# Toward Scalable Performance Visualization with Jumpshot*

*Omer Zaki, Ewing Lusk, William Gropp,* and *Deborah Swider*
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
{zaki,lusk,gropp,swider}@mcs.anl.gov

**Abstract**

Jumpshot is a graphical tool for understanding the performance of parallel programs. It is in the tradition of the upshot tool, but contains a number of extensions and enhancements that make it suitable for large-scale parallel computations. Jumpshot takes as input a new, more flexible logfile format, and comes with a library for generating such logfiles. An MPI profiling library is also included, enabling the automatic generation of such logfiles from MPI programs. Jumpshot is written in Java, and can easily be integrated as an applet into browser-based computing environments. The most novel feature of Jumpshot is its automatic detection of anomalous durations, drawing the user's attention to problem areas in a parallel execution. This capability is particularly useful in large-scale parallel computations containing very many events.

## 1 Introduction

Jumpshot is a graphical tool for investigating the behavior of parallel programs. It is a "post-mortem" analyzer, taking as input a file of time-stamped events, which we call here a *logfile*. The file is written by the companion package CLOG, also described here. Jumpshot and CLOG are only loosely coupled by the format of the logfile and may be used independently.

Jumpshot can present multiple views of logfile data. The primary view is a series of timelines, one for each process, showing with colored bars the state of each process at each time. This view can be zoomed and scrolled for close examination of specific times. Other views include histograms of state durations and a "mountain range" view showing the aggregate number of processes in each state at each time.

Jumpshot is implemented in Java. Advantages of this approach include portability and network capability (the ability to run as a applet in a Web browser). The choice of Java was also an experiment in exploring the features and capabilities of current Java environments, and we report here on our experiences.

Logfile-based tools similar to Jumpshot have a rich history. Commercial tools include Time-Scan [2] and Vampir [3], and academic tools that are distributed for use by others include Para-

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Graph [12, 11], TraceView [17], XPVM [15], XMPI [4], and Pablo [19]. Each of these shares some features with Jumpshot, but it is unique in its particular combination of features.

Jumpshot is freely available and is packaged with the MPICH portable implementation of MPI. MPICH can be obtained from `http://www.mcs.anl.gov/mpi/mpich`. The distribution includes the CLOG logging facility and an MPI profiling library for use with MPI programs. Jumpshot is particularly easy to use with MPICH's compiling and linking scripts [6].

This paper is organized as follows. In Section 2 we give a synopsis of the history of Jumpshot and the requirements that drove its development. In Section 3 we present a walkthrough of Jumpshot features. Section 4 presents the logging package that accompanies Jumpshot and writes the files Jumpshot reads. We present Java-related issues in Section 5. Section 6 shows a few interesting specific logfiles and illustrates the kinds of insights that can be obtained with such a tool. We conclude with a discussion of opportunities for future research in this area.

# 2 Background

In this section we give a brief history of the developments that led to Jumpshot, culminating in a list of requirements.

## 2.1 History

Jumpshot traces its history back to the original version of gist, a program that was delivered with, and only ran on, the BBN Butterfly parallel computer. Gist is still available as part of the TotalView© debugging environment, although it is now known as TimeScan. The original version of upshot [13] was written to provide the most useful feature of gist, its zoomable and scrollable timeline window, in a color version (gist was black-and-white only) that was non-proprietary and would run on any workstation supporting X-windows. (Upshot gets its name from gist, for which it is a synonym [5].) Over the years upshot expanded to include a number of other features as well [14]. The enhancements were motivated by the development of an application whose highly irregular and input-dependent behavior made an analysis tool of this type indispensable [16]. The logfile generation library developed along with upshot was called ALOG, for Argonne logging.

Upshot originally was written in X (with the Athena widget set). Because it was cumbersome to expand and maintain, we completely rewrote it in the Tcl scripting language, using the Tk companion graphics package [18]. This approach made for extremely rapid development, but the interpreted nature of the language made the graphics component slow when logfiles were large. To deal with this problem, we rewrote the main graphics component of upshot in C, using Tcl's C interface. This rewrite solved the performance problem but introduced a dependency on an unstable part of Tcl itself, as the C interface changed right after the publication of [18]. This version, called nupshot (for new upshot), can still be used, however, if one obtains an earlier version of Tcl/Tk. Until recently, nupshot has been our main performance analysis tool. Upshot, in pure Tcl/Tk, is also still available, but also requires an old version of Tcl. This instability of Tcl/Tk encouraged us to find an alternative.

A few years ago, we rewrote the logging library to improve in multiple ways on ALOG. There was also an intermediate library called BLOG. For historical continuity, therefore, the new package had to be called CLOG (pronounced see-log). It is described in detail in Section 4.

## 2.2 Requirements for a New System

More than ten years of experience with similar tools gave us a clear set of requirements that our next-generation performance analysis tool should meet.

- It is expected to have a long lifetime and to be the foundation for our future research in performance analysis. It should therefore be implemented in a stable and widely supported programming environment.
- The tool must be portable to all the types of workstations that parallel programmers would want to use. Although the use of X made previous tools portable within the Unix universe, portability to Microsoft-based environments is now required.
- It must be able to deal with large logfiles in a scalable way. While filtering of *very* large logfiles can be expected to take place either within the application program itself or in an intermediate preprocessing step, in order to be a practical tool, performance needs to be maintained in the presence of many thousands of graphical objects being displayed, zoomed, and scrolled.
- The display package (the Jumpshot part) must not be too tightly integrated with the logging package (the CLOG part), for the sake of flexibility within each.
- Support for nested and overlapping states is needed, although nested states occur far more often than overlapping ones, because of the natural nesting of subroutine calls in programs.
- Support for MPI concepts, such as communicators, is required. At the same time, it is overly restrictive to tie the tool to the message-passing model of parallel computation.
- It must be relatively easy to manipulate the display, turning specific states on and off, changing colors, and so forth.
- The ability to create printable versions of displays is crucial, for inclusion in documents.
- A desirable feature is to help the user find the "interesting" parts of what might be a large and confusing display. This is a research topic we are only beginning to explore.
- Multiple types of display are useful for looking at the behavior of a program from many points of view:
  - process timelines with zooming and scrolling
  - histograms of state durations and message data
  - "mountain ranges" for aggregating state data
- It must be possible for the user to define states that are meaningful in the context of the application, not just the system.
- It must be possible to query specific states and events for details.
- The logfile format needs to be flexible, to account for unanticipated types of events and states, summary data, and entirely new concepts.
- It is desirable that an event can be connected back to the source code that generated it.

Jumpshot currently makes some compromises in attempting to meet these requirements. These compromises are expected to be temporary.

# 3 Using Jumpshot

In this section we give a walkthrough of using Jumpshot to view a very simple parallel program. We defer to Section 6 the viewing of more interesting examples. Jumpshot can be used either in stand-alone mode by running the `jumpshot` command or, when properly installed, from a Web browser, in which case it will run as an applet. In either case it begins with a simple opening window, as shown in Figure 1. The "look and feel" of Jumpshot is by default "metal", the default appearance of the


Figure 1: Initial Jumpshot screen

Swingset class library. In the `System Options` menu it can be changed to have a Motif, Windows95, or MacOS appearance. Figure 2 shows the difference between Metal and Motif looks.

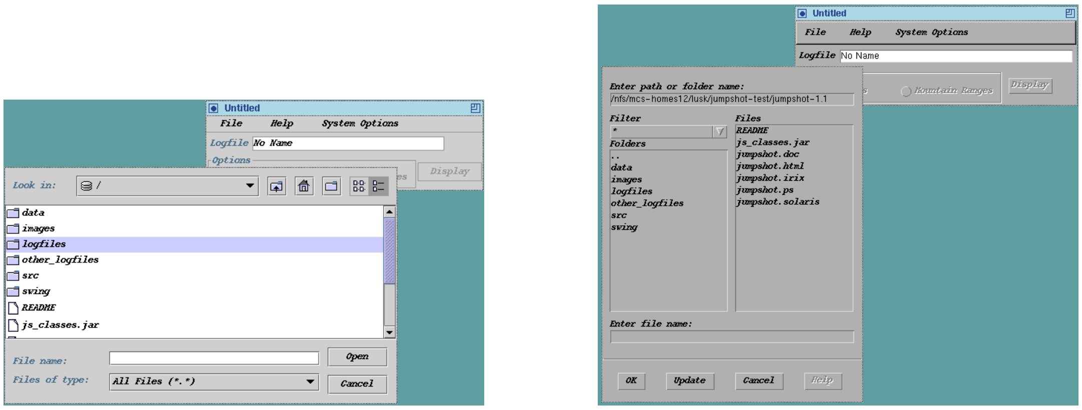


Figure 2: Two looks for picking a logfile to view

The logfile-selection menu allows one to browse in the file system for a logfile, produced in any of the ways described in Section 4. Then the main window appears, as shown in Figure 3. Since reading the logfile and drawing the initial screen may take some time, depending on the size of the logfile, a "progress" window appears while the logfile is being loaded. The initial display provides the primary view of logfile data, together with access to other views.

## 3.1 Timeline Display

The primary view of logfile data is the "timeline window" in which time (in seconds) is indicated from left to right and MPI process ranks are shown from top to bottom. Colored rectangles spanning sections of the timelines indicate that a particular process was in a particular state during the indicated time interval. These states are defined by the logging library and typically consist of MPI function call durations and the durations of user-defined states. Clicking with the mouse on such a rectangle pops up a small window containing detailed information (state name, precise duration, etc.) Arrows show messages, and details about a particular message (length, tag, etc.) appear when one clicks on the small circle appearing near the origin of an arrow.

As the user moves the mouse in this window, its position along the time axis is dynamically

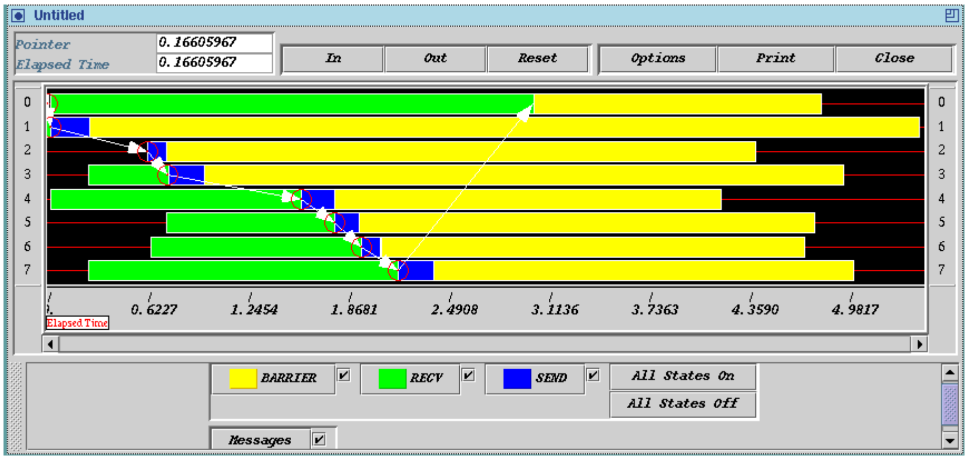


Figure 3: A simple ring program

tracked in the box labeled "Pointer" at the upper left of the frame. If the user presses the "t" key, the origin of the "Elapsed Time" field is set wherever the cursor is, and elapsed time is set to 0, enabling very precise measurements of time intervals with the mouse.

If the "z" key is pressed with the mouse cursor in the window, the zooming origin is set, and the "In" and "Out" buttons cause the display to expand and contract horizontally. When the display is zoomed in, the scrollbar along the bottom allows the user to scroll the view back and forth. A number of techniques, such as drawing portions of the display into memory even when they are not visible, smooth the action of the scrolling, even at high zoom factors. "Reset" returns the display to its original configuration. "Options" presents a menu of system configuration options, such as the default zoom factor, "Print" allows for printing a Postscript version of the display, and "Close" exits. One of the options is to show a "mountain range" view of the data, which agglomerates states to show the number of processes in each state at each time. Figure 4 shows a trace from the `cpilog` program distributed with MPICH, with both timeline and mountain range data shown in coupled scrollable windows. This program computes the value of $\pi$ by numerical integration and consists of compute stages intermixed with global operations. It illustrates the use of user-defined states. Currently the mountain range view is not valid for nested states.

## 3.2 The States Subwindow

The lower portion of the window contains buttons associated with states. By grabbing this subwindow's "handle" (the stippled area at the left) with the mouse, this subwindow can be "torn off" and displayed in a separate window. This feature is convenient when there are very many different states, as in the PETSc example in Section 6.

The check-boxes selectively turn on and off the display of state rectangles and arrows. This feature can be important when the display gets complicated. The colored rectangles that are labeled with state names are themselves buttons, each of which brings up a histogram window for that particular state.

## 3.3 Histogram Windows

A histogram window for the `cpi` example is shown in Figure 5.

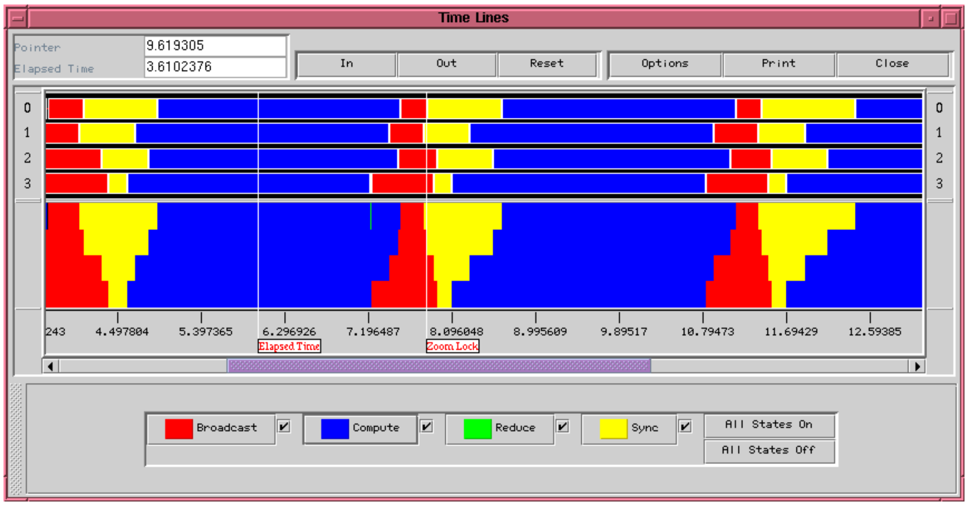


Figure 4: Timeline and mountain range views for the `cpilog` program

The histogram view is, like the timeline view, highly interactive. The panel at the top left describes the total histogram data, such as the range of values and the number of values. The top right panel describes the data that is presented in the histogram window itself. The number of bins for the histogram can be varied with the slider control, which dynamically changes the shape of the histogram. The histogram window can be zoomed and scrolled to select a particular set of state instances, and the "Blink states" button causes those states to blink on and off in the timeline view so that they can be easily located. We have found this to be an effective method for locating specific state instances even in a very busy display.

The most novel feature of this display is the method used to find states that last "too long." It makes the (not always true) assumption that each instance of a particular state should last approximately the same length of time, and therefore the state durations are liable to have a normal distribution. (This can be viewed as an application of the Central Limit Theorem of statistics.) Under this assumption, one can calculate the high and low cutoff that would identify particular percentages of the states durations. If the states truly last nearly all the same length of time, then this method of identifying the top 1%, say, of the states will find none at all. Thus we can attempt to find states with "anomalous" lengths, not just the longest and shortest durations. Clicking on the various percentage buttons selects a particular set of state instances, which easily can be identified in the timeline display when the user clicks on "Blink states".

This technique is admittedly crude. One of our research tasks over the next few years is to develop a sophisticated set of techniques for drawing the user's attention to that part of the computation where tuning can bring performance benefits.

The "Resize to fit" button raises the height of the histogram bars if the selection is such that they have become inconveniently short. The "Selected Regions" button allows one to selectively turn off the blinking of selected sets of state durations, the "Print" button prints the histogram window in Postscript, and "Close" exits.

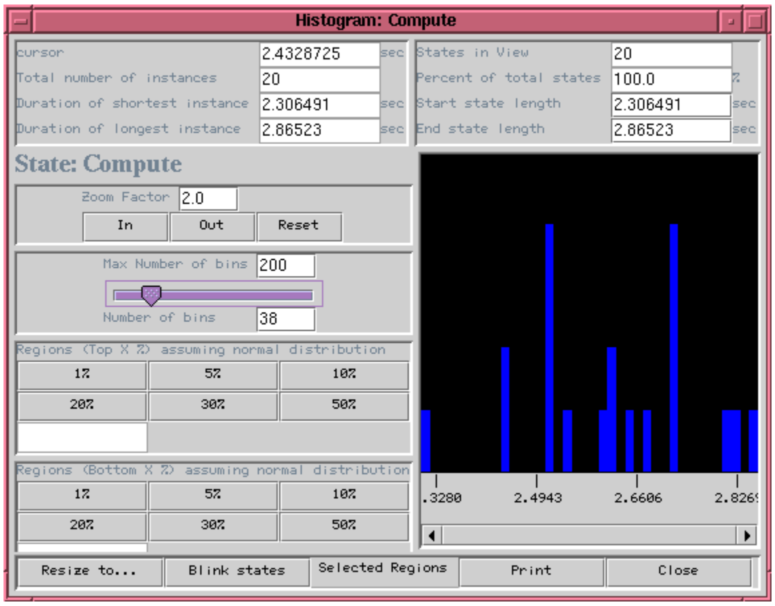


Figure 5: Histogram for the “compute” state

# 4 Producing Logfiles

The ALOG library used to write logfiles to be displayed with upshot and nupshot has served us well over the years, but has a number of limitations. The most significant is its fixed record format. An ALOG record contains 6 integers and a 12-character string. The fixed format means that the records are easy to traverse in memory and to parse from files, but also means that it is cumbersome, if not impossible, to add new fields, particularly fields that pertain to only one type of record. The ASCII format in files is excellent for portability and human readability of the files, but awkward for containing binary data. CLOG is more flexible: it contains a variety of predefined record formats, together with a “raw” record format that can be used for special cases. It is also relatively easy to extend with new record types without impacting the usefulness of existing logfiles.

## 4.1 How It Works

A logfile-creation library for parallel programs has a number of requirements. Meeting these requirements is eased by assuming that MPI is available, and we do make that assumption.

- Logging must be so efficient that it does not materially affect the behavior of the program. This means that I/O should be carried out only after the program finishes.
- It is convenient if at the end of the program there is only one logfile, rather than one for each process.
- Timestamps cannot, in general, be assumed to be synchronized among processes. On some machines this assumption can be made, such as on certain SMPs or if the switch clock on the IBM SP is being used, but in general some postprocessing is needed to synchronize the timestamps.
- The data in the logfile should be self-describing to some extent, for the convenience of the display tool.

CLOG makes a number of compromises in meeting these requirements.

- To obtain a timestamp, CLOG calls `MPI_Wtime`, which returns a floating-point number of seconds since some time in the past. The assumption is that `MPI_Wtime` is reasonably efficient on any MPI implementation, although this particular format may not be the most efficient way to get a timestamp on a given machine. We choose this approach for portability.
- When a log record is to be written, a subset of its content is stored in a relatively large buffer in memory. When this buffer fills up, another one is `malloc`'d from the system. An alternative would be to write the full buffer to disk and reuse the space, but the I/O might perturb the computation. Of course calling `malloc` also causes some perturbations in the computation, but not as much as I/O would. We are planning to make this mechanism scalable to larger log files by periodically dumping to disk.
- At the end of the computation, the buffers are processed to add information that is the same for all records in the buffer (process id, for example). At this time the timestamps are adjusted. A relatively straightforward algorithm is used to find relative clock offsets. Process 0 requests a local time value from each of the other processes, and assumes that other processes read their own clocks midway between the sending of this request and receiving the answer. The request is repeated several times, and the result with lowest error (shortest round-trip time) is used. More accurate but more complicated algorithms are known; our algorithm corrects only for displacement, not for dilation, but so far it has proved adequate. As a last resort, Jumpshot itself has a method for fine tuning the displacement of the events in a given process at display time by dragging an individual time line.

At the end of the run, all processes participate in merging the records by timestamp to create a single logfile. After the last log record is written and postprocessing of local buffers is complete, each process has a linked list of buffers, each containing a large number of log records. The processes form themselves into a binary tree, with (MPI) Process 0 at the root. The processes at the leaves begin by sending their buffers to their parents. Each nonleaf process performs a three-way merge of its own buffer with the buffers arriving from its children. When a merged buffer has been filled, it is sent to its parent. At the root, merged, filled buffers are written to the logfile. MPI is used for all communication. The file is in MPI's "external-32" portable format [7], which is the same format used by Java for portability of files, and which makes it portable, but requires byte-swapping on some machines for some fields in the log records.

The post mortem processing, although it makes collection of the logfiles less intrusive, makes jumpshot less useful for debugging, since the logfile is only fully assembled when the program terminates normally. This underscores the fact that Jumpshot is a *performance* debugging tool rather than a *correctness* debugging tool.

## 4.2 The MPI Profiling Interface

The easiest way to write CLOG logfiles is automatically, using the MPI profiling interface. (See Section 8.1 of [20] and Section 7.6 of [10].) The MPI standard specifies a mechanism by which all MPI calls may be intercepted by the user, who can define a *profiling* library containing his own versions of the MPI functions. Distributed with the MPICH portable implementation of MPI [6, 9] is such a profiling library to write CLOG records, logging the start time and end time of all MPI calls, thus logging a state for Jumpshot for the time a process is executing an MPI call. The profiling version of `MPI_Finalize` causes the local logs to be merged and the logfile to be written as described in Section 4.1. In order to cause this to happen, one need only link the profiling library in front of the MPI library one is using. In MPICH, this is conveniently accomplished by using MPICH scripts for linking, which have flags to link in various profiling libraries. For example,

```
mpicc -mpilog -o myprog myprog.c
```

will cause the profiling library for CLOG logging to be automatically linked in. When `myprog` is run with

```
mpirun -np 64 myprog
```

a CLOG logfile will automatically be produced, suitable for viewing with Jumpshot.

The `MPI_Pcontrol` function, when used with this profiling library, can be used to turn logging on and off dynamically in the program. This can be used, for example, to collect logging data for a single iteration of a complex loop that is executed many times.

### 4.3 The MPE Interface

For greater control, a user may log events directly, using the MPE library. MPE comprises useful tools that forms a companion library for enhancing MPI programs. It was originally designed for use with MPICH, but now is independent of MPICH and can be used with any implementation of MPI. It contains non-MPI synchronization functions (for serialization), graphics routines (see [8]), and logging functions. It is these logging routines that are called by the profiling library described in Section 4.2, rather than CLOG directly.

The usefulness of using this library to prepare logfiles for Jumpshot is that a user (or another library) can log events and states that are directly related to the application or library independently of the MPI calls made. One can use the MPE logging functions either without any automated logging at all or in conjunction with it. A large-scale example of use with a library is given in Section 6.4.

### 4.4 The CLOG Interface

The lowest level way to create logfiles is with the CLOG interface, which logs the various event types and manages the activities described in Section 4.1. The change from ALOG to CLOG logfile formats was accomplished by re-implementing the MPE logging library in terms of CLOG calls, without changing the MPE interface. In addition, the transition from ALOG format to CLOG format has been eased for MPICH users by providing a CLOG-to-ALOG translator, so that the older tools, upshot and nupshot, can still be used with the newer CLOG file format. This was important for the period between the adoption of the CLOG format and the implementation of Jumpshot.

## 5 Experiences with Java

Using Java for this fourth generation of our basic performance analysis tool was an experiment. The first part of the experiment was to determine whether the graphics performance of Java would be adequate. Although Jumpshot does not display complex views as quickly as did nupshot ( with its direct X interface) we find it adequate for our purposes. The increasing significance of Java is likely to cause performance to improve over time as well. The functionality of the graphics interface, considered against Jumpshot's rather modest needs, is also adequate, and also getting better.

The second, nontechnical, part of the experiment was to determine whether the only really new benefit conferred on the project by the use of Java, namely the ability for Jumpshot to run as a Web application, would be worth the effort. This capability turned out to be more useful than we anticipated, for two reasons. First, running Jumpshot from the Web browser on one's laptop connected to the network turned out to be useful for demonstrations and lectures, as well as in allowing others to try it over the net. (The Jumpshot applet can be viewed at `http://www.mcs.anl.gov/~lusk/aptest/lib/jumpshot`.) Second, the environment portability problems described below are greatly ameliorated by using the Web interface, with no loss in performance.

The main drawback to the use of Java was the lack of portability of Java environments. We use the JDK (Java Development Kit) for portability. Although the Jumpshot program's Java code itself is portable, the way Java is locally installed on Sun, SGI, and IBM workstations, and under Windows95 or NT, can be critical. Running Java programs typically involves setting certain environment variables, which can conflict with settings required for other applications (e.g., Web browsers). Since we would like users to use all our tools in the same way in all environments, this can be frustrating. Again, we expect the situation to improve over time.

Java libraries are rapidly evolving, and this caused considerable discomfort as JDK 1.1.2 evolved into 1.1.6, which is the level Jumpshot currently requires. With the acceptance of the Swing GUI, which Jumpshot uses now, and the upcoming release of JDK 1.2, we expect these problems to lessen.

At any rate, the overall verdict on the use of Java for this project is positive, and we expect to continue to use it as Jumpshot evolves.

## 6 Interesting Examples

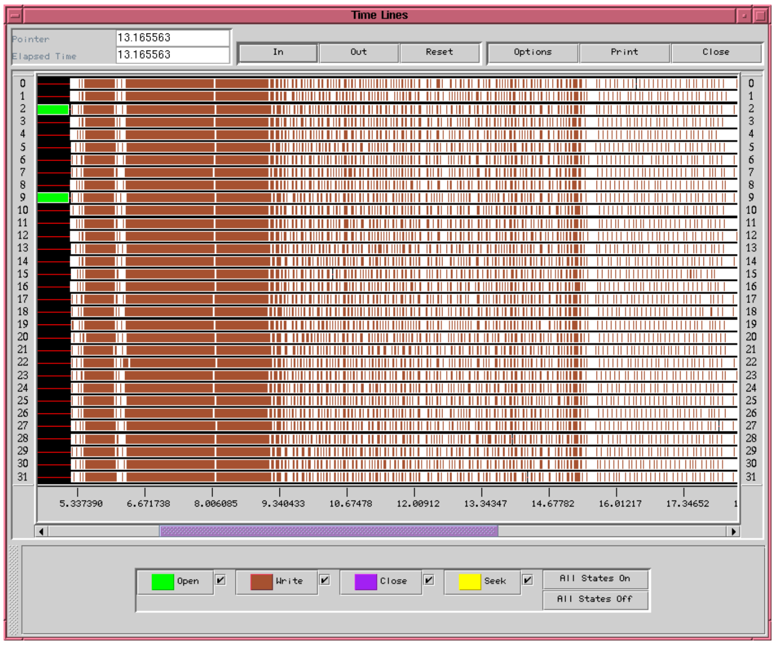


Figure 6: MPI-2 independent (Unix-like) writes

In this section we describe and illustrate the use of Jumpshot with more interesting examples.

## 6.1 Parallel I/O

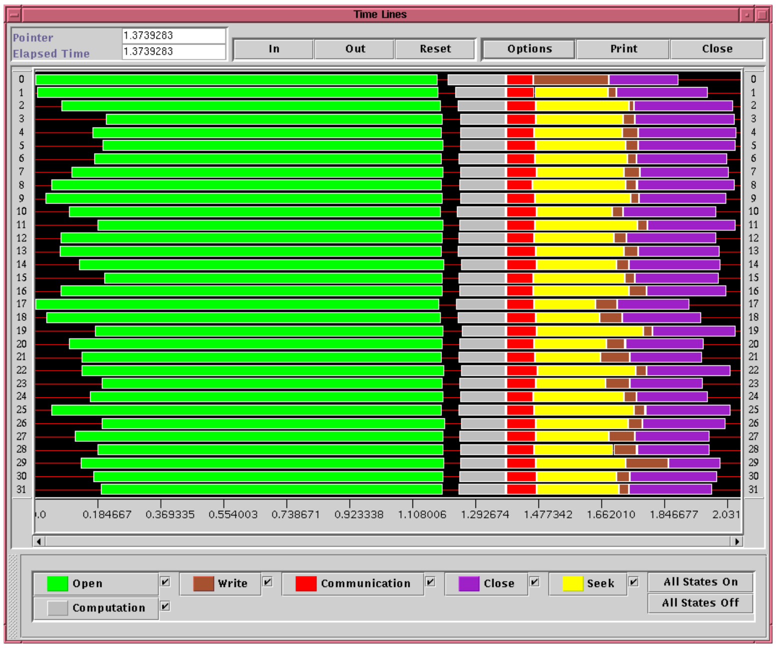


Figure 7: MPI-2 collective writes

In lectures encouraging people to use the new MPI-2 Standard for parallel I/O (see Chapter 7 of [7]), we find it useful to compare independent I/O with collective I/O on the same problem of writing a (block,block)-distributed array to disk. The many small writes done by the independent writes, which mimic the Unix interface, are illustrated in Figure 6. The many individual writes can be seen (the white sections are just the borders of small writes close together; they can be resolved by zooming in further). The same problem carried out with the MPI-2 collective I/O operations show how small communications and computational overheads can be traded for large amounts of I/O overhead (see Figure 7. Figure 6 is a zoomed-in view; the elapsed time for the entire I/O operation is about 27 seconds. In Figure 7 the elapsed time can be seen to be about 2 seconds, and more than half of this time can be seen to be taken up by the `open` operation. These runs were done on an Intel Paragon.

## 6.2 Beowulf Structure Revealed

Figure 8 demonstrates the phenomenon that viewing a parallel execution with tools like Jumpshot almost always reveals something unexpected. In this case we were conducting a simple test of a medium-sized Beowulf cluster with the `ringtest` program of Section 3. Here the program is being run on 64 processors, with one process on each. State display has been turned off to better show the paths of the messages. What is interesting is that the messages consistently take just a little longer to get from process 31 to 32 and from 47 to 48. We conjecture that what we are seeing is the hierarchical structure of the Fast Ethernet switch at Caltech, which is connecting groups of 16 machines. We don't know why the jump from 15 to 16 is not as noticeable as the others.

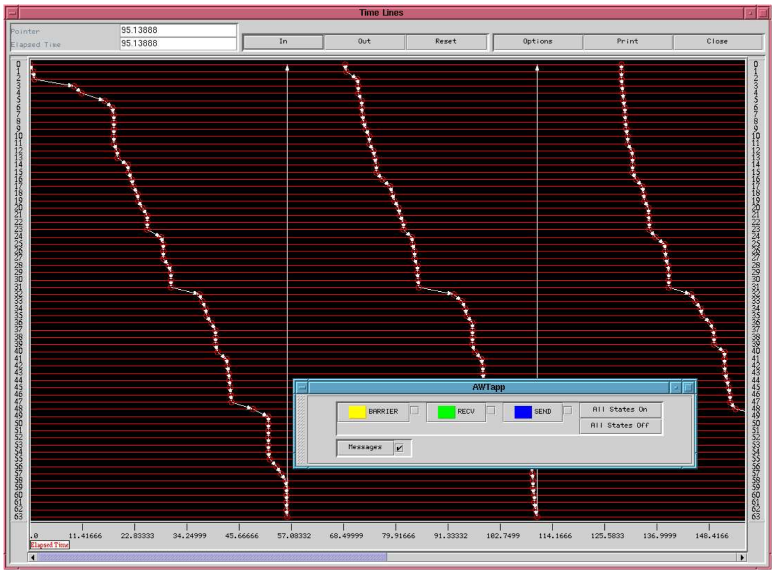


Figure 8: Ringtest on a Beowulf

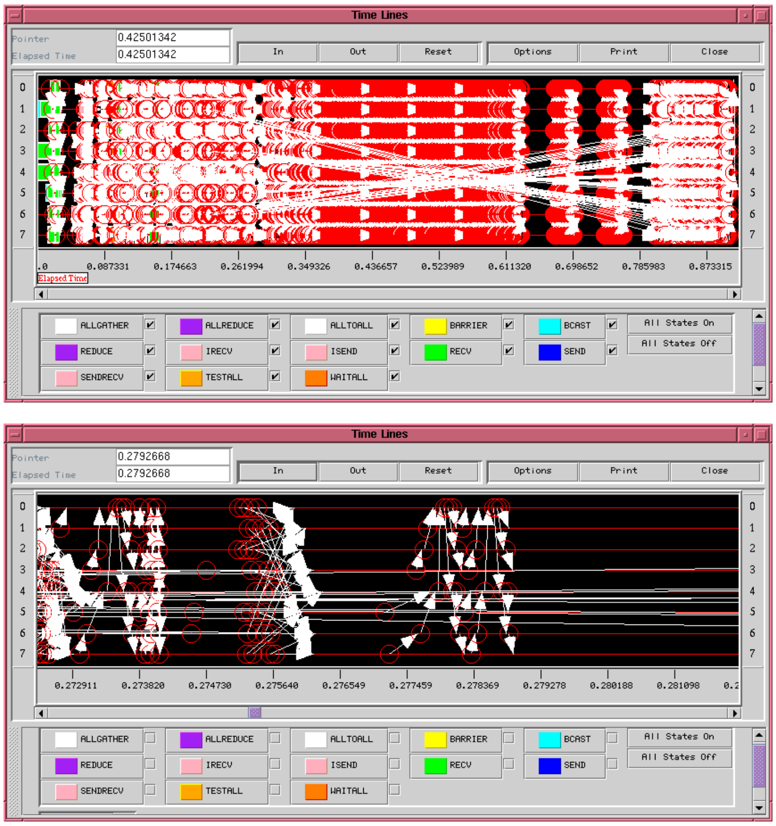


Figure 9: Initial and zoomed views from ASTRO3D

### 6.3 An Astrophysics Application

Work on Jumpshot is partially supported by the ASCI Center on Astrophysical Thermonuclear Flashes at the University of Chicago. The initial modeling code there is Andrea Malagoli's ASTRO3D. Jumpshot is being used to analyze its performance. An initial trial run is illustrated in Figure 9. The complexity of this picture, which just shows one iteration and 8 processes, whereas ASTRO3D is routinely run for many iterations on hundreds of processes, illustrates the need for continued work on advanced analysis features in support of scalability as log files become large.

### 6.4 Instrumenting a Mathematical Library


Figure 10: A trace from PETSc

The PETSc library [1] for the solution of partial differential equations has been instrumented by its authors to show all library routine calls. This enables them to better understand the relative effect on performance of various parts of the PETSc library. In Figure 10 one can see the nested states corresponding to the nested subroutine calls. Only a few states (PETSc functions) are visible in the states window, although the horizontal and vertical scrollbars for this window can be used to access the others. This is a case where the ability to tear off this window and display it in full size is useful.

## 7 Directions for Future Work

Jumpshot is still evolving. We are working on enhancements in a number of areas.

- We would like to bring Jumpshot closer to completely satisfying the requirements in the list in Sections 2.2 and 4.1. It meets most of these goals, as can be seen by the features discussed in this paper, but others are either missing or incomplete. In the near term we will be adding support for filtering by MPI communicator. The desirable feature of tracking events back to source code is probably much further away, because of the difficulty of achieving this in a portable way. The problems are more with capturing the information than with displaying it.

- Although Jumpshot and CLOG represent advances over upshot, nupshot, and ALOG in terms of handling large log files and complex displays, there is still much more to do. A step in this direction is a collaboration with IBM Research to develop a scalable logfile format (SLOG), capable of dealing with (and helping Jumpshot deal with) multi-gigabyte logfiles.
- The identification of states that last "too long" relies for the present on simple statistical measures such as the standard deviation. We would like to employ more sophisticated statistical techniques to identify "anomalous" parts of the run, as well as more sophisticated models.
- The current collection of displays have served us well, but as data volume increase, more elaborate visualization techniques will be needed. This is a research topic for the (near) future.

# References


[1] PETSc 2.0 for MPI. `http://www.mcs.anl.gov/petsc/petsc.html`.

[2] TimeScan Multiprocess Event Analyzer. `http://www.dolphinics.com/tw/ts30ann.htm`.

[3] Vampir 2.0 – Visualization and Analysis of MPI Programs. `http://www.pallas.de/pages/vampir.htm`.

[4] XMPI Web page. `http://www.mpi.nd.edu/lam/xmpi/xmpi21.html`.

[5] *The Random House Dictionary of the English Language* (unabriged edition). Random House, New York, 1971.

[6] W. Gropp and E. Lusk. Sowing MPICH: A case study in the dissemination of a portable environment for parallel scientific computing. *Journal of Supercomputer Applications and High-Performance Computing*, 11(2):103–114, 1997.

[7] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI—the Complete Reference: Volume 2, The MPI Extensions*. MIT Press, 1998.

[8] W. Gropp, E. Karrels, and E. Lusk. MPE graphics—scalable X11 graphics in MPI. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 49–54. IEEE Computer Society Press, 1994.

[9] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message-passing interface standard. *Parallel Computing*, 22:789–828, 1996.

[10] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[11] M. T. Heath and J. E. Finger. Paragraph: A performance visualization tool for MPI. `http://www.csar.uiuc.edu/software/paragraph`.

[12] M. T. Heath and J. E. Finger. Visualizing performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

[13] V. Herrarte and E. Lusk. Studying Parallel Program Behavior with Upshot. Technical Report ANL–91/15, Mathematics and Computer Science Division, Argonne National Laboratory, August 1991.

[14] E. Karrels and E. Lusk. Performance analysis of MPI programs. In Jack J. Dongarra and Bernard Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, pages 195–200. SIAM, 1994.

[15] J. A. Kohl and G. A. Geist. The PVM 3.4 Tracing Facility and XPVM 1.1. In *Proceedings HICSS-29: Twenty-Nineth Hawaii International Conference of System Sciences*, January 1996.

[16] E. Lusk, W. McCune, and J. Slaney. ROO—a parallel theorem prover. Technical Report MCS–TM–149, Argonne National Laboratory, 1991.

[17] A. Malony, D. Hammerslag, and D. Jablonowski. TraceView: A trace visualization tool. *IEEE-Software*, 8(5):19–28, September 1991.

[18] J. K. Osterhout. *Tcl and the Tk toolkit.* Addison-Wesley, 1994.

[19] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. Schwartz, and L. F. Tavera. Scalable performance analysis: the Pablo performance analysis environment. In A. Skjellum, editor, *Proceedings of the Scalable Parallel Libraries Conference.* IEEE Computer Society, 1993.

[20] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI—the Complete Reference: Volume 1, The MPI Core (second edition).* MIT Press, 1998.