NPAC Technical Report: SCCS-???

Using Java and JavaScript in the Virtual Programming Laboratory: A Web-Based Parallel Programming Environment

Kivanc Dincer ***and*** **Geoffrey C. Fox**
Northeast Parallel Architectures Center
Department of Electrical Engineering and Computer Science
111 College Place, CST Mail-Stop 3-217
Syracuse University
Syracuse, NY 13244-4100
{dincer, gcf}@npac.syr.edu
http://www.npac.syr.edu/users/{dincer,gcf}/

Abstract

The Virtual Programming Laboratory (VPL) is a Web-based virtual programming environment built based on a client-server architecture. The system can be accessed on any platform (Unix, PC, or Mac) using a standard Java-enabled browser. Software delivery over the Web imposes a novel set of constraints on design. We outline the tradeoffs in this design space, motivate the choices necessary to deliver an application, and detail the lessons learned in the process. We discuss the role of Java and other Web technologies in the realization of the design. VPL facilitates the development and execution of parallel programs. The initial prototype supports high-level parallel programming based on Fortran 90 and High Performance Fortran (HPF), as well as explicit low-level programming with the MPI message-passing interface. Supplementary Java-based platform-independent tools for data and performance visualization are an integral part of the VPL. Pablo SDDF trace files generated by the Pablo performance instrumentation system are used for postmortem performance visualization.

1. Introduction

Virtual Programming Laboratory (VPL) (Figure 1) is a Web-based, integrated, parallel programming environment consisting of a visual file manager for manipulating files and directories in a users account, laboratory modules for compiling and executing message-passing MPI [1] programs (written in Fortran, C, and Java) and data-parallel programs (written in Fortran 90 and High Performance Fortran [2]), performance analysis and visualization subsystem to depict executed programsperformance behavior as animated or static displays, and a graphic plotting component to materialize output data as two-dimensional plots. VPL is unusual among Web services, because it allows users to create, edit, and execute files, rather than simply retrieve them by following hypertext links or by making simple database queries.

We expect that in the future the Web will be the standard user interface in many organizations for accessing computational resources. And instead of X-windows on Unix platforms, or Windows environments on personal computers and Macs, Web browsers will be used to manipulate the systemsresources. VPL is a proof-of-concept prototype that we developed using standard Web technologies. Java and JavaScript is used primarily to ensure interactivity and visual animations at the client site in addition to static components such as HTML. The interactive Web, HPCC backends controlled by Web servers extended with CGI modules, and HTTP-based communication represent critical enabling technologies in this framework.

VPLs basic function is to provide a user-friendly interface to server-site useraccounts and allow the use of HPCC parallel computing platforms and software on them in a convenient manner while giving the user an opportunity to observe the behavior and behavior of his/her program visually. Users do not need to log into a Unix account or type any Unix commands. Once they supply the required username and password, they are logged into their accounts and can use the computational facilities as well as educational materials from the same interface, the Web browser.

In this paper we are going to describe our experience of building VPL, a Web-based virtual programming environment. We primarily concentrate on using VPL for educational purposes based on our recent experiences. In Section 2 we describe the system architecture and potential areas of usability for VPL. Section 3 discusses the shortcoming of other Web technologies in user interface design, and emphasizes Javas contributions in this area. Section 4 describes the main components of the VPL main control panel with emphasis on the file manager, text editor, and programming labs. Section 5 describes the role of Java for visualizing the data structures and performance data traces. We investigate three components related to visualization: the data visualization component, the performance visualization component, and the 2D graphics plot package. We conclude by talking more on the Web security and future directions.

VPL makes it easier for its maintainer to make changes to the posted software, to indicate certain resources as sharable, and to change access restrictions and management policies in a quick and convenient manner without needing superuser privileges. It also makes it possible to access posted resources from any platform.

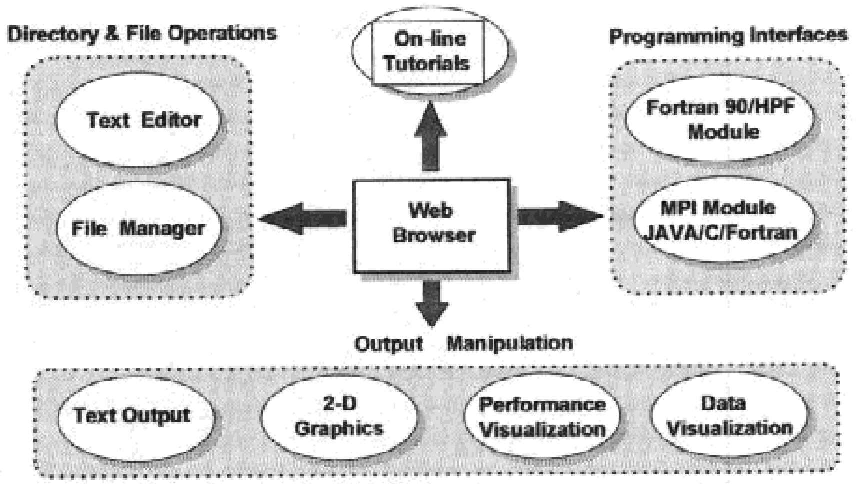

Figure 1. VPL Client-site configuration items.

2. The System Architecture and Areas of Usability

In recent years we have witnessed a rapid development of computer networks, dramatic improvements in the processing power of personal computers, and striking advances in magnetic storage technology. Furthermore, more and more colleges, universities, schools, companies, and private citizens connect to the Internet either through affiliations with regional not-for-profit networks or by subscribing to information services provided by for-profit companies. World-Wide Web (WWW) has emerged as an exciting and innovative front-end to the Internet. It provides Internet users with a uniform and convenient means of accessing the wide variety of resources (pictures, text, data, sound, video) available on the Internet, and Web browsers make the Internet a more user-friendly environment by integrating all those resources into a single tool that eliminates the necessity for novice users to struggle with a steep learning curve.

In this paper we are going to describe our experience of building VPL, a Web-based virtual programming environment. VPL is a client-server architecture --- a central Web server coordinates the accesses to local computational resources (Figure 2). The core server site components of the VPL architecture are the following:

1. *CGI scripts* written in Perl process the requests coming from the file manager, text editor, laboratory modules, and other Java applets. For example, scripts related to the programming laboratory pick up the program to be compiled, activate the selected compiler directly or by using makefiles execute the resultant program on the parallel machine, and report the user compilation/runtime errors or execution results.
2. The compilers for serviced languages in the context of VPL (i.e., Fortran 90, HPF, C and Fortran), MPI and other *runtime libraries* related to the compilation process, a Perl *interpreter,* Pablo software, all the related HTML files, and class libraries are stored on the server.

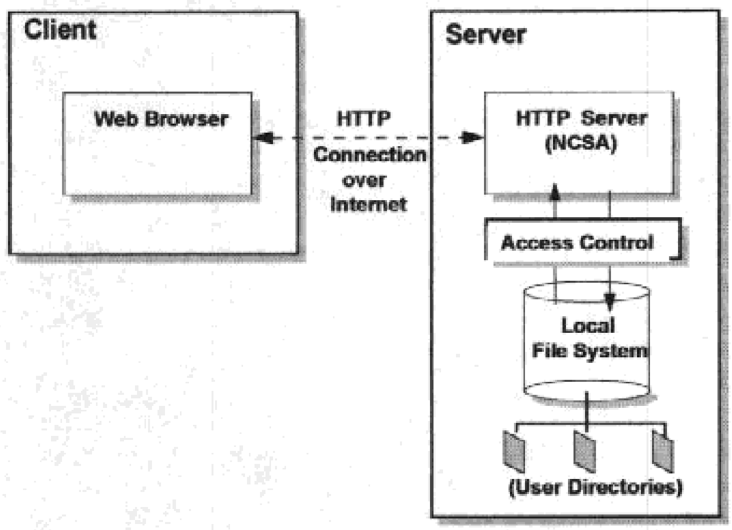

Figure 2. Client-server interaction in VPL.

Different versions of VPL were used for supervised on-site demos and unsupervised Web demos of local software products, and as a Web interface for using remote parallel computers. VPL can also be used for distant education as a part of collaborative distance teaching environments such as the Virtual University [3].

An initial prototype of VPL, called HPF/pC++ on the Web[4], with limited functionality was first used to demonstrate the current status of the PCRC [5] project at the Supercomputing 95 conference. NPACs F90D/HPF compiler and CSCs parallel C++ compiler were shown to share the same common runtime system through a Web interface supported by an HPCC and CGI back-end on a parallel cluster of workstations. We later used the same prototype in our on-site and remotely executed demos. The system dynamically generated specific HTML in response to compilation requests on arbitrary user codes. This is in contrast to the static compilation of demo programs, which was done off-line. Unlike static compilation of demo codes, this resulting document is unbounded in size and the HTML generated for any given code will change over time as the contents of the program code and data file change.

All the improvements in the Internet and Web technologies have opened many ways for educators to overcome time and distance in order to reach students. For educators, the WWW provides an exciting new opportunity for distance teaching and learning. The WWW and its digital libraries offer a powerful and continuously growing reservoir of educational material. Electronic mail, computer conferencing, and electronic bulletin boards facilitate communication among class members. Built-in, server-site extensibility mechanisms such as CGI and client-site support with Java, JavaScript, and helper applications open the way to world-wide distributed collaborative learning/teaching environments. The computer industry adds value in terms of quality browsers and multimedia VR front-ends.

We supplement the distance teaching efforts with the VPL software that provides virtual programming laboratory functions using Web browsers. We target high-performance computing parallel language teaching that requires to do many exercises and programming assignments. The Java language helps us to provide an interactive user-interface and well-developed graphical utilities for visualizing the results and behavior of programs.

With C. Hecht and K. Barbieri, we later tailored an upgraded version of our HPF on the Websystem for the Cornell Theory Center (CTC) environment (IBM SP-2 parallel machine on the Andrew File System) as the Web/HPF module. The Web/HPF module is used in CTCs Virtual Workshops since February 1997 as the interactive programming laboratory tool for teaching parallel programming and HPF.

VPL was also successfully used in a graduate level computational science course at Syracuse University during the Fall 1996 semester. Students used VPL to do their HPF and MPI programming assignments by accessing computational resources via standard Web browsers.

We recently started to invoke VPLs use in evaluating newly developed software among collaborating research institutions. Several commercial companies already have try-and-buyprograms that involve distributing their software on CDs with temporary licenses for potential customers. A similar need arises when a research institution plans to adopt a public domain software to use as part of its development cycle. Since many not-for-profit public domain software products do not satisfy all the criteria demanded by a commercial package, it may be crucial to test it under different circumstances before adopting and installing it on local computing platforms. This installation process sometimes takes several days, depending on the tools provided. It would be nice to assess the quality of the software before adopting it. Furthermore, a tool like VPL can help to post new patches to the current software on the Web, along with a running copy of the latest version of the software. For example, VPL users soon will be able to follow the day-to-day status of NPACs Java+MPIproject targeted towards writing SPMD-style Java code with calls to MPI message-passing routines.

3. Using Java for Graphical User Interface Design

Java is an excellent resource for building graphical user interfaces that we are used to seeing in many commercial software packages. When combined with the browser and other Web technologies, Java helps to provide a uniform user interface platform on every type of computer in the world, from personal computers to Unix workstations. In this section we describe the problems associated with using older, more standard Web technologies such as HTML and frames. We faced many problems while building various early prototypes of VPL. We investigate the shortcomings of HTML in the two categories described below. In the first category, we find that the introduction of JavaScript functions provides better support for more sophisticated applications, making user interaction more satisfactory and implementation cheaper and simpler. Yet Java, in addition to handling all the problems we mentioned in the first category, can also resolve the problems mentioned in the second category.

In general, there is a fundamental difference between the type of interaction supported by HTML and the forms of interaction to which we are all accustomed in graphical user interfaces. In GUIs, operations typically take the form of the user selecting an operand or operands through direct manipulation and then applying an operator by means of a menu selection or keyboard accelerator. In HTML-based user interfaces, there is no notion of selecting objects per se. Instead, the page the user is on is viewed as an implicit operand, and therefore the user can select a command to apply to that operand. In effect, an HTML interface can allow the user to apply a number of different commands to a single object, or a single command to one of a number of different objects. Commands that take multiple operands are much harder to implement.

We were able to simulate the behavior of traditional GUIs by using multiple frames, keeping the operands and operators in separate frames, and filtering all the submit and select actions through JavaScript functions before submitting them to the server.

3.1 Problems that can be solved using JavaScript

Transmitting information to server. There are only two ways for the browser to transmit information to the server from an HTML document. Pressing a submit button transmits the widget state, while selecting an anchor transmits a request to follow a hypertext link. Until a submit button is pushed, it is not possible for the server to determine anything about intermediate activities that a user might perform, such as typing text into an input field, toggling radio buttons or checkboxes, selecting items from menus, moving the mouse, and so on. JavaScript adds dynamism to the HTML pages through its ability to check widget states as soon as they are entered, and by being able to submitvalues explicitly without needing to use buttons or links. Consequently, it is possible to implement many features of sophisticated user interfaces over the Web such as immediate feedback to the user.

Asynchronous Communication with Server. It is not possible for an applicationcf2 to preempt the browser’s activity or provide any asynchronous communication.For example, it is not possible to notify the user asynchronously about the results of a background task or remind the user to save work. JavaScripts time-out mechanism and alert dialog capability can be used to do these kind of things. Furthermore, JavaScript modules can open HTTP connections asynchronously.

Transmission granularity. Each user action (submit or select) causes an entire new page or frame to be transmitted back to the browser over the network. There is no way for the application to cause an incremental update of a portion of the display. Even on high-bandwidth local area networks, transmitting and rendering large pages is time consuming; for distant browsers it becomes the dominant cost. We used hidden frames containing only the required field values and sent only those small frames to the server using JavaScript.

Widget set. The widget set available through HTML's forms capability is limited to submit buttons, radio buttons, checkboxes, pop-up menus, single/multiple selection scrolling lists, text fields and areas, mapped images, and text type-in widgets. It is not possible to combine either submit button or anchor behavior with a pop-up menu, or include icons in menus or scrollable lists to provide the sort of command selection model that is present in so many user interfaces. is no way to provide constraints on selection elements (e.g., toggling "list Java files" causes a file list menu to filter out non-Java files).

Native Submit Buttons. Web pages with custom image buttons instead of the native ones provided by the browsers look more uniform across platforms, but users must learn to recognize buttons anew for each idiosyncratic application. Despite this disadvantage, we chose to use the native Windows system's submit buttons for our interface to achieve a more uniform look.

Long pages with buttons. We found that some HTML pages, such as the tutorial and print windows, always produced pages that were multiple screens in height. We replaced these windows with two frames, one of which contains the global buttons that are always visible while manipulating the document in the other one.

Viewport positioning. Another problem is that HTML and HTTP do not provide effective control over viewport positioning. Because browsers cannot informthe server of the browsers window size or exact viewport position within a document, a server cannot cause the browser to scroll to a particular location. This means that it is often necessary for the server to refresh a whole page by forcing a redirection on the browser that would otherwise not be necessary simply in order to scroll to the desired position on the page. Although it is possible using named anchors to tell the browser where to scroll to on the new page, this is very coarse-grained control.

Action of a form. The action associated with a form is restricted to only one URL which inhibits having different actions with different submit buttons. Recently, JavaScript 1.1 made it possible to change dynamically the target of the action of a page or button.

3.2 Problems that can be solved only by Java.

Graying out invalid options. There is no way to gray-out either graphics or menu options in HTML that would have the obvious benefits of alerting the user of the existence of inapplicable commands and preserving registration. This means that the interface designer is forced to generate two sets of graphics: one for active commands, and one for any inapplicable commands. This doubles the network traffic associated with downloading the graphics, as well as increasing the human effort required to create the service.

Keyboard Accelerators. There is no way to associate keyboard accelerators with submit buttons. Binding the Enter or Return key to submit is an interesting special case of binding the default selection.

Little control over page appearance and registration. The application has very little control over the location of displayed objects on the finally rendered page, or on how and to where the browser will scroll the page if the page does not fit into a single screen. HTML explicitly yields rendering decisions to browsers. This has many advantages for browsing hypertext documents, but it proves awkward for interface design.

Scrollbars in textareas. It is not possible to control the position or presence of scroll bars on textarea widgets. This is a particular problem, because the default of most widget sets is to put the vertical scroll bar on the right of the text widget and the horizontal scroll bar on the bottom. For large textareas, this often results in the vertical scroll bar being scrolled off the right of the user's viewport, and the horizontal scroll bar being scrolled off the bottom of the viewport. For some applications it may be desirable to have scroll bars on all four sides of a viewport.

Pop-up menus. VPL user interface has a column of buttons at the left of the page that will invoke the file, directory, and other VPL functions. The number of buttons will only increase as the system becomes more sophisticated. Because HTML does not support pop-up submit buttons, we were unable to implement the obvious and familiar behavior of a menu bar. The irony of this is that associating an action with a menu selection is probably the most common form of interaction that users have with menus. The only "correct" model for command menus in HTML is the exhaustive enumeration of the commands as submit buttons.

When the number of commands becomes too large to support in this way, an alternative way is to partition the commands into broad classes and put the commands on menus and the name of each menu in front of it as a submit button. This means that the user must select an option from the menu and then click on the submit button to execute the selected operation. This is non-standard, but an applicable solution.

4. VPL Main Control Panel Items

This section describes the main control panel items of the VPL illustrated in Figure 3.

4.1 File Manager

File manager is a simple visual file browsing tool similar to its Windows-based PC counterparts that perform directory and file operations.The file manager has multiple frames: one for operation buttons, two for listing the usersfiles and directories, and another one for displaying the currently selected file and directory. Most of the actions selected by pressing the buttons are applied to this current directory/file pair.

Directory buttons are available for opening a new directory, for removing the currently selected, or for renaming the currently selected directory. File buttons are used to copy a currently selected file into the same or another directory, to remove the currently selected file, or to rename the currently selected file. View/Print buttons open a separate window to display the contents of the currently selected file. The user can print the contents of a file by using the browsers print function. In addition to all of the above, buttons for activating the text editor or HPF and MPI laboratory sessions on a separate window are provided for the users.

All user inputs are validated by client-side JavaScript modules before submission to the server site. This eliminates some of the unnecessary work on the server and tightens the security. For example, we filter the filenames to prevent meta-characters like ;or &and check to see if the new directory name conflicts with an already existent directorys name. The associated server site CGI scripts manage the directory and file operations appropriately and either send the user a new directory listing or an error message. Most of the time, the server will not send an error message, because of the precautions taken at the client site.

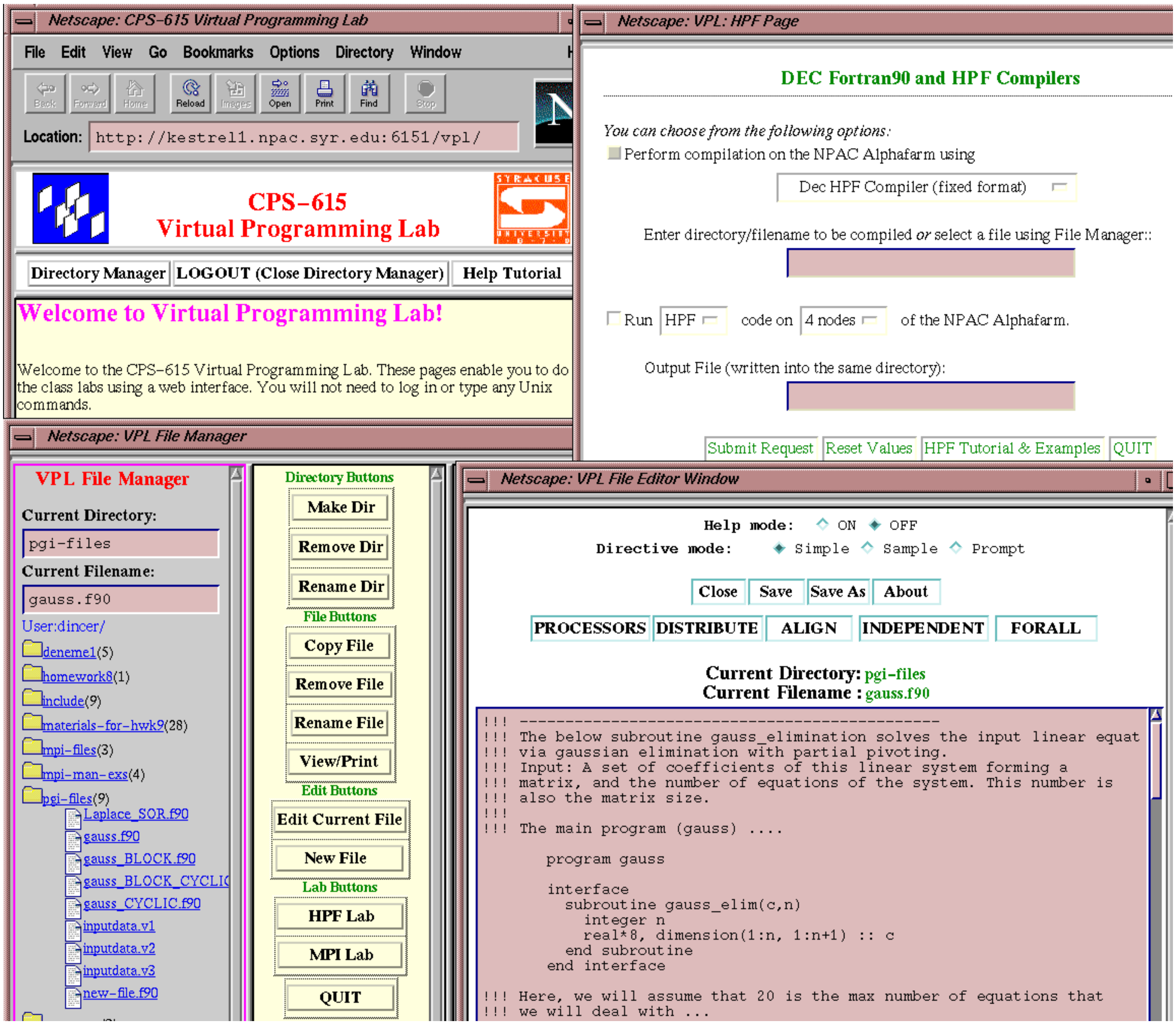

Figure 3. Snapshot of VPL User Interface.

4.2 Text Editor

The CGI-based text editor with extended functions written in JavaScript allows the user to create new files or edit the existing files without leaving the virtual lab environment. The editor functions are supported/complemented by CGI scripts on the server site.

There are two other alternatives for choosing a text editor to be used in the VPL environment: We could choose to use traditional Unix editors such as vi, emacs, or pico. We could choose to use a platform-independent, Java-based editor. Using a CGI- or Java-based editor has the advantages of doing most of the routine editing workon the client site and being platform-independent. They work similarly in every platform from which the user calls them. Their functionality is embedded in the browser itself. The editor module communicates with the server only for savingand loadinguser files. On the contrary, standard Unix editors can be used only when VPL is activated from Unix workstations supporting X-windows. Furthermore, since the editors are on the server site, every key stroke travels to the server site and back.

The most important reason for not using standard editors at all was that they give users a way to go out of their own directories, which we certainly could not allow. While using the CGI-based editor, the source of the **load** and **save** operations are verified by the JavaScript functions and CGI modules to make sure that the user accesses only the files in his/her own directory. The uploading/downloading of files from/to client site user accounts is an essential function of such an editor.

We also provide some integrated on-line helpwith the editor for the novice High Performance Fortran (HPF) users. When the user switches the help mode on, a hint window appears for each selected HPF directive. The supplied hint may be as simple as typing an example directive of the chosen type directly into the editor window (simple) to prompting the user to enter variable parts of the directive while the other parts are filled out automatically (prompt mode).

4.3 HPF and MPI Programming Laboratories

VPL users are represented with a forminterface where they select the services that they require. VPL supports HPF/Fortran 90 programming and MPI programming in C/Fortran 77/Java on parallel machines.Users may choose a file for compilation and an appropriate compiler along with the number of processors needed for execution on the target (Figure 4). The compilation is achieved either by directly activating related compilers on the target machine or by using makefiles that indirectly activate the compilers (only if the necessary object file is not already in the user space). Using makefiles prevents redundant re-compilations. Specifying input files or output files for redirection of program input and output is also possible. The output can be written in a specified file in the same directory.

MPI Lab opens the MPI laboratory window for compiling and executing selected C and Fortran 77 programs with calls to the MPI message-passing library. **HPF Lab** opens the HPF laboratory window for compiling and executing selected Fortran 90 and HPF programs. For the HPF and Fortran 90 programs, the source files should have an extension of .f90. This is also validated in the client site.

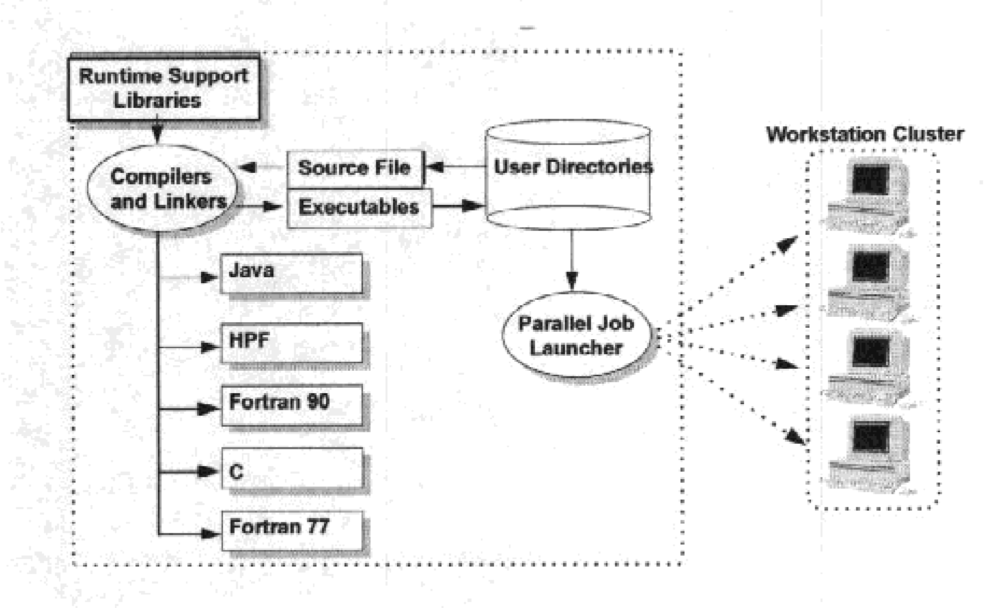

Figure 4. Server site compilation items.

In order to wisely manage disk space occupied by each user, executable files are implicitly named in a special way. The associated CGI scripts on the server site compiles the given programs. For each different type of compiler there is a unique target executable file name that is written to a users home directory, not in one of the subdirectories. This ensures that users will not have tens of executables taking a lot of space and that disk space occupied by each user is managed wisely at the expense of a bit less flexibility.

5. Using Java for Visualization

Visualization has been the cornerstone of scientific progress throughout history. Virtually all comprehension of science and technology calls on our ability to visualize. Graphical visualization is a standard technique for facilitating human comprehension of complex phenomena and large volumes of data. In fact, the ability to visualize is almost synonymous with understanding. Visualization can be thought of as the last step of solving computational problems or a form of assessment of the results.

We employ visualization components written in Java for post-mortem performance visualization of parallel message-passing programs and for the visualization of data structures of parallel programs and also the results produced. 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. Therefore, it 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 the data. Through a mixture of tools and techniques we seek to promote new dimensions of insight into problem solving by using current technology.

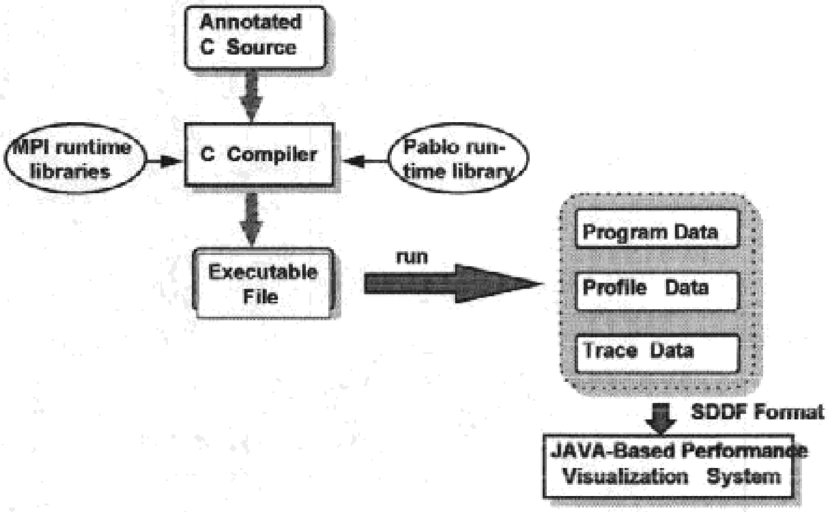

Figure 5. Preparing visualization traces for a representative C + MPI code.

5.1 Java-Based Performance Visualization System (JPVS)

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. We can count Comet [6], IPS [7], ParaGraph [8], Paws [9], and TraceView [10] among these tools.

We have developed a software tool, **J**ava-Based **P**erformance **V**isualization **S**ystem (JPVS), that provides a detailed, dynamic, graphical animation of the behavior of message-passing parallel programs, as well as graphical summaries of their performance. A concept demo of JPVS was prepared for the ARPA PI meeting held in June ‘96. JPVS helps to visualize execution traces (in Self-Defining Data Format - SDDF [11]) generated from Fortran or C codes instrumented with Pablo trace collection (instrumentation) library calls (Figure 5). Pablo [12] 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.

In an attempt to gain insights that might be missed by any single view, JPVS provides many different visual perspectives from which to view the same performance data.. It includes modules for visualizing processor utilization, inter-processor communication overhead, input/output behavior, and overall task performance.

Postmortem Analysis. JPVS is currently used only for postmortem visualization, It uses an SDDF trace file created during the execution of the parallel program and saved for later study. Although a real-time performance visualization would be possible, it is not desirable because of three major impediments. First, it is difficult to extract performance data from the distributed-memory processors and send it to the outside world during execution without significantly perturbing the application program being monitored. Second, the network bandwidth between the parallel machine and the graphical workstation, as well as the drawing speed of the workstation, are usually inadequate to handle the extremely high data transmission rates that would be 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.

In designing JPVS, our principal goals were to build a system that is easy to understand, easy to use, and portable from platform-to-platform.JPVS 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 users parallel program need not be extensively modified to obtain the data on which the visualization is based. JPVS 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 users 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 having a high-powered graphical UNIX workstation at the client end. On the other hand, JPVS is based on the Java AWT and thus runs on a wide variety of scientific workstations and personal computers from many different vendors. JPVS 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, Thinking Machines). Therefore, the package is capable of displaying execution behavior from different parallel architectures and parallel programming paradigms.

JPVS provides sixteen 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. 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.

5.1.1 Displays

In this section we describe the individual displays provided by JPVS.The displays of JPVS fall into one of four basic categories: utilization, communication, input/output and task information (). ***Utilization displays*** are concerned primarily with processor utilization. They are helpful in determining the effectiveness with which the processors are used and how evenly the computational work is distributed across the processors. ***Communication displays*** depict interprocessor communication and they are particularly helpful in determining the frequency, volume, and overall pattern of communication. ***Input/output displays*** show the input/output events, which are the events that involve reading from or writing to the disk. ***Task displays*** use information provided by the user. With the help of the Pablo instrumentation system they depict the portion of the users parallel program that is executing at any given time. Specifically, the user defines taskswithin 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 that we tested this software.

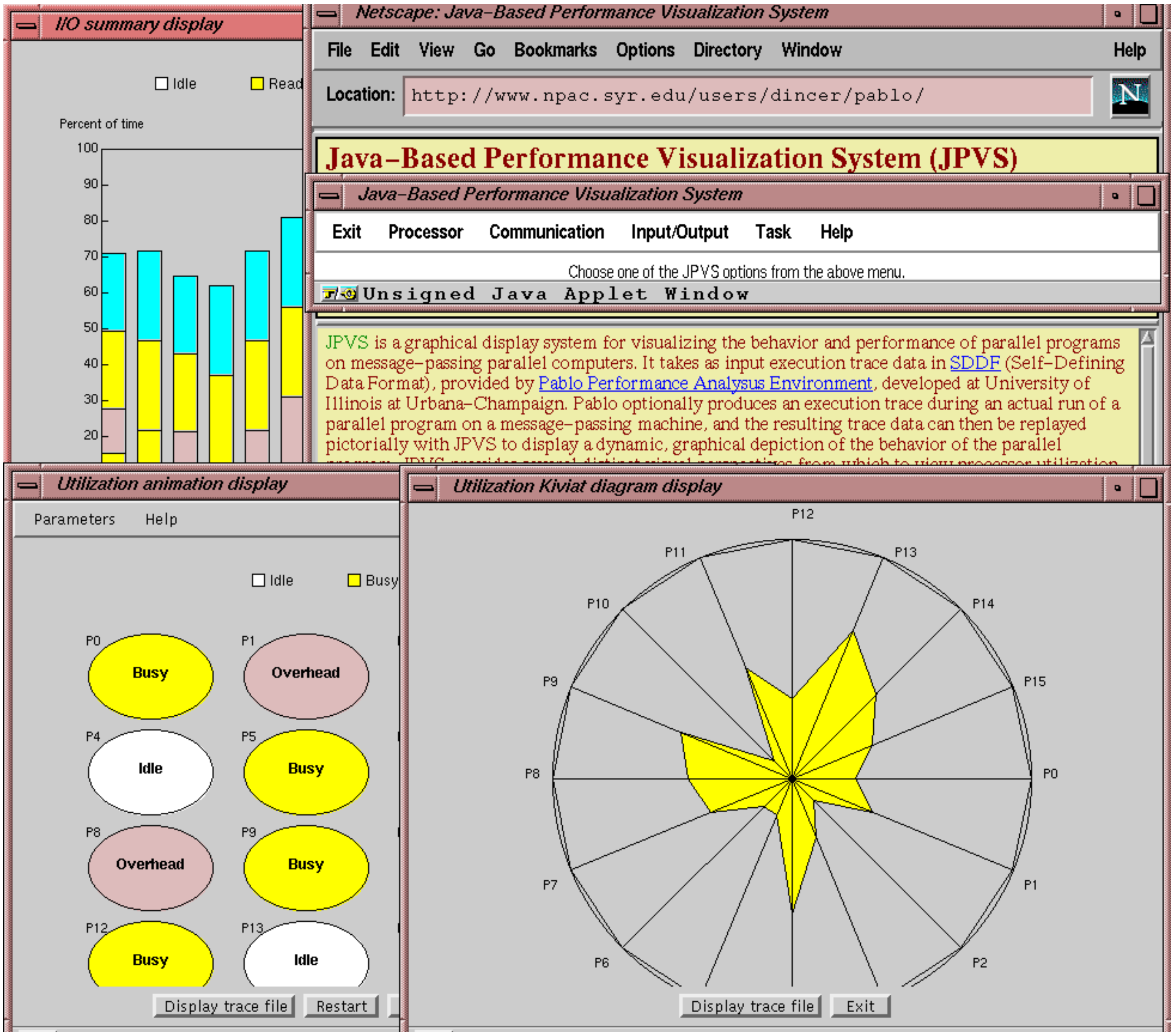

Figure 6. Snapshot of a sample JPVS session

A. Common Displays

1. **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.

2. **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.

3. **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 lightsof 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.

4. **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.

5. **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.

6. **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.

7. **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.

8. **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. The data provided include 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.

B. Specific Displays

1. **Processor Utilization Kiviat Diagram**
   This display gives a geometric depiction of the utilization of individual processors and the overall load balance across processors. Each processor is represented by a spoke of a wheel. 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.

2. **Communication Spacetime Diagram**
   In the Spacetime Diagram, 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.

3. **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.

C. Parameters

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

- **Time Unit:** The relationship between simulation time and the timestamps of the trace events is determined by the *time unit* chosen. 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 tickof the simulation clock corresponds to 100 microseconds in the original execution of the parallel program.
- **Start Time and Stop Time:** By default, JPVS 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.*

5.1.2 Interaction with Pablo

Pablo [12] is a performance analysis environment designed to provide performance data capture, analysis, and presentation across a wide variety of scaleable 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, support for mapping performance data to both graphics and sound. From these three components, we adopted only the first one to use in the VPL. JPVS replaces the functions of the other two components. The Pablo instrumentation component [13] 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 users code. In the VPL environment, we instead let 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.

Pablos 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 which allows the same application data to be captured there, thus permitting cross-architecture performance comparisons. The Pablo trace capture library is scaleable 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 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 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. This feature makes it a meta-format. Self-describing data files include a group of record definitions 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 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. run. 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 7 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 1and 2near 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 Sourceis 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 7. A sample SDDF file in ASCII format.

5.2 *VPL*Plot: Using Java for Plotting 2-D Data Graphs

VPLPlot is an interactive tool for drawing 2-D data plots (Figure 8). Its implementation in Java makes the VPL*Plot* platform-independent. It can accept data from programs executed in the context of VPL 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 is customizable through a GUI.

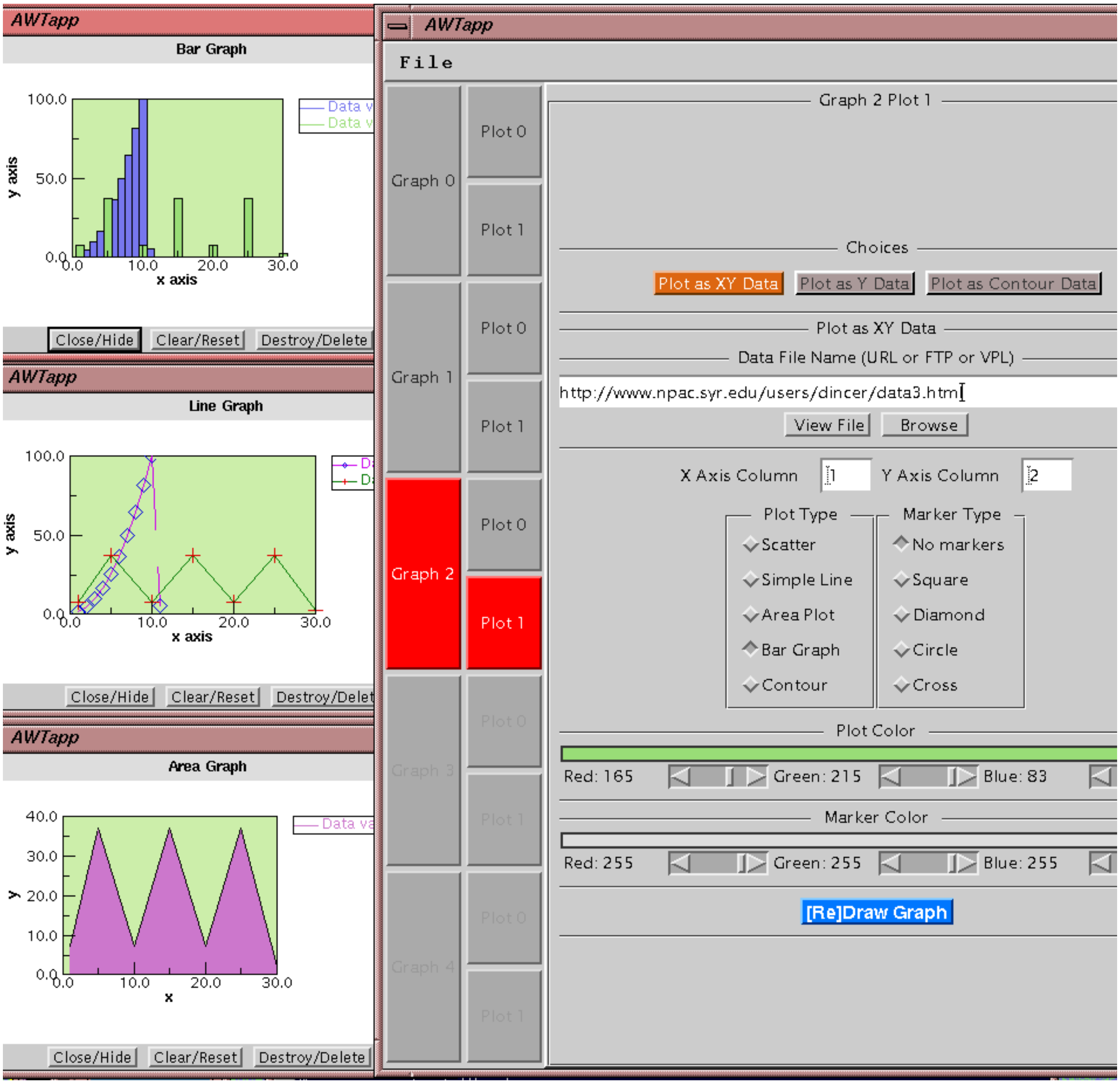

Figure 8. Snapshot of a VPLPlot session.

Currently we support 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. You 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 VPLPlot (i.e., files selected, plot and graph customization choices, etc.) into a configuration file, and retrieve back this file for later use.

We are planning an extension to the VPLPlot that will make it possible to plot arbitrary GNUPlot [14] files, and to save the current VPLPlot configuration in a GNUPlot file format.

VPLPlot can access data files spread over the Internet on Web or FTP sites via the Javas built-in network routines. Moreover, VPLPlot can be used to plot data files in a VPL users account without hindering the security and privacy of VPL users. The VPLPlot is a Java applet which can communicate with a back-end CGI file access module at the VPL server site in order to obtain the users directory information. The contents of the directory is shown to the user using 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

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 format of data within a file can be selected by the user. 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, VPLPlot 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.

Plot Styles. This option allows customization of the 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 a marker 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. Contour style is used to draw contour graphs.

5.2.2 Customize 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 labels.

Tics. By default, 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.** This command sets the *x*-axis (*y*-axis) label that is centered along the *x* (*y*) axis. 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.** Setting the *x*-axis (*y*-axis) zeroaxis draws the *x*-axis (*y*-axis). By default, this option is on.

5.3 Data Wrappers for Visualizing Program Data Structures

Data wrappers allow users to pass data from programs written in C or Fortran to Java applets (and vice versa) for steering the computations or monitoring and visualization of the data items. The data wrappers at both ends inherently communicate with each other using the Berkeley Unix socket mechanism (i.e., TCP domain sockets). The data flows from applet to the executable program and vice versa in a way similar to message-passing with blocking calls. The program and the applet are synchronized loosely by passing messages over the sockets. The data wrappers use the External Data Representation (XDR) format when passing data between such dissimilar machines to take care of different byte order, floating-point format, etc.

Data wrapper functions can be investigated in two categories. **Description functions** help to define the parameters that the user can adjust at run time to affect the action of the computation, and the data items that will be passed to the applet, and **Communication functions** take care of the actual send and receive process during the actual simulation. The user should put the necessary functions at appropriate points in the program and Java applet in order to ensure correct behavior.

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.

Any data type can be used as input, but generally only the primitive data types are suitable for use as parameters. The only difference between a parameter and module input is that parameters are usually associated with user interface widgets. For example, a text parameter can be viewed or set using a text field widget. Users generally should be allowed to control parameter values. However, the program can set a parameter value internally at any time, which may be necessary if the user sets a parameter to an illegal value. The following are sample data wrapper function definitions (in Fortran) that we developed for data visualization and computation steering:

```
int create_parameter_TYPE(name, init, minval, maxval)
int create_in_port_vector(data, dim1, type)
int create_in_port_2D_data(data, dim1, dim1,type)
int create_out_port_vector(data, dim1, type)
int create_out_port_2D_data(data, dim1, dim1,type)

int set_parameter_TYPE(name, value)
int set_vector_TYPE(name, value)
int set_2D_data_TYPE(name, value)
int get_parameter_TYPE(name, value)
int get_vector_TYPE(name, value)
int get_2D_data_TYPE(name, value)

int connect_widget(param_num, widget_type)
int modify_parameter_TYPE(name, type, init, minval, maxval)
```

The name specifies the name of the parameter or data item. Init, minval, and maxval specify a parameters initial value and valid range of values, respectively. type changes according to the type of the parameter or data. Value keeps the value of the parameter or data item, while dim1 and dim2 declare the number of elements in 1D and 2D arrays.

6. Comparison of VPL with EPIC: a Client-Site Virtual Programming System

EPIC (EPCC Interactive Courseware) [15] is an on-line interactive education software developed at Edinburgh Parallel Computing Centre that combines both on-line exercises and hypertext course materials. EPIC allows users to read through the course notes on the Web at their own pace, giving the user the option of assimilating information at a self-determined speed. EPIC also contains an on-line interactive exercise component that helps the users to test and make use of their newly acquired skills.

Having the exercises accessible directly within the framework of one single courseware package allows a much smoother method of working. In many instances this will remove the need for configuring some machines to run the programs during the courses. EPIC can be used to automatically configure the machine with the appropriate software and allow the users to spend time studying the course materials rather than setting up of the system.

As seen, EPIC and VPL are similar tools that have been built using Web and HPCC technologies. The most significant differentiating property between them is that EPIC provides a virtual programming system using the applications and scripts at the client-site, while VPL depends mostly on the server-site software. We can compare the other properties of these systems as follows:

Ease of installation. In order to use the EPIC package, the user must first prepare the local system for EPIC which involves downloading, uncompressing, and un-tarring the shell and Perl scripts that control the execution. In addition, the user needs to transfer the client-side of the EPIC tutorials and add EPIC Mime type to the .mailcap file (i.e., file applications/x-epic entry) and specify the EPIC control master as the corresponding helper application. This causes the browser to launch the specified helper application that is capable of understanding the information sent from an EPIC server, rather than trying to decode EPIC Mime types. This setup is required of all new users.

Although automatic ways to download and install these software packages to the users system have been included in the EPIC package, the user still needs to make decisions about where to install the new software and how several environment variables should be set up. In addition, several potential problems may be encountered during the setup of the scripts. VPL, on the other hand, requires no initial setup by the user. Anybody having a Java- and JavaScript-enabled browser can easily access the VPL.

Portability and client-site expectations.EPIC needs several software packages (Perl, xterm, and make facilities) to be installed on the users system. In addition, each exercise needs its own local applications. For example, to run an MPI application, the CHIMP version of MPI developed by the Edinburgh Parallel Computing Centre should be installed on the users machine. To run an HPF application, either PGIs or DECs HPF compilers should exist on the local machine. EPIC is designed to support users on a set of selected Unix platforms.

These requirements of EPIC are expensive requirements for sites with limited funds and computational power. Many of the licensed software packages, such as the commercial HPF compilers, are b notwithin the reach of small institutions. In addition, an individual user may find the disk space required to install all those software packages a limiting factor.

In contrast, a VPL client may have a Mac, a PC running Windows or Linux, or any Unix workstation as long as there is a browser. Every type of software package that is expensive or that requires a lot of disk space is already installed on the VPL server site.

Usability Areas.EPIC is customized only for teaching. On the other hand, VPL is also targeted towards being a generic interface for parallel computer platforms. We mentioned various uses of VPL in the Introduction.

Cost. VPL makes it easy to update required educational software easy. It can be extended to include new scheduling policies, or new parallel machines at the back-end. All these changes can be made transparently to the user. EPIC users, on the other hand, have already installed the necessary client-site files, and would need to download the new copy of the software to take advantage of the improved version of EPIC.

Adaptability. VPL can be configured to offer more capabilities to Unix clients. For example, the user who has Xterm facility will be able to choose an editor of choice such as vi, emacs, or pico and may use xv, or debuggers for programming.

7. Server Security Measures

Since users have the capability of executing real programs using VPL, the server was configured in a very careful manner in order not to hinder the security of the entire system. As a first precaution, we used a Web server authorization mechanism to restrict accesses to the system. The Web authorization mechanism allows the Web administrator to specify as protected certain directories or files under a server. A password file is generated containing the usernames and associated passwords of all the valid users of the system. We have protected all the document directories and CGI script directories in this way. A user who does not supply a matching username/password pair is not allowed to access the class directories; instead a special page containing information about the system and contact information to the Web administrator is shown. One concern may be the security of transferring these passwords over the internet. In spite of common belief, in the decent browsers such as Netscape 3.0 passwords are not transferred in clear textform, but in encrypted form just as Telnet or FTP does. Therefore, it is as secure as using those popular network tools.

Another concern was possible attacks by other legitimate users of the system. To prevent this we set up a special account for the VPL and ran the server under this account. This is in direct opposition to the common approach of running the Web servers as nobodywith minimum privileges. This brought us the flexibility to use UNIX system file protection mechanisms to protect our directories from other users of the same system. In the common nobodyapproach, the access rights for all the files should be set as readable and executable by the world,which make them vulnerable to bad guystrying to steal homework solutions of other users. Furthermore, as much as possible, the system never indicates the actual location of the class accounts to the user. We always precluded this information from diagnostics and status messages or error messages.

In order to make it secure to run a server from a privileged account, we took several measures to prevent the mischief: First, there is no way for the users of the system to go out of their directories by using any of the utilities provided by the system. Commands such as delete/make/rename directory or delete/remove/rename/copy file always manipulate items in the users home account. Using JavaScript for validity checks at the client site, and more advanced CGI script based checks on the server site, guaranteed this result.

The input fields typed by the users are checked against all kinds of special Unix meta-characters, and rejected if determined to be invalid. In a sense, the users of this system have fewer rights than they would have with an actual account on the same system. However, for a class account the tools and utilities provided by this system should be more than enough. In many cases, Unix accounts give the users more rights than they actually needed anyway.

Users can only run executables that they have generated. They are not allowed to use any of the commands other than the provided by the VPL system. Furthermore, programs supplied by the user for execution may contain possible system calls. We filter the user programs for system calls,and reject the execution if one found. The user may also gain some access to system files such as the password file using input/output statements. The same problem has caused the Java language verifier to reject all input/output statements embedded in the applets loaded into the client side. We could have done something similar, but we chose to restrict the input/output syntax/semantics a little, and allow user input/output. The input, output statements are filtered against any files the users open outside their own directories. We do not allow using variables as filenames in open statements, since this would elaborate the filtering process and might force us to adapt Javas way of thinking. The user must always put the filename in quotes in the open statement so that the system can validate it.

8. Summary

In this paper we have summarized our experiences in building a Web-based virtual programming environment for parallel computing platforms. Our goal was to create a general programming environment to facilitate the development and execution of parallel message-passing and data parallel programs. VPL is a unique tool that allows browsing, program development, directory management, performance, and scientific data visualization within the framework of a single package. It is one of the first prototypes that uses the Web as the standard interface for accessing computational resources. We described the role of the Java language in providing platform-independent graphical user interfaces and visualization software components. We also described the VPL performance and data visualization components written in Java.

Acknowledgments

We would like to thank S. ElMohamed, M. Egilmezbilek, W. Furmanski, T. Haupt, D. Leskiw, X. Li, N. McCracken, M. Sen, and H. Topcuoglu for their feedback in various stages of this project and E. Weinman for proofreading this manuscript.

References

[1] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," International

Journal of Supercomputer Applications, vol. 8, no. 3 & 4, pp. 157--416, 1994.

[2] High Performance Fortran Forum, "High Performance Fortran Language Specification: Version 1.0," Scientific Programming, vol. 2, no. 1 & 2, 1993.

[3] Furmanski, W., "Next Generation World-Wide Web Technologies for Distance Education," talk presented at the Virtual University Conference, The Wharton School of Management, University of Pennsylvania, Jan.1995. (At http://kayak.npac.syr.edu:2005/WebTools/PotPourri/ WhartonTalk.ps)

[4] Cowie, J., Dincer, K., and Li, X., "Towards a Web-based PCRC Programming Environment," SCCS Technical Report, NPAC, Dec. 95.

[5] Parallel Compiler Runtime Consortium Project, at http://www.npac.syr.edu/projects/ pcrc/.

[6] Kumar, M., "Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications," IEEE Transactions on Computers, vol. 37, pp. 1088--1098, 1988.

[7] Miller, B. P. and Yang, C.-Q., "IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs," In Proc. Of the Seventh Conference on Distributed Memory Computer Systems, vol. 7, pp. 482--489, 1987.

[8] Heath, M. T. and Etheridge, J. A., "Visualizing the Performance of Parallel Programs," IEEE Software, vol. 8, no. 5, pp. 29--39, 1991.

[9] Pease, D., Ghafoor, A., Ahmad, I., Andrews, D. L., Foudil-Bey, K., Karpinski, T. E., Mikki, M., and Zerrouki, M., "PAWS: A Performance Evaluation Tool for Parallel Computing Systems," IEEE Computer, vol.24, no. 1, pp. 18--29, 1991.

[10] Malony, A. D., Hammerslag, D. H., and Jablonowski, D. J., "TRACEVIEW: A Trace Visualization Tool," IEEE Software, vol. 8, no. 5, pp. 19--28, 1991.

[11] Aydt, R. A., "The Pablo Self-Defining Data Format," Technical Report, Department of Computer Science, University of Illinois, Sept. 1996). (At URL http://bugle.cs.uiuc.edu/Projects/ Pablo/documents.html).

[12] Reed, D. A., Aydt, R., Madhyastha, T. M., Noe., R. J., Shields, K. A., and Schwarz, B. W., "The Pablo Performance Analysis Environment," Technical Report, Dept. Of Computer Science, University of Illinois, 1992. (At URL http://bugle.cs.uiuc.edu/Projects/Pablo/documents.html).

[13] Noe, R. J., "Pablo Instrumentation Environment User’s Guide," Technical Report, Department of Computer Science, University of Illinois, October 1996. (At URL http://bugle.cs.uiuc.edu/ Projects/Pablo/documents.html).

[14] Williams, T. and Kelly, C., "GNUPlot: An Interactive Plotting Program Manual, version 3.6a," at URL http://www.cs.dartmouth.edu/gnuplot/gnuplot.html.

[15] Edinburgh Parallel Computing Centre, "EPIC Home Page", at URL http://www. epcc.edu.ac.uk:0080/epic/.

URL for this paper: http://www.npac.syr.edu/users/dincer/papers/vpl/