WWVM is an interactive tool for drawing 2-D data plots (Figure 5-4). Its implementation in Java makes it platform-independent. It can accept data from programs executed in the context of WWVM as well as from ASCII files in tabular format (i.e., tables of columns of numbers) at user-specified URL addresses. All the options of the plotted graph are customizable through a GUI.

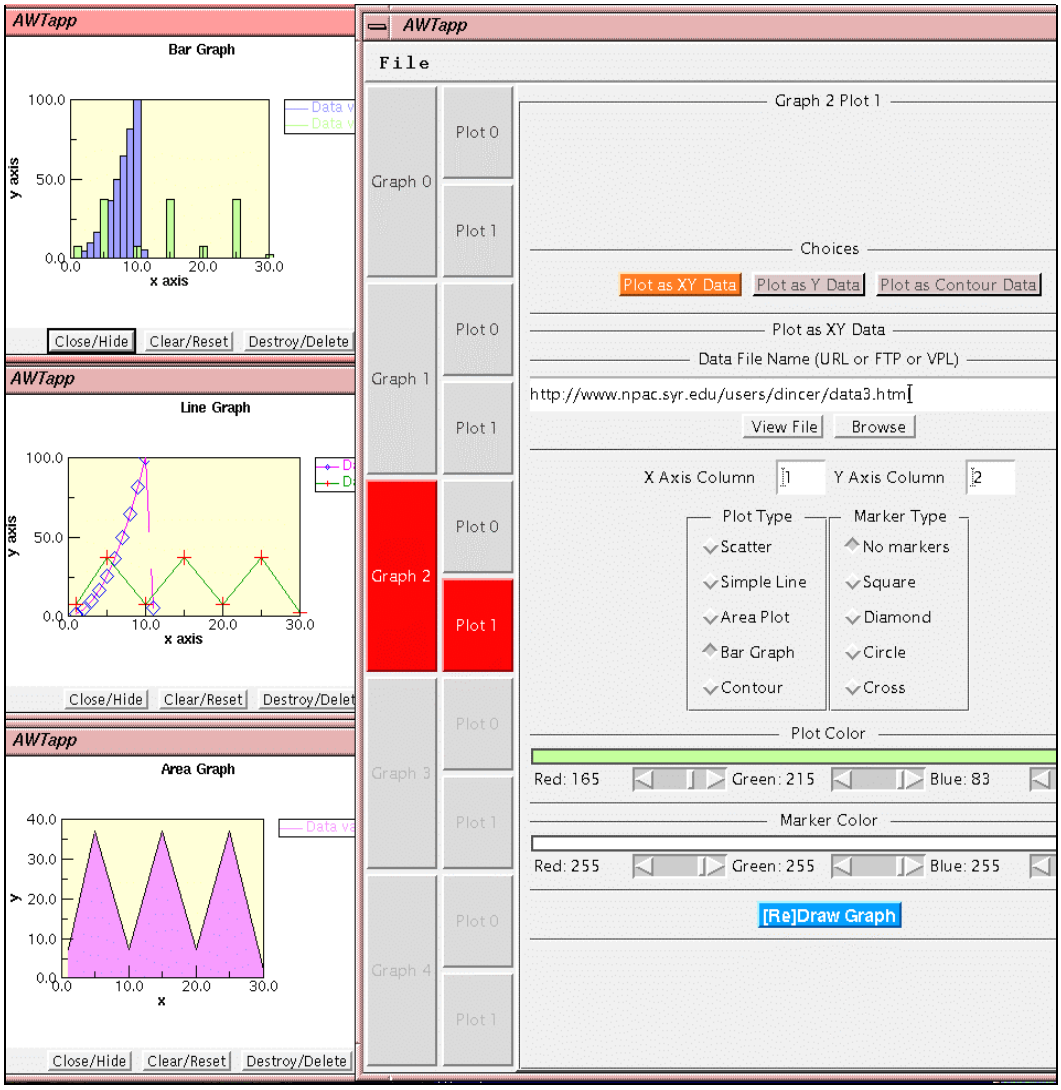

Figure 5 4. A snapshot of a sample 2-D Data Visualization session.

Currently the system supports line and scatter plots, bar charts, area graphs, and contour graphs. The graphs can be annotated with a title and axis labels in various font styles and colors. Users can view multiple data sets within the same window, the same data set in different windows, or different data sets in different windows.

It is also possible to delete previously drawn plots, replace the currently selected plot with another plot, or print out the currently selected plot. Furthermore, it is possible to save the current configuration of the system (i.e., files selected, plot and graph customization choices, etc.) into a configuration file, and retrieve this file for later use.

Extensions to the data visualization system is planned that will make it possible to plot arbitrary GNUPlot [WieK] files and save the current system configuration in a GNUPlot file format.

The data visualization component can access data files spread over the Internet on Web or FTP sites via Java's built-in network routines. Moreover, it can be used to plot data files in a WWVM user's account without hindering users' security and privacy. The data visualization component is built as a Java applet that can communicate with a back-end CGI file access module at the WWVM server site in order to obtain users' directory information. The contents of the directory are shown to the user through an extended network-capable version of the Java file dialog display. The selected file can then be sent to the applet through the socket connection. To save the current configuration, the data flows in the reverse direction towards the server.

5.2.1 Customize Plot Menu

5.2.1.1 Data Format

Data files are ASCII files with numeric data arranged in one or more columns separated by blank space. Lines beginning with a number sign (i.e., "#") are treated as comments and ignored. In all cases the numbers on each line of a data file must be separated by blank space dividing the line into columns. The user can select the format of data within a file. In the case of *XY* Plots, chosen *x* and *y* values from a line are plotted as a series of *XY* values to be plotted against *y*- and

x-axises. In *Y* Plots, WWVM data visualization component interprets the input data as a series of *Y* values to be plotted against a set of constantly-spaced *x-axis* intervals. Contour plots are done similarly.

5.2.1.2 Plot Styles

This option allows the customization of line colors and styles. Currently, plots may be displayed in one of six styles: lines, scatter points, lines with points, area plots, bar charts, and contour graphs. Line plots connect data points with lines so that changes or trends within the data can be observed. Scatter plots show the data points as markers so that groupings of data can be easily seen. When a dot type of marker is selected, there is a tiny dot at each point; this is useful for scatter plots with many points. Area plots fill in the data points with solid color so that similar and dissimilar data points are easily viewed. Bar charts display data in vertical bars so that it will be easy to compare data values. The contour style is used to draw contour graphs.

5.2.2 Customizing Graph Menu

The following are the customizing options of the graph menu:

Graph Background Color option sets the window background color.

Font Type and Color option selects the font and font color used in the graphics window for drawing the title and *x*- and *y*-axis labels.

Tics are drawn inwards on the left and bottom borders only. This is useful when doing impulse plots.

Title option produces a plot title that is centered at the top of the plot. Using the optional adjustment option, the title can be centered, or left- or right-justified at the top of the plot window.

X- and Y-axis labels option sets the *x*- and *y*-axis label that is centered along the *x*- and *y*-axis, respectively. Vertical (i.e., rotated) text is centered vertically at the left of the plot.

X- and Y-axis range option sets the horizontal (vertical) range that will be displayed. If only one value is provided, the range in the opposite direction is unaffected (or still autoscaled). To set a range back to autoscale, give a star as the value.

X- and *Y*-axis zero-axis option draws the *x*- or *y*-coordinate zero-axis. By default, this option is on.

5.3 Data Wrappers for Visualizing Program Data Structures

Data wrappers is a key base technology provided in the context of the WWVM that allows client-side Java applets to interact with running parallel Fortran, C, and HPF codes for steering computations or for monitoring and visualizing distributed data structures in a program. Implementation of data wrappers as a small portable library makes it possible to link it with programs written in different languages.

5.3.1 Implementation

The implementation of data wrappers was built upon the Parallel Tools (PTOOLS) Consortium's Distributed Array Query and Visualization (DAQV) system for [HacM 96, HMR 95, HacM 93, HacM 94] High Performance Fortran. DAQV supports a framework in which visualization and analysis clients connect to an HPF application with DAQV control for program-level access to array values and for computational steering. DAQV provides an OSF/Motif compatible user interface that is implemented using Tcl/Tk. Data wrappers use the Java AWT in the implementation of visualization and steering clients. DAQV's event control mechanisms were also modified to fit into the data wrappers framework.

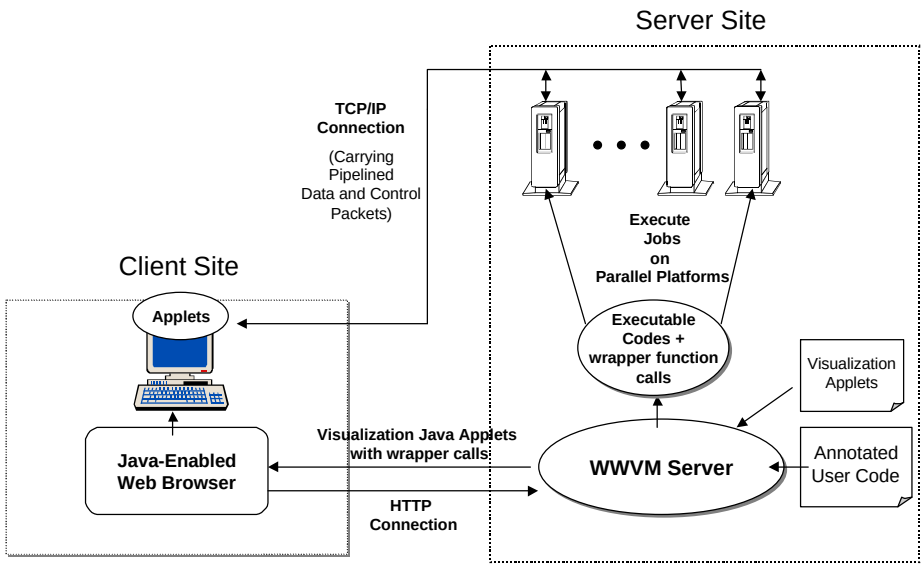

Figure 5 5. Operation of data wrappers for visualization and interactive computation steering.

Data wrappers inherit the server/client-based architecture of the WWVM. They were implemented as a software library containing distinct client- and server-side routines. The client-side and server-side routines of the data wrappers establish a link using TCP/IP sockets in order to convey data between both sides. Data wrappers use the External Data Representation (XDR) format when passing data between such dissimilar machines to take care of data presentation differences such as different byte order and floating-point format.

Client-side routines were completely implemented in Java, but server-side C routines and interfaces to Fortran and HPF languages were adopted. The client-side routines are dynamically linked with the visualization and/or computational steering Java applets and provide the means to communicate with the user program residing at the server-side. Their basic function is to access

the data elements in the user's program and either to convey them to the client-side functions or to modify their values based on the client-side commands.

Two basic models of interaction are supported between the client-side and server-side functions: push and pull. The *push model* is implemented by inserting simple function calls into the user's program code. These calls initiate a send or receive operation to transfer data between the server- and client-side. The push model is adequate if the programmer knows exactly what arrays are to be visualized and when they are to be visualized. This model resembles the loosely-synchronous, message-passing programming model implemented using blocking send and receive operations. The *pull model* allows the program execution and selection of arrays for visualization to be controlled through an external interface. A control client process, which is a kind of event handler, is linked with the user's program and runs on the server-site to direct the program execution. It also configures communication links and initiates data transfers to visualization clients. This model uses a sophisticated event protocol to interact with the control client. In the case of HPF programs, the pull model requires one process to play the role of the control process or the communication server. The data wrappers implement array access in a standard and portable manner in the HPF context by using HPF intrinsics. Different array access functions were written for different types of arrays with different ranks. The use of HPF for this purpose brings efficiency and correctness.

The conceptual difference between these two models is where the decision to extract an array originates. In the push model, the program pushes the data out, while in the pull model an external client reaches in and pulls the data out.

5.3.2 Operation of Data Wrappers

In both models the user's program should be annotated manually in order to insert the required function calls into the data wrapper functions at the proper points. Similarly, the user should insert the necessary functions at appropriate points in the Java applet in order to ensure correct behavior.

The first step in annotating a program is to register the data items to be manipulated. The registration process allows the data wrapper library to determine (at runtime) the arguments required to invoke a data access or manipulation function and to set the parameters in order to properly convey these data items over the socket connections. This information about each data item is stored in internal data structures. There are two general classes of data in the system, primitive data and aggregate data. *Primitive* data items are simple objects such as bytes, integers, single- and double-precision floating-point numbers, and text strings. *Aggregate* data items are vectors and two-dimensional arrays with an arbitrary number of elements of unsigned character (byte), integer, single-precision floating-point, and double-precision floating-point at the moment. For arrays, the rank, number of elements in each dimension, distribution and alignment information (only in HPF) are set in addition to the name, value, and type of the data item. A handle that is later used to access the properties of the data item is returned from registration functions.

Generally, only the primitive data types are suitable for use as parameters for computational steering. They can be associated with user interface widgets, and users are allowed to control their values. For example, a text parameter can be viewed or set using a text field widget.

Second, the connections between the user's program and the client-side should be established using a socket initiation function by specifying a port number and the buffer size for messages to

be sent. In the context of the WWVM this port is determined automatically, depending on the WWVM server being used.

Other functions that need to be called depend on the model being used. In the push model, `send()` and `receive()` functions accomplish the transfer of the specified data items between the client- and server-side. In the pull model, data client configuration takes place as part of the interactive control (event) protocol, and instead of `send()` and `receive()` functions, a `yield()` function is called which transfers the execution to the control client. The control client reads the requests in an *event queue* waiting to be serviced and calls the proper functions for accessing or setting specified data items. Once the events are serviced, the control is returned to the user's program, where it stays until the `yield()` function is again called. The responsibility of calling this function routinely belongs to the user's code.

5.3.3 Use of Data Wrappers for HPF Data and Performance Visualization

Languages such as HPF allow the programmer to specify processor arrangements, distribution and alignment of arrays, and execution of parallel loops. However, as programmers are removed from the fundamental operation of the system, the gap gets bigger between the low-level system information that is easy to monitor automatically and the application-level constructs that are meaningful to the user.

Traditional performance visualizations are based on physical processor load, and low-level message passing may not be very useful in the HPF context. HPF visualizations could be more effective if they were dependent on HPF's underlying programming and operating semantics. Data wrappers can be used to explore the effects of data distributions by linking a program's performance to its performance behavior. Since the program's performance directly depends upon

how its data structures are distributed, it is important to explain and illustrate the behavior of a program by graphical visualizations. Data-related views show the progress of the program's execution with emphasis on data alignment, distribution, and movement. A user can selectively choose and view some of the arrays used in the program. The visualization of data structures is especially important for languages like HPF, where data distribution determines the overall program performance. Programmers should see not only when and where communication takes place, but also which subarrays are moving between which processors.

Generic visualization displays based on universally available, low-level execution trace data are inevitably only indirectly related to the application level. Any high-level abstraction that relates to a particular application is likely to be application-dependent and require custom-written data collection and display techniques, therefore most performance visualization tools have featured only generic displays in order to avoid being explicitly application-dependent and restricted in applicability. In this respect, data wrappers provide an easy way to use application-specific visualization displays with Java applets. At the present time, data access patterns can be determined by manually inserting calls to special functions after each array access that mimics the owner-computes rule. These functions analyze the left- and right-hand sides of the array assignments and determine the type of access and whether communication is required to process the statement.

5.3.3.1 Related Work

The design and implementation the data wrapper libraries are influenced by a number of related research projects. The research work integrating Pablo with the Fortran D compiler [Adve 95], Tau with the pC++ compiler [BHM+ 94], Connection Machine Prism environment [Prism

96], and Breezy [BMM 95] (which provides high-level interaction with the pC++ system) demonstrate the importance of providing high-level semantic context.

The GDDT tool [KGV 95] provides a static depiction of the effects of different distribution methods in allocating parallel arrays on processors. The DAVis [Kemp 96] tool combines dynamic data visualization and distribution visualization. The IVD tool [KarH 93] uses a data distribution specification provided by the user to reconstruct a distributed data array that has been saved in a partitioned form.

Kimelman et al. [KMS+ 94, KMS+ 95] introduces new high-level views of HPF program behavior that show the communication activity in the context of the array distribution from which the compiler derived the communication. The programmer can see where the communication takes place, as well as what subarrays are moving between which processors.

The visual programming tool for Fortran D developed by Kondapaneni, Pancake, and Ward [KPW 92] provides some aid in specifying data distributions and alignments.

The Program Visualization (PV) environment [KimS 92] by IBM integrates the generation of visualization with language compilation. This environment supports the parallelization of sequential Fortran codes and generates traditional performance visualizations such as control flow and dependence graphs.

The work of Srinivas and Gannon [SriG 94] recognizes the importance of providing data visualizations in terms of semantics of the application of new languages like HPF and pC++. The proposed environment provides source code analysis facilities and visualization of data structures.