ParaGraph: A Performance Visualization Tool for MPI *

Michael T. Heath, University of Illinois
Jennifer E. Finger, University of Tennessee

August 31, 2003

Abstract

ParaGraph is a graphical display tool for visualizing the behavior and performance of parallel programs that use MPI (Message-Passing Interface). The visual animation of a parallel program is based on execution trace information gathered during an actual run of the program on a message-passing parallel computer system. The resulting trace data are replayed pictorially to provide a dynamic depiction of the behavior of the parallel program, as well as graphical summaries of its overall performance. The same performance data can be viewed from many different visual perspectives to gain insights that might be missed by any single view. We describe the visualization tool, discuss the motivation and philosophy behind its design, and provide details on how to use it.

*Research supported by the U.S. Department of Energy through the University of California under subcontract number B341494.

Contents

1 Motivation and Design Philosophy

Graphical visualization is a standard technique for facilitating human comprehension of complex phenomena and large volumes of data. The behavior of parallel programs is often extremely complex, and hardware or software performance monitoring of such programs can generate vast quantities of data. Thus, 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. ParaGraph provides detailed, dynamic, graphical animation, as well as graphical performance summaries, of parallel programs that use MPI (Message-Passing Interface) [28, 33]. The main purpose of this document is to explain how to use this visualization tool to analyze the behavior and performance parallel programs.

1.1 Visual Playback

We will use the word "playback" to refer to the graphical animation of a parallel program. Just as in audio or video recording, ParaGraph provides a playback, or reenactment, of events that were recorded while the program under study executed on a real parallel computer system. Although ParaGraph is generally used only in such a "post processing" manner, using a tracefile created during the execution of the parallel program and saved for later study, the data for the visualization could in principle be arriving at the graphical workstation as the parallel program executes on the parallel machine.

In practice, however, there are major impediments to such real-time performance visualization. With most parallel architectures, it is difficult to extract performance data from the processors and send it to the outside world during execution without significantly perturbing the application program being monitored. Moreover, the network bandwidth between the parallel system 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 high-resolution 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. One of the strengths of ParaGraph is the insight that can be gained from repeated replays, often in slow motion, of the same execution trace data, much the way "instant" replays are used in televised sports events.

Program visualization can be thought of in either static or dynamic terms. After a parallel program has completed execution, the tracefile of events saved on disk can be considered as a static, immutable object to be studied by various analytical or statistical means. Some performance visualization packages reflect this philosophy in that they provide graphical tools designed for visual browsing of the performance data from various perspectives using scrollbars and the like. In designing ParaGraph, we have adopted a more dynamic approach whose conceptual basis is algorithm animation [1]. We see the tracefile as a script to be played out, visually reenacting the original live action of parallel program execution in order to provide insight into the program's dynamic behavior. There are advantages and disadvantages in both static and dynamic approaches. Algorithm animation is good at capturing a sense of motion and change, but it is difficult to control the apparent speed of playback. The static "browser with scrollbars" approach, on the other hand, gives the user control over the speed with which the data are viewed (indeed, "time" can even move back-

ward), but does not provide such an intuitive feeling for the dynamic behavior of parallel programs. In designing ParaGraph, we have opted for the dynamic animation approach, sacrificing some control over playback speed (as will be discussed later in greater detail).

1.2 Design Goals

Our principal goals in designing ParaGraph were:

- Ease of understanding
- Ease of use
- Portability

We now briefly discuss each of these goals in turn.

Ease of understanding. Since the whole point of visualization is to facilitate human understanding, it is imperative that the visual displays provided be as intuitively meaningful as possible. The charts and diagrams should be aesthetically appealing, and the information they convey should be as self-evident as possible. A diagram is not likely to be helpful if it requires an extensive explanation. The type of information conveyed by a diagram should be immediately obvious, or at least easily remembered once learned. The choice of colors used should take advantage of existing conventions to reinforce the meaning of graphical objects, and should also be consistent across views. Above all, it is essential to provide many 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. ParaGraph provides some thirty different displays or views, all based on the same underlying execution trace data.

Ease of use. One of the main purposes of software tools is to relieve tedium, not promote it. Through the use of color and animation, we have tried to make ParaGraph painless, perhaps even entertaining, to use. ParaGraph has an interactive, mouse- and menu-oriented user interface so that the various features of the package are easily invoked and customized. Another important factor in ease of use is that the user's parallel program (the object under study) need not be modified extensively to obtain the data on which the visualization is based. The execution trace data required by ParaGraph are provided by a tracing package that takes advantage of the profiling interface of MPI [33] to produce the necessary data with little or no user intervention. We have also tried to keep the user's learning curve for ParaGraph very easy, even at the expense of some limitations on the flexibility of its data processing and graphical display capabilities. Our intent is to require minimal data manipulation and to provide a variety of views that are customized to the task at hand, rather than providing more general data processing and graphics capabilities in a toolkit for constructing views of program behavior. The latter approach provides greater flexibility, but requires far more effort on the part of the user.

Portability. ParaGraph runs on any workstation that supports the X Window System. ParaGraph does not require any X toolkit or widget set, as it is based directly on the standard Xlib library, which is available in any distribution of the X Window System.

Although ParaGraph is most effective in color, it also works on monochrome and grayscale monitors. It automatically detects which type of monitor is in use, but this automatic choice can be overridden by the user, if desired. ParaGraph is also applicable to any MPI program, regardless of the particular parallel computer system or programming language that may have been used by the programmer.

1.3 Development History

Graphical visualization of algorithm behavior is not a new idea. Graphical animation techniques for visualizing serial algorithms began to receive serious attention in the early 1980s [1], and similar approaches to visualizing parallel computations were under development by the late 1980s. For a general overview of the field, see [2, 10, 16, 17, 18, 32, 35]. Early work that inspired or influenced the initial development of ParaGraph includes [3, 20, 21, 24, 25, 26, 31, 34].

The original version of ParaGraph was developed during the period 1989–1991 [11, 12, 13]; its further evolution is described in [14, 15]. ParaGraph was initially based on PICL (Portable Instrumented Communication Library) [7, 8], which runs on numerous distributed-memory parallel systems and automatically provides the execution trace data required by ParaGraph. Upon their initial release in 1991, ParaGraph and PICL were made freely available through `netlib` [4] and were used by hundreds of users worldwide. ParaGraph was also picked up by several vendors of distributed-memory parallel computers, including Intel and Ncube, for distribution with their parallel systems. For some case studies of performance tuning with ParaGraph, see [11, 14, 18, 36, 39].

As its name suggests, one of the original motivations for PICL was the lack of uniformity in vendor-supplied communication libraries. PICL provided a common syntax for message passing so that the resulting parallel program is portable across many parallel systems. PICL runs on top of whatever native communication library is provided by the system vendor. In addition to providing portability, this extra software layer also enables communication events to be monitored and recorded in a tracefile for subsequent study using a tool such as ParaGraph.

With the establishment of the MPI standard in 1994 [28] and its ultimate acceptance by all major vendors of parallel computer systems, there is no longer a need for the portability features provided by PICL, as most message-passing parallel programs are now written using MPI, which motivates the adaptation of ParaGraph to support MPI directly. Fortunately, the profiling interface of MPI enabled the development of MPICL, a tracing facility for MPI analogous to that provided by PICL, with a similar data format to which ParaGraph could easily be adapted. Much less straightforward was accommodating the huge number of functions and the multiplicity of message-passing protocols provided by MPI, compared with the relative simplicity of PICL. Nevertheless, ParaGraph has been adapted to deal with essentially the full functionality of MPI prescribed by the MPI Standard [28, 33], while remaining backward compatible with the simpler PICL message-passing model and the tracefiles it generates. ParaGraph does not yet support some of the new functions added in the MPI2 Standard [29].

1.4 Relationship to MPICL

MPICL is a package developed by Patrick Worley of Oak Ridge National Laboratory to provide PICL-style instrumentation [7, 8, 38] for parallel programs using MPI. It is freely available from `www.msr.epm.ornl.gov` and also from `www.netlib.org`. In principle, MPICL should work with any correct implementation of MPI, although modifications may be required in some instances to deal with machine-specific features such as clocks, I/O, string handling, and C/Fortran calling conventions. Versions for some specific platforms are available in the standard distribution.

MPICL takes advantage of the profiling interface of MPI to create a tracefile that records all invocations of MPI functions (e.g., sending and receiving messages) in the execution of the user's parallel program. The tracefile contains one event record per line, and each event record consists of a set of numerical values specifying the event type, timestamp, processor number, message length, and other similar information. The format of the tracefile is documented in detail in [38]; see Appendix D below for a summary. ParaGraph has a producer-consumer relationship with MPICL: ParaGraph consumes trace data produced by MPICL. By using MPICL in conjunction with MPI, the user can create the data files necessary to use ParaGraph to analyze the behavior and performance of parallel programs.

For a meaningful playback, the timestamps of the events should be as accurate and consistent across processors as possible. This is not necessarily easy to accomplish when each processor has its own clock with its own starting time and running at its own rate. Moreover, the resolution of the clock may be inadequate to resolve events precisely. Poor resolution and/or poor synchronization of the processor clocks can lead to "tachyons" in the tracefile, that is, messages that appear to be received before they are sent. Such an occurrence will confuse ParaGraph, since much of its logic depends on correctly pairing sends and receives, and will invalidate the information in some of the displays. For this reason, MPICL goes to considerable lengths to synchronize processor clocks and adjust for clock drift [5], so that the timestamps will be as consistent and meaningful as possible.

Another important issue is the amount of additional overhead introduced by the collection of trace information compared to the execution time of an equivalent uninstrumented program. MPICL tries to minimize the perturbation due to tracing by saving the trace data locally in each processor's memory, then downloading it to disk only after the program has finished execution. Nevertheless, such monitoring inevitably introduces some extra overhead; in MPICL this overhead adds a fixed amount (independent of message size) to the cost of sending each message. The overall perturbation is thus a function of the frequency and volume of communication traffic, and it also varies from machine to machine. This perturbation is usually small enough that the behavior of parallel programs is not fundamentally altered. In any case, in our experience the lessons we learn from visual study of instrumented runs invariably lead to improved performance of uninstrumented runs as well.

2 Programming Model

The parallel programming model supported by MPI—and by ParaGraph—is that of multiple processes communicating with each other by sending and receiving messages. Such a paradigm is most natural for distributed-memory multicomputers and for networks of work-

stations, but it can also be implemented very efficiently on shared-memory multiprocessors as well. The message-passing paradigm is extremely flexible and permits programmers the fine control over data locality needed to optimize parallel performance. Because of its universality and low-level control, message-passing is sometimes called the "assembly language" of parallel computing. However, as with most low-level paradigms that closely reflect the underlying hardware, explicitly parallel programming with message passing can be tedious and error prone. For a tutorial introduction to MPI, see [9, 30]; for comprehensive coverage, see [28, 33].

Consistent with the limited communication model provided by PICL, the original version of ParaGraph [13] supported a single style of message-passing in which sends are nonblocking (i.e., the sending process resumes execution immediately) and receives are blocking (i.e., the receiving process suspends execution until the requested message arrives). MPI greatly expands this repertoire by supporting several communication modes (standard, buffered, synchronous, and ready) and providing both blocking and nonblocking versions of both send and receive. Supporting all of these options required considerable generalization of ParaGraph. For example, many different types of events in MPI may cause processor idle time, whereas in the previous paradigm, processor idle could be caused only by a blocked receive.

Although the trace data format used by MPICL for a given event record includes separate fields for the process number and the processor number at which the event occurred, both MPICL and ParaGraph currently support only one process per processor. (Specifically, MPICL currently uses the rank in `MPI_COMM_WORLD` as the processor number and the process number is always zero. ParaGraph uses the processor number from each event record and ignores the process number.) Such a programming style is typical of SPMD (Single Program Multiple Data) programs for multicomputers and is adequate for most applications on these machines. Nevertheless, this restriction is sometimes limiting, and may be relaxed in future releases of MPICL and ParaGraph (anticipated future generalization is why separate fields for the two entities were provided from the outset). In a shared-memory architecture, the affinity between processes and processors is often less clear, but we still use the MPI rank of a process as its "processor number." In any case, the terms "process,", "processor," and "node" are used interchangeably in ParaGraph and in this document. Some parallel systems have a distinguished *host* process `MPI_HOST`, but ParaGraph does not depict such a host graphically and ignores all trace events involving the host, if any.

In MPI, each point-to-point message carries an integer tag. The value for the tag is assigned by the sending process and can be used by the receiving process to search the queue of incoming messages selectively for a particular type of message. Such tags can play an important role in synchronization and control of the parallel program. In some of ParaGraph's communication displays, communication events can be optionally color coded by the values of their message tags.

Another innovation in MPI is the ability to define a subgroup of processes and collective communication operations among them, such as broadcast, reduction, and all-to-all exchanges. Since ParaGraph has no knowledge of the underlying implementation of such collective communication operations, its depicts them *as if* individual point-to-point messages were sent from each sender to each receiver. This approach is consistent with the semantics for collective communication stated in the MPI standard. In the context of

collective communication, it is important to note that the "processor number" generated by MPICL and used by ParaGraph is always the "global number" given by the rank in `MPI_COMM_WORLD` rather than the relative rank in the particular group involved. Thus, the processor number used by ParaGraph is always unique and unambiguous.

3 Software Design

ParaGraph is an interactive, event-driven program. Its basic structure is that of an event loop and a large switch that selects actions based on the nature of each event. There are two separate event queues: a queue of X events produced by the user (mouse clicks, keypresses, window exposures, etc.) and a queue of trace events produced by the parallel program under study. ParaGraph alternates between these two queues to provide both a dynamic depiction of the parallel program and responsive interaction with the user. Menu selections (and optional initialization files) determine the execution behavior of ParaGraph, both statically (initial selection of displays, options, parameter values) and dynamically (pause/resume, single-step mode, etc.).

ParaGraph is written in C, and the source code contains about 22,000 lines. The `main` program of ParaGraph calls the `preprocess` function to determine necessary parameters, initializes many variables, allocates graphical resources such as windows and fonts, and then goes into a `while` loop that repeatedly calls the functions `get_event` and `get_trace`, which check the X event queue and the trace event queue, respectively, for the next event upon which to act. The `get_event` routine is simply a switch containing a series of calls to appropriate routines to handle the various X events. The `get_trace` routine calls `scan` to read a trace event record, and then calls `draw` to update the drawing of the displays that have been selected.

The X event queue must be checked frequently enough to provide good interactive responsiveness, but not so frequently as to degrade the drawing speed during playback. On the other hand, the trace event queue should be processed as rapidly as possible while playback is active, but need not be checked at all if the next possible event must be an X event (e.g., before playback starts, after playback finishes, when in single-step mode, or when playback has been paused and can be resumed only by user input). To address these issues, the alternation between the two queues is not strict. Since not all trace event records produced by MPICL are of interest to ParaGraph, it "fast forwards" through any series of such "uninteresting" records before rechecking the X event queue. Moreover, both blocking and nonblocking calls are used to check the X event queue, depending on the circumstances, so that workstation resources are not consumed unnecessarily when playback is inactive.

The relationship between the apparent playback speed and the original execution speed of the parallel program is necessarily somewhat imprecise. The speed of graphical playback is determined primarily by the drawing speed of the workstation, which in turn is a function of the number and complexity of displays that have been selected. There is no way, in general, to make the apparent playback speed uniformly proportional to the original execution speed of the parallel program. The reason is that the time required to compute some event on the parallel machine and the time required to draw a graphical depiction of that event on the workstation screen are not necessarily correlated. For the most part, ParaGraph simply processes the event records and draws the resulting displays as rapidly as

it can. If there are gaps between consecutive timestamps, however, the intervening time is "filled in" by a spin loop so that there is at least a rough (but not uniform) correspondence between playback time and original execution time. Fortunately, this issue does not seem to be of critical importance in visual performance analysis. The most important consideration in understanding parallel program behavior is simply that the correct relative order of events be preserved in the graphical replay. Moreover, the figures of merit produced by ParaGraph are based on the actual timestamps, not the apparent speed with which the playback unfolds.

Since ParaGraph's speed of execution is determined primarily by the drawing speed of the workstation, it can be slowed down or speeded up by selecting more or fewer displays, and by the options used within those displays (e.g., jump vs. smooth scrolling). For a given fixed configuration of displays and options, there is no way to speed up the playback, since it is already drawing as fast as it can. If slower playback is desired, however, a "slow-motion" slider is provided for precise control over playback speed. Finally, one can use single-step mode for detailed, step-by-step study of program behavior.

4 Using ParaGraph

ParaGraph supports the following command-line options:

- `-c` to specify color display mode
- `-d` to specify a hostname and screen (e.g., `hostname:0.0`) for remote display across a network (alternatively, the `DISPLAY` environment variable can be set appropriately)
- `-e` to specify an initialization file (default: `.pgrc`)
- `-f` (or no switch) to specify the pathname of a tracefile directory or of an individual tracefile
- `-g` to specify grayscale display mode
- `-l` to specify an animation layout file (default: `.pganim`)
- `-m` to specify monochrome display mode
- `-n` to specify a name for the base window (default: `ParaGraph`)
- `-o` to specify a file defining an alternate order and/or optional names for the processors (default: `.pgorder`)
- `-r` to specify a file of RGB color values for use in color-coding user-defined tasks (default: `.pgcolors`)

It is normally unnecessary to specify the display mode (color, grayscale, or monochrome), as ParaGraph by default detects the most appropriate choice for the workstation in use. Overriding this automatic choice of display mode can be useful, however, for making black-and-white hardcopies from a color screen or to accommodate workstations with multiple screens of different types. The initialization file, if present, defines the initial selection of displays and options with which ParaGraph begins execution. Typically, such an initialization file is created and saved during a previous ParaGraph session. Specifying a unique name for the base window (i.e., the main menu window) is useful for distinguishing among multiple

instances of ParaGraph when running them simultaneously. These and other options are explained in greater detail below.

The tracefile can be specified on the command line, or it can be selected using the `tracefile` submenu available from the main menu. The tracefile directory can be specified on the command line, or it can be entered (or changed) during execution by typing the pathname in the subwindow provided in the `tracefile` menu. If the pathname of a tracefile is specified on the command line, then the directory portion of that pathname is taken as the tracefile directory.

Once a tracefile has been selected, ParaGraph preprocesses the tracefile to determine relevant parameters automatically (e.g., time scale, number of processors) before graphical playback begins; most of these values can be overridden by the user, if desired, by using the `options` menu. Faulty tracefiles are usually detected during the preprocessing stage, in which case ParaGraph issues an error message and terminates before going into graphical playback. To produce the necessary trace records for use in ParaGraph, tracing in MPICL should be done with `tracelevel(1,1,0)`. For graphical animation with ParaGraph, the events in the tracefile must be sorted into increasing time order, whereas MPICL produces a tracefile with records in node order. The necessary reordering can be accomplished with the Unix sort command:

```
% sort +2n -3 +1rn -2 +0rn -1 tracefile.raw > tracefile.sorted
```

By default, ParaGraph initially displays only its main menu, which contains buttons for controlling execution and for selecting various additional menus. The submenus available include those for three types, or families, of displays (`utilization`, `communication`, and `tasks`), an `other` menu of miscellaneous additional displays, a `tracefile` menu for selecting a tracefile, an `options` menu for specifying various options and parameters, and a `record options` menu for selecting displays that are to produce numerical output to files, if desired. As many displays can be selected as will fit on the screen; the displays can be resized within reasonable bounds. Although it is difficult to pay close attention to many displays at once, it is still useful to have several available simultaneously for comparison and selective scrutiny with repeated replays.

Many of the displays have various options that can be selected by clicking on an appropriate subwindow. Pressing the right or middle mouse button cycles forward through the choices, while pressing the left mouse button cycles backward. The selection of displays, their sizes, their locations on the screen, and the options in effect can be saved in an initialization file for use in subsequent ParaGraph sessions, as explained below.

The `tracefile` menu provides a browser for selecting a desired tracefile. If a directory pathname is supplied on the command line when invoking ParaGraph, then it will appear in the `pathname` subwindow; otherwise, the current working directory when ParaGraph is invoked is taken as the default directory. A new pathname can be typed into the `pathname` subwindow of the `tracefile` menu at any time. The filenames in the given directory are displayed (in some cases, the window may need to be resized in order to see all of the filenames), and the user selects the desired tracefile (or another directory) by clicking on the corresponding name. If the name selected is a directory, then it becomes the new tracefile directory and the files it contains are displayed. If the name selected is a tracefile,

then the filename is highlighted in reverse video, and the corresponding tracefile is processed by ParaGraph. A new tracefile can be selected at any time simply by clicking again on a different filename. Note that the directory may contain the names of files that are not in fact legitimate tracefiles. It is the responsibility of the user to select only valid tracefiles for processing by ParaGraph. You may wish to adopt some standard filename suffix, such as `.trf`, to help distinguish tracefiles from other files. To provide greater selectivity in listing filenames, a `pattern` subwindow is provided that supports the wildcard characters `*`, which stands for any string, and `?`, which stands for any single character. The pattern can be changed by typing in the `pattern` subwindow. Only those filenames in the current tracefile directory that fit the given pattern are displayed for possible selection. Typical patterns might be `*.trf` or `run??`.

After selecting the desired displays, options, and tracefile, the user presses `start` to begin graphical playback of the parallel program based on the tracefile specified. The animation proceeds straight through to the end of the tracefile, but it can be interrupted for detailed study by use of the `pause/resume` button. Repeated use of this button alternates between pausing and resuming playback. For even more detailed study, the `step` button provides a single-step mode that processes the tracefile one event (or a user-specified number of events) at a time. A particular time interval can be singled out for study by specifying starting and stopping times (the defaults are the beginning and end of the tracefile), or playback can be optionally stopped each time a user-specified event occurs in the tracefile. The entire animation can be restarted from the beginning at any time (whether in the middle or at the end of the tracefile) by pressing the `start` button again. Most of the displays show program behavior dynamically as individual events occur, but some show only overall summary information at the end of the run (a few displays serve both purposes, as will be discussed below).

The `slow motion` button opens a window containing a "slider" for controlling playback speed. Clicking or dragging the mouse cursor along the slider slows down execution as much as desired. The position of the slider can be moved dynamically during the animation, and such changes take effect immediately.

The `save env` button causes a record of the current screen configuration and the various option settings in ParaGraph to be written in a file, so that, if desired, the same selection of displays and options can be established immediately upon subsequent invocations of ParaGraph. By default, the initialization file is called `.pgrc`, but a different name can be specified using the `-e` command-line option. The screen locations of all displays are among the information saved in the initialization file, but this placement may or may not be honored by a given window manager. Regardless of user requests, some window managers insist on interactive placement of windows and others insist on choosing their own locations beyond the user's control.

The `open env/close all` button alternately opens whatever set of displays are specified in the current initialization file, or closes all currently open displays except the main menu. The intent is to allow for quick reconfiguration of displays, including reestablishment of the initial setup, without having to close or open many windows individually or restart ParaGraph.

The `screen dump` button enables any window (e.g., a single display or the entire screen) to be printed on a hardcopy output device, usually a laser printer. After pressing the `screen`

`dump` button, a particular window is selected for printing by clicking the mouse with the cross-hairs cursor in the desired location. The appropriate local command for routing the resulting screen dump to a suitable output device must appear in the `print command` subwindow of the `options` menu (see below).

The `reset` button clears all displays and returns to the beginning of the current tracefile, without restarting the animation. The `quit` button terminates ParaGraph.

5 Displays

In this section we describe the individual displays provided by ParaGraph. For color illustrations of many of the displays, see [12, 13]. Some of the displays change in place dynamically as events occur, with execution time in the original run represented by playback time in the replay. Others depict time evolution by representing execution time in the original run by one space dimension on the screen. The latter displays scroll as necessary (by a user-controllable amount) as playback time progresses, in effect providing a moving window for viewing what could be considered a static picture. No matter which representation of time is used, all displays of both types are updated simultaneously and synchronized with each other.

As stated earlier, most of the displays fall into one of three basic categories—utilization, communication, and task information—although some displays contain more than one type of information, and a few do not fit these categories at all. Below we provide brief descriptions of the displays. Most of the displays scale to reasonably large numbers of processors, but a few contain too much detail to scale up well. The current limit for most of the displays is 512 processors; the few exceptions are noted specifically below.

5.1 Utilization Displays

The displays described in this section are concerned primarily with processor utilization. They are helpful in determining the effectiveness with which the processors are used and how evenly the computational workload is distributed among the processors. At any given time, ParaGraph categorizes each processor as

- *idle* if it has suspended execution awaiting completion of a communication operation (or if it has ceased execution at the end of the run),
- *overhead* if it is executing in the communication subsystem but is not explicitly suspended, or
- *busy* if it is executing some portion of the program other than the communication subsystem.

Ideally, we would like to interpret *busy* as meaning that a processor is doing useful work, *overhead* as meaning that a processor is doing work that would be unnecessary in a serial program, and *idle* as meaning that a processor is doing nothing. Unfortunately, the monitoring required to make such a determination would almost certainly be nonportable and/or excessively intrusive. Thus, the "busy" time we report may well include redundant work or other work that would not be necessary in a serial program, since our monitoring detects only the overhead associated with communication. However, we find that the definitions

we have adopted based on the data provided by MPICL are quite adequate in practice to convey the effectiveness of message-passing parallel programs pictorially.

We note that our distinction between idle and overhead does *not* correspond directly to the distinction in MPI between blocking and nonblocking communication. In the MPI standard, the term "blocking" does not necessarily imply that the process suspends execution, but simply means that the calling process can re-use the resources specificed in the call immediately upon return. If a "blocking" MPI send is buffered, for example, then the sending process need not suspend execution. One case that warrants comment is the "generic" send, `MPI_Send`, which, according to the MPI standard, may or may not be buffered, depending on the specific implementation. Because most implementations of MPI employ a variety of protocols depending on message size, we have taken the generic send to generate overhead rather than idle, consistent with the previous convention in PICL.

Utilization Count. This display shows the total number of processors in each of three states—busy, overhead, and idle—as a function of time. The number of processors is on the vertical axis and time is on the horizontal axis, which scrolls as necessary as playback proceeds. The color scheme used is borrowed from traffic signals: green (go) for busy, yellow (caution) for overhead, and red (stop) for idle. By convention, along the vertical axis green is at the bottom, yellow in the middle, and red at the top. Since the three categories are exhaustive and mutually exclusive, the total height of the composite is always equal to the total number of processors.

Gantt Chart. This display, which is patterned after graphical charts used in industrial management [6], depicts the activity of individual processors by a horizontal bar chart in which the color of each bar indicates the busy/overhead/idle status of the corresponding processor as a function of time, again using the traffic-signal color scheme. Processor number is on the vertical axis and time is on the horizontal axis, which scrolls as necessary as playback proceeds. The Gantt chart provides the same basic information as the Utilization Count display, but on an individual processor, rather than aggregate, basis; in fact, the Utilization Count display is simply the Gantt chart with the green sunk to the bottom, the red floated to the top, and the yellow sandwiched between.

Kiviat Diagram. This display, which is adapted from related graphs used in other types of performance evaluation [22, 27], 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).

Taken together, the points for all the processors determine the vertices of a polygon whose size and shape give a pictorial indication of both processor utilization and load balance across processors. Low utilization causes the polygon to be concentrated near the center, while high utilization causes the polygon to lie near the perimeter. Poor load balance across processors causes the polygon to be strongly skewed or asymmetric. Any change in load balance is clearly shown pictorially; for example, with many ring-oriented algorithms the moving polygon has the appearance of a rotating camshaft as the heavier workload

moves around the ring. Other algorithms show a rhythmic oscillation of the polygon, much like a systolic "heartbeat."

The current utilization is shown in dark shading, while the "high water mark" seen thus far is shown in lighter shading. Since the Kiviat polygon may not be convex, and the high water mark for different processors may occur at different times, the outer figure may not have simple straight sides connecting the spokes. The "current" utilization used in this diagram is in fact a moving average over a time interval of user-specified width, since instantaneous utilization would of course always be either zero or one for each processor. The width of this smoothing interval can be changed via the `options` menu. A button is provided for the user to choose whether the utilization plotted includes only busy time, or both busy and overhead (i.e., not idle).

Streak. This display is based loosely on newspaper listings of team sports standings that often include data on winning and losing streaks. Processor numbers are on the horizontal axis, and the length of the current streak for each processor is on the vertical axis. Busy is always considered winning and idle is always considered losing. Overhead (perhaps analogous to ties in sports) can be lumped in with either winning or losing, as selected using the button provided (so that the streaks might be more accurately termed undefeated or winless, respectively). By convention, winning streaks rise from the horizontal axis and losing streaks fall below the horizontal axis. This distinction is further emphasized by color coding (green for winning and red for losing). As the current streak for each processor grows, the corresponding vertical bar rises (or falls) from the horizontal axis. When a streak for a processor ends, its bar returns to the horizontal axis to begin a new streak. At the end of playback, the longest streaks (both winning and losing) at any point during the run are shown for each processor. This display often provides insight into rhythmic patterns in parallel programs or imbalances across processors.

Utilization Summary. This display shows the cumulative percentage of time, over the entire run, that each processor has spent in each of the three busy/overhead/idle states. The percentage of time is shown on the vertical axis and the processor number on the horizontal axis. Again, the green/yellow/red color scheme is used to indicate the three states. In addition to giving a visual impression of the overall efficiency of the parallel program, this display also gives a visual indication of the load balance across processors.

Utilization Meter. This display uses a colored vertical bar, with the usual green/yellow/red color scheme, to indicate the percentage of the total number of processors that are currently in each of the three busy/overhead/idle states. The visual effect is similar to that of a thermometer or some automobile speedometers. This display provides essentially the same information as the Utilization Count display, but saves screen space (which may be needed for other displays) by changing in place rather than scrolling with time.

Concurrency Profile. For each possible number of processors k, $0 \leq k \leq p$, where p is the maximum number of processors for this run, this display shows the percentage of time during the run that *exactly* k processors were in a given state (i.e., busy/overhead/idle).

The percentage of time is shown on the vertical axis and the number of processors k is shown on the horizontal axis. The profile for each possible state is shown separately, and the user can cycle through the three states by clicking on the appropriate subwindow. This display is defined only at the end of playback. The actual concurrency profile for real programs shown by this display is usually in marked contrast to the idealized conditions that are the basis for Amdahl's Law, in which the concurrency profile is assumed to be bimodal, with nonzero values at $k = 1$ and $k = p$ and zero elsewhere (i.e., at any given time the computational work is either strictly serial or fully parallel).

5.2 Communication Displays

The displays described in this section are concerned primarily with depicting interprocessor communication. They are helpful in determining the frequency, volume, and overall pattern of communication, and whether there is congestion in the message queues or on the links of the interconnection network.

Communication Traffic. This display is a simple plot of the total traffic in the communication system (interconnection network and message buffers) as a function of time. The curve plotted is the total of all messages that are currently pending (i.e., sent but not yet received), and can be optionally expressed either by message count or by volume in bytes. The communication traffic shown can also optionally be either the aggregate over all processors or just the messages pending for any individual processor the user selects. Message volume or count is shown on the vertical axis, and time is shown on the horizontal axis, which scrolls as necessary.

Spacetime Diagram. This display is patterned after the diagrams used in physics, particularly in the theory of relativity, to depict interactions between particles through space and time. This type of diagram has been used by Lamport [23] for describing the order of events in a distributed computing system. The same pictorial concept was used over a century ago to prepare graphical railway schedules [37, page 31]. In our adaptation of 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. These sending and receiving times are from user process to user process (not simply the physical transmission time), and hence the slope of the resulting message line gives a visual indication of how soon a given piece of data was actually needed by the receiving processor after it was produced by the sending processor.

The message lines are color coded by size, tag, or distance, as chosen by the user with the Color Code display (see below). A given message line cannot be drawn until its receive time has been reached, so this display may appear to be "behind" other displays that depict messages as soon as the send event is encountered. The Spacetime Diagram is one of the most informative of all the displays, since it depicts both individual processor utilization

and all message traffic in full detail. For example, it can easily be seen which particular message "wakes up" an idle process that was previously suspended awaiting its arrival. Unfortunately, this fine level of detail does not scale up well to large numbers of processors, as the diagram becomes extremely cluttered, and its current limit is 128 processors.

Message Queues. This display depicts the size of the queue of messages for each processor by a vertical bar whose height varies with time as messages are sent, buffered (on either the sending or receiving processor), and received. The processor number is shown on the horizontal axis. The appropriate queue is incremented when a message is sent and decremented when a message is received. At the user's option, the queue size can be measured either by the number of messages or by their total length in bytes.

In most message-passing systems, the physical transmission time between processors is negligible compared to the software overhead in handling messages, so that the time interval between the send and receive events is a reasonable approximation to the time a given message actually spends in message queues at the source or destination. In some message-passing systems, messages are queued on the sending processor for eventual transmission to the receiver when requested, whereas in others messages are transmitted immediately to the destination processor, where they are queued for eventual reception by the user process. In this display, with the "outgoing" option the message queues are drawn as though messages are queued on the sending processor, and with the "incoming" option the message queues are drawn as though messages are queued on the receiving processor.

Depending on message tags, messages may not be received in the same order in which they arrive, so the queues may grow and shrink in complicated ways. As before, dark shading depicts the current queue size on each processor, and lighter shading indicates the "high water mark" seen so far. The Message Queue display gives a pictorial indication of whether there is communication congestion or the messages are consumed at about the same rate that they are created. It is best if messages are available slightly before they are actually needed, so that the receiving process does not become idle awaiting a message. A large backlog of undelivered messages can consume excessive buffer space, however, so a happy medium (analogous to "just in time" manufacturing) is desirable.

Communication Matrix. In this display, messages are represented by squares in a two-dimensional array whose rows and columns correspond to the sending and receiving processors, respectively, for each message. During playback, each message is depicted by coloring the appropriate square at the time the message is sent, and erasing it at the time the message is received. The color used is determined by the Color Code display (see below). Thus, the durations and overall pattern of messages are depicted by this display. The nodes can be ordered along the axes in either natural, Gray code, or user-defined order, and the choice may strongly affect the appearance of the communication pattern. At the end of playback, this 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.

Communication Meter. This display uses a vertical bar to indicate the percentage of maximum communication volume (or number of messages) currently pending (i.e., sent but

not yet received). This display provides essentially the same information as the Communication Traffic display, but saves screen space (which may be needed for other displays) by changing in place rather than scrolling with time. Conceptually, this thermometer-like display is similar to the Utilization Meter display, except that it shows communication instead of utilization, and the two are interesting to observe side by side.

Animation. In this display, the parallel system is represented by a graph whose nodes (depicted by numbered circles) represent processors, and whose arcs (depicted by lines between the circles) represent communication between processors. The status of each node (busy, overhead, idle, sending, receiving, or collective communication) is indicated by its color, so that the circles can be thought of as the "front-panel lights" of the parallel computer. A straight 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 playback 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. In particular, the possible routing of messages through intermediate nodes is not depicted unless the program being visualized does such forwarding explicitly.

The nodes can be arranged in ring, mesh, or user-defined configurations by clicking on the appropriate subwindow. For the mesh, the user can also select the desired aspect ratio and row-wise or column-wise numbering by clicking on the appropriate buttons. In addition, the nodes can be arranged in natural, Gray code, or user-defined order, and the user's choice may strongly affect the appearance of the communication pattern among processors. The various arrangements of the nodes are merely pictorial conveniences, and do not necessarily imply anything about the structure of the underlying interconnection network topology on which the parallel program was run.

If a user-defined layout is selected, then the processors can be placed arbitrarily within the display by using the mouse. Initially, the nodes are arranged in a default layout. A given node can be moved anywhere within the window by first clicking on the chosen node to select it, and then clicking again at the desired new location. If desired, a layout determined in this way can be saved in a file for future use. The default name for such an animation layout file is `.pganim`, but a different name can be specified using the `-l` command-line option or by typing the name into the appropriate subwindow when using the `read file` or `write file` options of the display.

Note that various combinations of states are possible for the sending and receiving processors on either end of a message line. For example, both processors could be busy, one having already sent the message and resumed computing, while the other has not yet stopped computing to receive the message. Upon conclusion, this display shows a summary of all (logical) communication links used throughout the run. Because of its level of detail, this display is currently limited to depicting up to 128 processors.

Hypercube. This display is similar to the Animation display, except that it provides a number of additional layouts for the nodes in order to exhibit more clearly communication patterns corresponding to the various networks that can be embedded in a hypercube. Note that this display does not require that the interconnection network of the machine on which

the parallel program executed actually be a hypercube; it merely highlights the hypercube structure as a matter of potential interest. The scheme for coloring nodes and drawing arcs is the same as that for the Animation display, except that curved arcs are often used to avoid, as much as possible, intersecting other nodes. To help determine if the communication in the parallel program honors the hypercube's physical connectivity, message arcs corresponding to physical hypercube links are drawn in a different color from those along "virtual" links that do not exist in a hypercube and would require routing through intermediate processors. If the actual number of processors is not a power of two, then any "unused" nodes in the selected layout are indicated by black shading. Upon conclusion, this display shows a summary of all (logical) communication links used throughout the run. Unfortunately, the method used to draw this rather elaborate display does not scale up well to large numbers of processors, so it is currently limited to 16.

Network. This display depicts interprocessor communication in terms of various network topologies. Unlike the Animation and Hypercube displays, the Network display shows the actual path that each message takes in a user-selected network, which may include routing through one or more intermediate nodes between the original source and ultimate destination. Obviously, depicting message routing through a network requires a knowledge of the interconnection topology, but there is no such information in the trace data collected by MPICL. Instead, the user selects from among several of the most common interconnection networks, each of which may also have a choice of routing schemes. Some of the available topologies are represented as multistage networks, with duplicate sets of source and destination nodes, between which are several "stages" of nodes or switches through which intermediate routing occurs. Networks depicted in this manner include butterfly, hypercube, omega, baseline, and crossbar. Other available topologies are represented by a single set of nodes that serve as both sources and destinations, with messages moving in either direction through the network. Networks depicted in this manner include binary tree, quadtree, and mesh.

Each physical link in the network is color coded according to the number of messages currently sharing that link. A temperature-like color code is used, so that "hot spots" appear red while less heavily used links appear blue. In monochrome, the message count on a link is indicated instead by the line width, so that, for example, the tree networks look like "fat" trees, as the message count tends to be higher nearer the root. Unlike the Animation or Hypercube displays, in the Network display the sending or receiving of a message does not necessarily cause the drawing or erasure of a given link, but will often merely change its color to be one step hotter or cooler than it was previously. A given message may use several links, causing each link involved to be incremented or decremented separately. On conclusion, the coloring of the links indicates the cumulative message count over the entire run, and the color-code legend is recalibrated accordingly to indicate the range of cumulative totals for the various links.

The choices of network and routing scheme (and also the aspect ratio and row-wise or column-wise ordering for the mesh) are selected by clicking on the appropriate subwindow. The choice of network topology and routing scheme need not match those of the machine on which the parallel program actually ran, but the representation is obviously most accurate if they do match. On the other hand, one might want to choose a different network deliberately

in order to get some idea how a program that ran on one topology might perform on a different topology. Thus, for example, the user of an mesh can see a visual playback of the behavior his program might have on a quadtree, or vice versa.

The node numbers of the peripheral nodes are always shown, but by default the interior node numbers are omitted in the multistage and mesh networks to avoid excessive clutter amid the message lines. All of the node numbers can be shown, however, by clicking on the appropriate option subwindow. Due to its high degree of detail, this display is currently limited to 128 processors.

Node Data. This display provides, in graphical form, detailed communication data for any single processor the user selects. The choices of data plotted are the source/destination, tag, length, and distance traveled for all messages sent to or from the chosen processor. The length of a message is in bytes, and the distance traveled is in hops from source to destination as determined by the distance function chosen using the `options` menu. Time is on the horizontal axis, and the chosen statistic is on the vertical axis, with incoming and outgoing messages shown in separate subwindows. This display is helpful in analyzing communication behavior in detail, especially in perceiving trends or patterns in the communication structure that improve understanding of program behavior and performance. It has been used as an aid in designing "synthetic programs," which are simple programs that mimic the behavior and performance of much more complex programs, and are useful for performance modeling and benchmarking. This display is currently limited to depicting 256 processors.

Color Code. This display permits the user to select which statistic determines the code for coloring the messages in displays such as Spacetime and Communication Matrix. The choices include the size of the message in bytes, the distance between the source and destination nodes (according to the distance function chosen using the `options` menu), and the message tag. Clicking on the subwindow cycles through the choices, and the resulting color code is displayed to enable proper interpretation of the other displays that use it.

5.3 Task Displays

The displays we have considered thus far depict a number of important aspects of parallel program behavior that help in detecting performance bottlenecks. However, they contain no information indicating the location in the parallel program at which the observed behavior occurs or the rate at which useful work is being accomplished. To remedy this situation, a number of "task" displays are provided that use information supplied by the user, with the help of MPICL, to depict the portion of the user's parallel program that is executing at any given time, and optionally the amount and rate of useful work being done.

Specifically, the user defines "tasks" within the program by calling an MPICL routine to mark the beginning and ending of each task and assign it a user-selected, nonnegative task number. The scope of what is meant by a task is left entirely to the user: a task can be a single line of code, the body of a loop, an entire subroutine, or any other unit that is meaningful in a given application. For example, in matrix factorization one might define the computation of each column to be a task, and assign the column number as the task number. Tasks are delineated by inserting calls to MPICL's `traceevent` routine,

with the desired task number as the `eventtype` argument, immediately before and after the selected section of code. This causes MPICL to produce event records that are used by ParaGraph to enable it to depict the given task, using displays described below. Task definitions are required *only* if the user wishes to view the task displays. If the tracefile contains no event records defining tasks, then the task displays will simply be blank, but the remaining displays in ParaGraph will still show their normal information.

If desired, an optional additional argument to the `traceevent` at the end of each task can be used to indicate the amount of work performed by the given node during that task, expressed in user-chosen units appropriate for the problem (flops, zones, particles, visits, traversals, transactions, etc.). The latter information, if present, is used in some of ParaGraph's task displays to depict the rate and cumulative volume of work accomplished. This interpretation of the additional `traceevent` parameter as representing task work is merely a convention used in ParaGraph and is not inherent in MPICL or the MPICL trace format.

Tasks can be nested, one inside another, but if so these should be properly bracketed by matching task entry and exit records. More than one processor can be assigned the same task, or, more accurately, each processor can be assigned its own portion of the same task, as happens when multiple processors contribute toward the computation of a given column in the matrix factorization example cited earlier. Indeed, the model we have in mind is that multiple processors collaborate on each task, rather than that each task is assigned uniquely to a single processor (although the latter interpretation is supported as well). The numbering of tasks can be chosen arbitrarily, but in many contexts, such as the matrix factorization example, there is a natural ordering and corresponding numbering of the tasks that will make interpretation of the resulting graphical display much easier.

In most of the task displays described below, the task numbers are indicated by a color coding. Since the number of tasks may be larger than the number of colors that can be easily distinguished, we recycle a limited number of colors to depict successive task numbers. We use a maximum of 64 different colors for indicating individual tasks. To aid in distinguishing consecutively numbered tasks (the most common case) we stride through these 64 colors in groups of eight rather than in strict rainbow sequence. If desired, the user can override these default task colors by supplying a file containing up to 64 sets of RGB values. The default name for such a file is `.pgcolors`, but an alternative filename can be specified on the command line by using the `-r` option. Each line of the color file contains four integers, the first of which is the color number (0–63) that is being replaced, and the other three are the R, G, and B values (0–255) for the substitute color. In monochrome mode, stipple patterns are used to distinguish tasks. The eight different stipple patterns available are recycled as needed for larger numbers of tasks.

Task Count. During playback, this display shows the number of processors that are executing a given task at the current time. The number of processors is shown on the vertical axis and the task number is shown on the horizontal axis. At the end of playback, this display changes to show a summary over the entire run. Specifically, it shows the average number of processors that were executing each task over the lifetime of that task (i.e., the time interval starting when the first processor began the task and ending when the last processor finished the task).

Task Gantt. This display depicts the task activity of individual processors by a horizontal bar chart in which the color of each bar indicates the current task being executed by the corresponding processor as a function of time. Processor number is on the vertical axis and time is on the horizontal axis, which scrolls as necessary as playback proceeds. This display can be compared with the Utilization Gantt chart to correlate busy/overhead/idle status with the task information.

Task Speed. This display is similar to the Task Gantt display, but the color coding of the horizontal bars indicates the speed with which work is done during each task, where "speed" is defined as the work done during a task divided by the elapsed time of the task. Of course, the necessary work information must be contained in the task exit event records in the tracefile. This display can be viewed alongside the Task Gantt display to identify the tasks involved.

Task Status. In this display the tasks are represented by a two-dimensional array of squares, with task numbers filling the array in row-wise order. Initially, all of the squares are white. As each task is begun, its corresponding square is lightly shaded to indicate that the task is now in progress. When a task is subsequently completed, its corresponding square is then darkly shaded.

Task Summary. This display, which is defined only at the end of playback, indicates the duration of each task (from earliest beginning to last completion by any processor) as a percentage of the overall execution time of the parallel program, and furthermore places the duration interval of each task within the overall execution interval of the parallel program. The percentage of the total execution time is shown on the vertical axis, and the task number is shown on the horizontal axis.

Task Surface. This display attempts to show the relationships among tasks, processors, and time in a single diagram. Processor and task numbers are shown on the two axes. One can think of these two quantities as forming a plane, over which a surface can be plotted (coming out of the screen). Rather than an actual three-dimensional representation, the resulting surface is color-coded to show height above the processor-task plane. At the user's option, three different quantities can be chosen for the surface value: elapsed time, work, or speed. The latter two require that the necessary work information be contained in the task exit event records in the tracefile.

Task Work. This display uses the optional work information contained in task exit events, if present, to depict both the rate at which work is completed and the cumulative volume of work completed. The rate and cumulative volume of work can be displayed for any user-selected node or can be aggregated over all nodes.

5.4 Other Displays

In this section we describe some additional displays that either do not fit into any of the three main categories or else cut across more than one category.

Clock. This display provides both digital and analog clock readings during graphical playback of the parallel program. The current playback time is shown as a numerical reading, and the proportion of the full tracefile that has been completed thus far is shown by a colored horizontal bar. The clock reading is updated synchronously with the other displays, and it "ticks" through all integral time values, not just those that happen to correspond to event timestamps.

Trace. This is a non-graphical display that prints an annotated version of each trace event as it is read from the tracefile. 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. Although the trace events are printed in this display one at a time, space is allowed to show several consecutive trace events in context, and the display scrolls vertically as necessary with time. By default, all trace events are printed, but events can be printed selectively, by node or type, by changing the appropriate setting in the `options` menu.

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 percentage of busy, overhead, and idle time; total count and volume (in bytes) of messages sent and received; maximum queue size; and maxima, minima, and averages for the size, distance traveled (according to the distance function chosen using the `options` menu), and transit time for both incoming and outgoing messages. The numerical quantities provided in this tabular display can be useful in preparing tables and graphs for printed reports, or for performance modeling. Due to limited space on the screen, this display shows data for at most 16 processors at a time; the subset of processors shown is chosen by clicking along the slider subwindow provided, which enables browsing of the entire data array.

Processor Status. This is a comprehensive display that attempts to capture detailed information about processor utilization, communication, and tasks, but in a compact format that scales up well to large numbers of processors. This display contains four subdisplays, in each of which the processors are represented by a two-dimensional array of squares, with processor numbers filling the array in row-wise order. The upper left subdisplay shows the current state of each processor (busy/overhead/idle), using the usual green/yellow/red color scheme. The upper right subdisplay shows the task currently being executed by each processor, using the same task coloring scheme as discussed previously. The lower left subdisplay shows the volume of messages currently being sent by each processor, and the lower right subdisplay shows the volume of messages currently awaiting receipt by each processor; both of these communication subdisplays indicate message volume in bytes using the same color code as discussed previously for the other communication displays. Although this comprehensive display is somewhat difficult to follow due to the large amount of information it contains, it has the virtue of readily scaling to very large numbers of processors.

Critical Path. This display is similar to the Spacetime display described earlier, but uses a different color coding to highlight the longest serial thread in the parallel compu-

tation. Specifically, the processor and message lines along the critical path are shown in red, while all other processor and message lines are shown in light blue. This display is intended to aid in identifying performance bottlenecks and tuning the parallel program by focusing attention on the portion of the computation that is currently limiting performance. Any improvement in performance must necessarily shorten the longest serial thread running through the computation, so this is a primary place to look for potential algorithm improvements. For larger numbers of processors, the noncritical message lines are suppressed so that they do not obscure the critical path.

Phase Portrait. This display is patterned after the phase portraits used in differential equations and classical mechanics to depict the relationship between two variables (e.g., position and velocity) that depend on some independent variable (e.g., time). In our case, we are attempting to illustrate pictorially the relationship over time between communication and processor utilization. At any given point in time, the current percentage utilization (i.e., the percentage of processors that are in the busy state), and the percentage of the maximum volume of communication currently in transit, together define a single point in a two-dimensional plane. This point changes with time as communication and processor utilization vary, thereby tracing out a trajectory in the plane that is plotted graphically in this display, with communication and utilization on the two axes. To filter out noise in plotting the trajectory, this display uses the same smoothing interval as the Kiviat diagram, and thus the amount of smoothing can be controlled via the `options` menu.

Since the overhead and potential idleness due to communication inhibit processor utilization, one expects communication and utilization generally to have an inverse relationship. Thus one expects the phase trajectory to tend to lie along a diagonal of the display. This display is particularly useful for revealing repetitive or periodic behavior in a parallel program, which tends to show up in the phase portrait as an orbit pattern. The color used for drawing the trajectory is determined by the current task number on processor 0 (default is black if no such task is active), so by setting task numbers appropriately, the user can color code the trajectory to highlight either major phases or individual orbits.

Coordinate Information. This is a non-graphical display used to print information produced by mouse clicks on other displays. Many of the displays respond to mouse clicks by printing in the Coord Info display the coordinates (in units meaningful to the user) of the point at which the cursor is located at the time the button is pressed. This feature is intended to enable the user to determine precisely information that may be difficult to read accurately from the axis scales alone. In addition, clicking on a node in the Animation display causes the following information to be printed in the Coord Info display: time, node number, task number (if any), number of incoming messages pending, and number of outgoing messages pending. The latter information can be used in conjunction with the color code in the Animation display to determine the exact state of the nodes more precisely.

5.5 Application-Specific Displays

All of the displays we have discussed thus far are generic in the sense that they are applicable to any parallel program based on message passing and do not depend on the particular

application or problem domain that the program addresses. While this wide applicability is generally a virtue, knowledge of the specific application can often enable one to design a special-purpose display that reveals greater detail or insight than generic displays alone would permit. In studying a parallel sorting algorithm, for example, generic displays can show which processors are communicating with each other, and the volume of communication, but they cannot show which specific data items are being exchanged between processors. Since such application-specific displays obviously could not be provided within a general-purpose tool, ParaGraph was instead made extensible so that users can add application-specific displays of their own design that can be selected from a menu and viewed along with the usual generic displays.

The mechanism for supporting this capability works as follows. ParaGraph contains calls at appropriate points to routines that provide initialization, data input, event handling, drawing, etc., for application-specific displays. If the corresponding routines for such displays are not supplied by the user when the executable module for ParaGraph is built, then dummy "stub" routines are linked into ParaGraph instead, and the `user` submenu selection does not appear on the main menu in the list of available submenus. If application-specific displays have been linked into ParaGraph and the resulting module is executed, then a `user` item appears in the main menu, and its selection opens a `user` submenu that is analogous to the other submenus of available displays. The `user` submenu may contain any number of separate user-defined displays that can be selected individually. Each such user-supplied display is given access to all of the event records in the tracefile that ParaGraph reads, as well as all X events, and can use them in any manner it chooses. In particular, the user-supplied displays can receive input interactively via the mouse or keyboard.

The usual events generated by MPICL may suffice for the application-specific displays, or the user may wish to insert additional events during execution of the parallel program in order to supply additional data for the application-specific display. The `tracedata` function of MPICL is perhaps the most useful for this purpose, as it allows the user to insert into the tracefile timestamped records containing arbitrary lists of data, which might be used to provide loop counters, array indices, memory addresses, identifiers of objects such as particles or transactions, or any other information that would enable the user-supplied display to convey more fully and precisely the activity of the parallel program in the context of the particular application.

Unfortunately, writing the necessary routines to support application-specific displays is a decidedly nontrivial task that requires a general knowledge of X Window programming. But at least the potential user of this capability can concentrate on only those portions of the graphics programming that are relevant to the application, taking advantage of the supporting infrastructure of ParaGraph to provide all of the other necessary facilities to drive the overall graphical playback. As an aid to users who may wish to develop application-specific displays to add to ParaGraph, the source code for several prototype application-specific displays is provided along with the source code for ParaGraph. These example applications include parallel sorting algorithms, recursive matrix transposition, general matrix computations, and graph algorithms, as well as displays for operation counts (e.g., flops, particles, transactions).

6 Options

The execution behavior and visual appearance of ParaGraph can be customized in a number of ways to suit each user's taste or needs. The individual items in the `options` menu are described in this section. Some of the menu items cycle through the available choices as the mouse is clicked on the corresponding subwindow, while others accept keyboard input to specify numerical values or character strings. The type of user input expected for a given menu entry is indicated by the type of mouse cursor it displays. For the menu items that take keyboard input, existing values can be erased by the `backspace` or `delete` keys. When typing a new value, the characters are echoed in reverse video. Pressing the `return` key completes the keyboard input and makes the new value take effect, at which point the new characters revert to normal video display. In this section, we briefly discuss some of the choices available in the `options` menu.

Order. In many of the displays, the user can choose to have the processors arranged in natural, Gray code, or user-defined order, and the choice will affect the appearance of communication patterns. The Gray code order is not permitted if the number of processors is not a power of two. If desired, a user-defined ordering can be supplied by means of an order file. The default name for an order file is `.pgorder`, but an alternate filename can be given by using the `-o` command-line option. An order file contains two numbers per line, the first of which is a node number and the second of which is the desired position of that node in the user-defined order. Optionally, a third field can be specified on each line which, if present, is interpreted as a character string to be used as the name for that node. The ability to rename the nodes is intended to support heterogeneity in either the application program (e.g., master-slave) or the underlying architecture (e.g., a network of various workstations), in which case it may be desirable to be able to distinguish nodes of different types. Due to limited space available in the displays where they will be used, node names are limited to three characters. If no order file is supplied by the user, then the `user` item does not appear among the choices for the ordering.

Scrolling. Those windows that represent time along the horizontal dimension of the screen can smoothly scroll or jump scroll by a user-specified amount as playback time advances. Smooth scrolling provides an appealing sense of visual continuity, but results in a slower drawing speed.

Time Unit. The relationship between playback time and the timestamps of the trace events is determined by the `time unit` chosen. MPICL provides event timestamps in seconds with a nominal resolution of microseconds. A value of 100 for the time unit in ParaGraph, for example, means that each "tick" of the playback clock corresponds to 100 microseconds in the original execution of the parallel program. During preprocessing, ParaGraph scans the timestamps in the the tracefile and attempts to determine a reasonable value for the time unit. The user can override this automatic choice, however, simply by entering a different choice in the `time unit` subwindow. Once the time unit is set, all displays (as well as user input) are expressed in terms of this time unit rather than the units of the original raw timestamps in the tracefile.

Magnification. This parameter determines the visual resolution of the horizontal axis in the displays that scroll with time. It specifies the number of pixels on the screen that represent each unit of playback time. A larger number of pixels per time unit magnifies the horizontal dimension of the scrolling displays to bring out more detail, but with less of the overall behavior of the program visible at once. The choices available for the magnification factor are 1, 2, 4, and 8. The user can override the default value chosen automatically by ParaGraph. The visual effect of this parameter is much like that of using a magnifying glass of the given power. The magnification factor and the time unit, as we have defined them, are related to each other in the effect they have on the appearance of the displays that scroll with time, but they serve distinct purposes: the choice of `magnification` determines the visual resolution of the drawing on the screen, while the choice of `time unit` determines the time resolution of trace events. Thus, these two quantities can be varied in concert to produce any desired effect.

Start Time and Stop Time. By default, ParaGraph starts the playback at the beginning of the tracefile and proceeds to the end of the tracefile. By choosing other starting and stopping times, however, the user can isolate any particular time period of interest for visual scrutiny without having to view a possibly long playback in its entirety. Once the specified stopping time is reached, playback pauses, and then can be resumed by typing a new (still later) stopping time, or by clicking on the `pause/resume` menu button, or by clicking on the `step` menu button to proceed from this point in single-step mode.

Step Increment. This parameter determines how many consecutive records from the tracefile are processed each time the `step` button is pressed on the `controls` menu. The default value of one provides the finest control for detailed scrutiny, but can be tedious and time consuming to use in some cases, so the user may prefer a larger value.

Smoothing Interval. The user can select the amount of smoothing used in the Kiviat Diagram and Phase Portrait displays to avoid an excessively noisy or jumpy appearance. The amount of smoothing is determined by the width of a moving time interval, with a larger value giving more smoothing and a smaller value giving less smoothing. This parameter is expressed in playback time units and it can be changed simply by typing a new value.

Pause on Tracemark/msg. Another way to stop playback for detailed study at a given predetermined point is to insert `tracemark` or `tracemsg` event records into the tracefile during the original execution of the parallel program. These special records are generated by calling MPICL's `traceevent` routine and can be used to mark milestones in the user's program, such as the completion of a major phase of the program or the beginning of a new one, or a point at which a bug is suspected. This provides a program-dependent means of isolating particular points of interest for close scrutiny. After the playback has stopped at a `tracemark` or `tracemsg` event, it can be resumed by any of the usual actions, including single stepping.

Pause on Error. This option determines whether playback is paused if an error is detected, such as a mismatched send/receive pair or incorrectly nested blocks. Again, playback can be resumed by any of the usual actions.

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`.

Print Command. This specifies the command string used by the `screen dump` button on the `controls` menu to route images to a printer for hardcopy output. The default print command is installation dependent. It can be changed simply by typing a new print command in this subwindow. The command string will often include invocation of a remote shell and piping through a number of filters for converting image formats, etc., before reaching the physical output device.

Distance Function. This option determines the network topology used to compute the distance traveled by each message, which may be used in some displays for color coding messages and is also tallied in summaries of communication statistics. The choices available include Hamming distance (appropriate for hypercubes, for example), 1-D and 2-D mesh (without wrap-around), 1-D and 2-D torus (with wrap-around), binary tree, quadtree, and unit distance (appropriate for a fully connected network, for example). For the mesh and torus, the user also selects the desired aspect ratio and whether the processors are numbered row-wise or column-wise. The choice of distance function does not necessarily have to agree with the layout or topology chosen for the Animation and Network displays.

7 Recording Options

The data generated by ParaGraph for drawing the various displays can be saved in files, if desired. Such data may be useful for inclusion in printed documents, for mathematical modeling or statistical analysis of performance, or as input to other graphical packages. The `record options` menu is used to select which displays, if any, are to have their data recorded in files on disk during playback. Each data file created in this manner will have the name shown in the prefix subwindow, with a suffix added to indicate the particular display name. The default filename prefix is the pathname of the tracefile, if one was specified on the command line when ParaGraph was invoked. The filename prefix can be changed by entering a new name into the prefix subwindow. Another subwindow allows the user to specify start and stop times for saving data in files, which by default include the entire run. Each data file produced in this manner begins with a header that identifies the subsequent fields, followed by one line of data per event.

8 General Advice

In this section we provide a few tips that may make using ParaGraph easier and more meaningful. Perhaps the most important piece of advice is to keep the tracefile to be viewed as small as possible without losing the phenomenon to be studied. The best way to accomplish this is to use a relatively small number of processors and a brief execution time. Although ParaGraph currently supports the use of up to 512 processors, and has no limit on the duration of playback, the size of the tracefile for a large number of processors and/or a long execution time can be enormous (many megabytes). Such large tracefiles consume large amounts of disk space and will require a great deal of time for ParaGraph to preprocess and then animate visually. Fortunately, basic algorithm behavior and many fundamental bottlenecks and inefficiencies in parallel programs are usually already apparent when viewed with small numbers of processors and relatively small test problems that run quickly. Moreover, many programs display repetitive behavior, so that only a few iterations need be examined in detail in order to get the gist of their behavior, rather than a long sequence of replicated behavior. In a lengthy program, it is also a good idea to invoke MPICL's tracing commands only for the portion of immediate interest. One should also refrain from tracing in the host process, if such a host exists, since ParaGraph ignores all events involving the host anyway.

On some machines, a more insidious problem with large numbers of processors and/or long run times is the increasing likelihood of "tachyons" (messages that appear to be received before they are sent) in the tracefile as the number of processors increases or as individual processor clocks drift apart with time. On machines with independent node clocks, MPICL tries to compensate for clock skew and drift based on clock synchronizations before and after the run. The user can force additional synchronization points, and corresponding adjustments for clock skew and drift, by explicitly calling MPICL's `clocksync0` function. Depending on the particular parallel system, MPICL may rely solely on the `MPI_Wtime` function of MPI, or it may use lower-level system clocks directly for higher resolution in some cases. ParaGraph reports any tachyons it detects, but if they do occur, tachyons may cause unpredictable behavior in ParaGraph, possibly including outright failure. Other common causes of faulty tracefiles include failure to sort into time order, inadvertent concatenation of multiple tracefiles, and incomplete tracefiles due to full trace buffers.

The various parameters given in the `options` menu can have a dramatic effect on the behavior of ParaGraph for a given tracefile, and the user may or may not find the default values to be the most desirable. For example, during preprocessing a rough heuristic is used to choose an appropriate `time unit`, and the value chosen strongly affects the appearance and behavior of the scrolling displays. An attempt is made to choose a value that will fill at least one window width but not need to scroll more than a few window widths. The value chosen automatically may be so large that it obscures detail the user would like to see, or so small that playback runs for too long. The user should feel free to adjust the value for the `time unit`, if desired. Note, however, that the `magnification` parameter also affects the visual resolution of the scrolling displays, so it may also be changed to produce a desired effect. In addition, the speed of the drawing is strongly affected by the type and amount of scrolling employed, so this is subject to experimentation as well. In using the Kiviat Diagram and Phase Portrait displays, some experimentation with the `smoothing`

`interval`, as well as the `time unit`, may be required to produce the most meaningful visual results.

As pointed out previously, the execution speed of ParaGraph is normally determined by how fast it can read trace records and perform the resulting drawing. If the visual playback is too rapid for the eye to follow, then its execution can be slowed down either by using the `slow motion` slider or else by selecting some additional displays, especially those that scroll with time. If the visual playback is too slow, it can be speeded up by using fewer displays at a time or by selecting jump scrolling. Changing the `time unit` and/or `magnification` also affects the drawing speed, so these are subject to experimentation as well. Finally, the `step` button or repeatedly clicking `pause/resume` can also be used to control the speed with which the animation unfolds. By some combination of these means, the user should be able to produce an animation speed that can be followed visually in sufficient detail, yet does not take an inordinate amount time to finish.

In building an executable module for ParaGraph from the distributed source code, there are a few compile-time parameters that the user may wish to modify for a particular situation. The following parameters are found at the beginning of the `defines.h` file:

`ALL:` Integer destination value used to indicate global sends (default -1). This parameter is not relevant for MPI, but is retained for backward compatibility with PICL.

`HOST:` Integer identifier for host processor (default 32767). This parameter is unlikely to be used in MPI, but is retained for backward compatibility with PICL.

`MAXLINE:` Maximum number of characters per trace record (default 1000). The default value is more than adequate for any standard trace record produced by MPICL, but might need to be increased if long user-defined records are present in the tracefile.

`MAXP:` Maximum number of node processors allowed (default 512). If sluggish behavior is encountered due to limited memory, this value can be reduced, say to 128.

By default, ParaGraph uses two fonts, namely `6x12` and `8x13`, which are both available in most distributions of X Windows. In case these fonts are not available, however, the names of alternate constant-width fonts of similar size can be substituted into the initialization of `font1_name` and `font2_name` in the `defaults.h` file.

9 Future Work

In terms of the number and appearance of displays it provides, ParaGraph is a reasonably mature software tool, although we intend to add more displays as helpful new perspectives are devised. There are a few technical points about ParaGraph that could stand improvement. The contents of many of the displays are lost if the window is obscured and then reexposed. This inability to repair or redraw windows, short of rerunning playback from the beginning, was a deliberate design decision based on a desire to conserve the memory that would be required to save the contents of all windows for possible restoration. Nevertheless, this "feature" can be annoying at times and should eventually be fixed. A related problem is that ParaGraph cannot reliably support dynamic changes in parameters during playback (e.g., dynamic zooming of the time resolution).

A more serious limitation of ParaGraph in its current form is the number of processors that can be depicted effectively. A few of the current displays are simply too detailed to scale

up beyond about 128 processors and still be comprehensible. Most of the displays scale up well to a level of 512 processors on a normal sized workstation screen, but at this point they are down to representing each processor by a single pixel (or pixel line), and hence cannot be scaled any further in their current form. To visualize programs for massively parallel architectures having thousands of processors, we must devise new displays that scale up to this level, or else we must adapt the existing displays, either by aggregating or selecting information. For example, the current displays could depict either clusters of processors or subsets of individual processors (e.g., cross sections).

While it is fairly easy to imagine how graphics technology might be adapted to meet the needs of visualizing massively parallel computations, it is much less obvious how to handle the vast volume of execution trace data that would result from monitoring thousands of processors. Even with the more modest numbers of processors currently supported by ParaGraph, storage and processing of the large volume of trace data resulting from runs of significant duration are already difficult problems. To go beyond the present level will almost certainly require some degree of abstraction of essential behavior in a more concise and compact form, both in the data and in its graphical presentation. We simply cannot afford to continue to record or display all communication events when they become so voluminous. Unfortunately, many of the current displays in ParaGraph depend critically on the availability of data on each individual event. Thus, the development of new visual displays and new data abstractions must proceed in tandem so that the execution monitoring facility will produce data that can be visually displayed in a meaningful way to provide helpful insights into program behavior and performance.

References

[1] M. H. Brown. *Algorithm Animation*. MIT Press, Cambridge, MA, 1988.

[2] T. L. Casavant, guest editor. Special Issue on Parallel Performance Visualization, *Journal of Parallel and Distributed Computing*, 18(2), June 1993.

[3] A. L. Couch. Graphical representations of program performance on hypercube message-passing multiprocessors. Technical Report 88-4, Dept. of Computer Science, Tufts University, Medford, MA, April 1988.

[4] J. J. Dongarra and E. Grosse. Distribution of mathematical software via electronic mail. *Communications of the ACM*, 30(5), May 1987, pages 403–407.

[5] T. H. Dunigan. Hypercube clock synchronization. *Concurrency: Practice and Experience*, 4(3), May 1992, pages 257–268.

[6] H. L. Gantt. Organizing for work. *Industrial Management*, 58, August 1919, pages 89–93.

[7] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. PICL: a portable instrumented communication library, C reference manual. Technical Report ORNL/TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, July 1990.

[8] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A users' guide to PICL, a portable instrumented communication library. Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, Oak Ridge, TN, October 1990.

[9] W. Gropp, E. Lusk and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, 2nd edition. MIT Press, Cambridge, MA, 1999.

[10] G. Haring and G. Kotsis, editors. *Performance Measurement and Visualization of Parallel Systems*. Elsevier Science Publishers, Amsterdam, The Netherlands, 1993.

[11] M. T. Heath. Visual animation of parallel algorithms for matrix computations. In D. Walker and Q. Stout, editors, *Proceedings of the Fifth Distributed Memory Computing Conference*, volume II, pages 1213–1222, IEEE Computer Society Press, Los Alamitos, CA, April 1990.

[12] M. T. Heath and J. A. Etheridge. Visualizing performance of parallel programs. Technical Report ORNL/TM-11813, Oak Ridge National Laboratory, Oak Ridge, TN, May 1991.

[13] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5), September 1991, pages 29–39.

[14] M. T. Heath. Recent developments and case studies in performance visualization using ParaGraph. In G. Haring and G. Kotsis, editors, *Performance Measurement and Visualization of Parallel Systems*, pages 175–200, Elsevier Science Publishers, Amsterdam, The Netherlands, 1993.

[15] M. T. Heath. Performance visualization with ParaGraph. In *Proc. Second Workshop on Environments and Tools for Parallel Sci. Comput.*, pages 221–230, SIAM, Philadelphia, 1994.

[16] M. T. Heath, A. D. Malony, and D. T. Rover. The visual display of parallel performance data. *IEEE Computer*, 28(11), November 1995, pages 21–28.

[17] M. T. Heath, A. D. Malony, and D. T. Rover. Parallel performance visualization: From practice to theory. *IEEE Parallel Distrib. Tech.*, 3(4), Winter 1995, pages 44–60.

[18] M. T. Heath. Visualization of parallel and distributed systems. In A. Y. Zomaya, editor, *Parallel and Distributed Computing Handbook*, pages 897–916, McGraw-Hill, New York, 1996.

[19] M. T. Heath, A. D. Malony, and D. T. Rover. Visualization for parallel performance evaluation and optimization. In J. Stasko, J. Domingue, M. H. Brown and B. A. Price, editors, *Software Visualization*, pages 347–365, MIT Press, Cambridge, MA, 1998.

[20] V. Herrarte and E. Lusk. Studying parallel program behavior with Upshot. Technical Report ANL-91/15, Argonne National Laboratory, Argonne, IL, August 1991.

[21] D. N. Kimelman and T. A. Ngo. The RP3 program visualization environment. *IBM Journal of Research and Development*, 35(5/6), September/November 1991, pages 635–651.

[22] K. Kolence and P. Kiviat. Software unit profiles and Kiviat figures, *Performance Evaluation Review*, 2(3), September 1973, pages 2–12.

[23] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21, July 1978, pages 558–565.

[24] T. J. LeBlanc, J. M. Mellor-Crummey and R. J. Fowler. Analyzing parallel program executions using multiple views. *Journal of Parallel and Distributed Computing*, 9, 1990, pages 203–217.

[25] T. Lehr, Z. Segall, D. F. Vrsalovic, E. Caplan, A. L. Chung and C. E. Fineman. Visualizing performance debugging. *IEEE Computer*, 22(10), October 1989, pages 38–51.

[26] A. D. Malony and D. A. Reed. Visualizing parallel computer system performance. Technical Report UIUCDCS-R-88-1465, Dept. of Computer Science, University of Illinois, Urbana, IL, September 1988.

[27] M. F. Morris. Kiviat graphs — conventions and figures of merit. *Performance Evaluation Review*, 3(3), October 1974, pages 2–8.

[28] MPI Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4), Fall/Winter 1994, pages 157–416.

[29] MPI Forum. MPI2: A message-passing interface standard. *International Journal of High Performance Computing Applications*, 12(1/2), Spring/Summer 1998, pages 1–299.

[30] P. S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann, San Francisco, 1997.

[31] D. T. Rover, Visualization of program performance on concurrent computers. Ph.D. Thesis, Iowa State University, Ames, IA, 1989.

[32] M. Simmons, R. Koskela and I. Bucher, editors. *Parallel Computer Systems: Performance Instrumentation and Visualization.* ACM Press, New York, 1990.

[33] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker and J. Dongarra. *MPI: The Complete Reference*, 2nd edition. MIT Press, Cambridge, MA, 1998.

[34] D. Socha, M. L. Bailey and D. Notkin. Voyeur: graphical views of parallel programs. *SIGPLAN Notices*, 24(1), January 1989, pages 206–215.

[35] J. Stasko, J. Domingue, M. H. Brown and B. A. Price, editors. *Software Visualization.* MIT Press, Cambridge, MA, 1998.

[36] G. Tomas and C. W. Ueberhuber. *Visualization of Scientific Parallel Programs.* Lecture Notes in Computer Science, Vol. 771. Springer-Verlag, New York, 1994.

[37] E. R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, Cheshire, CT, 1983.

[38] P. H. Worley. A new PICL trace file format. Technical Report ORNL/TM-12125, Oak Ridge National Laboratory, Oak Ridge, TN, October 1992.

[39] P. H. Worley and J. B. Drake. Parallelizing the spectral transform method. *Concurrency: Practice and Experience*, 4(4), June 1992, pages 269–291.

Appendix A. Unix man page

NAME

ParaGraph - performance visualization for MPI programs

SYNOPSIS

PG [-c | -g | -m] [-d hostname:0.0] [-e envfile] [-f tracefile] [-o orderfile] [-r rgbfile]

DESCRIPTION

ParaGraph is a graphical display tool for visualizing the behavior and performance of parallel programs that use MPI (Message-Passing Interface). The visual animation of a parallel program is based on execution trace information gathered during an actual run of the program on a message-passing parallel computer system. The resulting trace data are replayed pictorially to provide a dynamic depiction of the behavior of the parallel program, as well as graphical summaries of its overall performance. The same performance data can be viewed from many different visual perspectives to gain insights that might be missed by any single view. The necessary execution trace data are produced by MPICL (available from netlib), which uses the profiling interface of MPI.

ParaGraph runs on any workstation supporting the X Window System. It uses no X toolkit and requires only Xlib. Although most effective in color, it also works on monochrome and grayscale monitors. It has a graphical, menu-oriented user interface that accepts user input via mouse clicks and keystrokes. The execution of ParaGraph is event driven, including both user-generated X Window events and trace events in the input data file provided by MPICL. Thus, ParaGraph displays a dynamic depiction of the parallel program while also providing responsive interaction with the user. Menu selections determine the execution behavior of ParaGraph both statically (e.g., initial selection of parameter values) and dynamically (e.g., pause/resume, single-step mode). ParaGraph preprocesses the input tracefile to determine relevant parameters (e.g., time scale, number of processors) automatically before the graphical simulation begins, but these values can be overridden by the user, if desired.

ParaGraph provides about 30 different displays or views, all based on the same underlying trace data, but each giving a distinct perspective. Some of these displays change dynamically in place, with execution time in the original run represented by simulation time in the replay. Other displays represent execution time in the original run by one space dimension on the screen. The latter displays scroll as necessary (by a user-controllable amount) as visual simulation time progresses. The user can view as many of the displays simultaneously as will fit on the screen, and all visible windows are updated appropriately as the tracefile is read. The displays can be resized within reasonable bounds. Most of the displays depict up to 512 processors in the current implementation, although a few are limited to 128 processors and one is limited to 16.

ParaGraph is extensible so that users can add new displays of their own design that can be viewed along with those views already provided. This capability is intended primarily to support application-specific displays that augment the insight that can be gained from the generic views provided by ParaGraph. Sample application-specific displays are supplied with the source code. If no user-supplied display is desired, then dummy "stub" routines are linked with ParaGraph instead.

The ParaGraph source code comes with several sample tracefiles for use in demonstrating the package and verifying its correct installation. To create your own tracefiles for viewing with ParaGraph, you will need MPICL, which is also available from netlib. The tracing option of MPICL produces a tracefile with records in node order. For graphical animation with ParaGraph, the tracefile must be sorted into time order, which can be accomplished with the Unix sort command:

```
% sort +2n -3 +1rn -2 +0rn -1 tracefile.raw > tracefile.trf
```

When using MPICL to produce tracefiles for viewing with ParaGraph, set tracelevel(1,1,0) to produce the trace events required by ParaGraph. You may also want to define tasks using the traceevent command of MPICL to delimit sections of code and assign them task numbers to be depicted by ParaGraph in some of its displays as an aid in correlating the

visual simulation with your parallel program. ParaGraph does not depict a distinguished HOST processor graphically and ignores all trace events involving such a HOST, if any.

OPTIONS

The following command-line options are supported by ParaGraph.

-c to force color display mode.
-d to specify a hostname and screen (e.g., hostname:0.0) for remote display across a network.
-e to specify an initialization file (default: .pgrc).
-f (or no switch) to specify a tracefile name.
-g to force grayscale display mode.
-m to force monochrome display mode.
-o to specify an order file (default: .pgorder).
-r to specify a file containing RGB values of task colors (default: .pgcolors).

By default, ParaGraph automatically detects the appropriate display mode (color, grayscale, or monochrome), but a particular display mode can be forced, if desired, by the corresponding command-line option. This facility is useful, for example, in making black-and-white hardcopies from a color monitor.

FILES

The following environment files can optionally be supplied by the user to customize ParaGraph's appearance and behavior. The default filenames given below can be changed by the appropriate command-line options.

.pgrc	defines the initial state of ParaGraph upon invocation, including which menus and displays are open and various options.
.pgorder	allows the user to define an optional order or alternative names for the processors.
.pgcolors	allows the user to define the color scheme to be used for identifying tasks.

The following files are provided in the ParaGraph distribution from netlib.

*.c	several C source files.
*.h	several include files.
Makefile.*	sample makefiles for several machine configurations, which should be modified to incorporate the local location for Xlib, etc.

manual.ps	a user guide in postscript format.
pg.man	a Unix man page.
tracefiles	a directory containing several sample tracefiles.
u_*	several directories containing example application-specific displays.

SEE ALSO

A user manual for ParaGraph is provided in postscript form along with the source code from netlib. Additional information is contained in the article "Visualizing Performance of Parallel Programs" in the September 1991 issue of IEEE Software, pages 29-39, and in the technical report ORNL/TM-11813. Documentation for MPICL is available from netlib and in the technical reports ORNL/TM-11130 and ORNL/TM-11616.

BUGS

Some of the displays are not restored when re-exposed after having been partially obscured. Changing parameters dynamically while the visual animation is active may give erratic results. The apparent speed of visual animation is determined primarily by the drawing speed of the workstation and is not necessarily uniformly proportional to the original execution speed of the parallel program.

AUTHORS

Michael T. Heath and Jennifer E. Finger

Appendix B. MPICL tracing commands

```
------------------------------------------------------------------------
clocksync0()

Synchronize clocks.  Calculates clock offset and drift and synchronizes
processors.  First call sets clock start time, and all subsequent
timestamps are normalized accordingly.  Subsequent calls resynchronize
but do not alter clock start time.

If synchronization option is selected in call to tracenode, clocksync0
is called internally by MPICL at beginning and end of run to
synchronize clocks and adjust for offset and drift.  Explicit calls to
clocksync0 by user are not normally needed, but may be helpful if
tachyons are encountered due to inconsistent timestamps despite normal
synchronization.

------------------------------------------------------------------------
tracedata(eventid, dataid, datatype, size, data)
   int eventid, dataid, size;
   char *datatype, *data;

Record user data.

eventid : user defined nonnegative integer.
dataid  : user defined integer.
datatype: "character", "integer", "long", "float", "real", or "double".
size    : number of data items to be written
data    : array containing data items to be written

This routine needed with ParaGraph only to supply data for user-defined
displays, if any.

------------------------------------------------------------------------
traceevent(eventtype, eventid, nparams, params)
   char *eventtype;
   int eventid, nparams, *params;

Record user-defined event.

eventtype: "entry", "exit", "mark", "label", or "message".
eventid  : user-defined nonnegative integer.
nparams  : number of parameters (may be zero if none).
params   : array of parameters (may be absent if nparams==0).
```

```
Task displays of ParaGraph take "entry" record to denote beginning of
task eventid and "exit" record to denote end of task eventid.  If
present, first additional parameter of "exit" record is taken by
ParaGraph to denote amount of "work" (in user-chosen units) performed
during task eventid.  If omission of additional argument causes
problems, e.g., with some Fortran/C interfaces, then NULL value should
be explicitly supplied by user.

Other values of eventtype, such as "mark" or "message", can be used to
indicate particular milestones in the user's program.  ParaGraph does
not depict these graphically, but can optionally pause playback when
such an event is encountered.

------------------------------------------------------------------------
tracefiles(tempfile, permfile, verbose)
   char *tempfile, *permfile;
   int verbose;

Specify tracefile name and type.

tempfile: name of file (optionally including directory path prefix) for
   temporary storage of trace data.  Empty string usually suffices, but
   other values can be useful for systems with disks attached directly to
   node processors.  Node number is appended as suffix to make temporary
   filename unique.
permfile: name of file (optionally including directory path prefix) in
   which trace data will be written for "permanent" storage.
verbose: flag to select compressed or annotated trace records.
   == 0 : compressed trace records are output.
   != 0 : annotated trace records are output.

Typical setting for ParaGraph is tracefiles("", "tracefile", 0).

ParaGraph supports only compressed format trace records, so verbose
must be zero.  Note that Trace display of ParaGraph supplies annotation
for compressed tracefile.  Final tracefile is written in node order and
must be sorted into time order for use in ParaGraph:

   % sort +2n -3 +1rn -2 +0rn -1 tracefile.raw > tracefile.sorted

------------------------------------------------------------------------
tracelevel(mpicl,user,trace)
   int mpicl,user,trace;
```

```
Specify types and levels of event data recorded.

Types:
   mpicl: MPI and low-level mpicl events.
   user : user-specified events and high-level mpicl events.
   trace: tracing commands.

Levels:
   < 0 : tracing disabled.
   >= 0: statistics collected.
   > 0 : event records generated.

Correct setting for ParaGraph is tracelevel(1, 1, 0).

---------------------------------------------------------------------------
tracenode(tracesize, flush, sync)
   int tracesize, flush, sync;

Initialize tracing.  Should be called only once by each node.

tracesize: number of bytes to allocate in local memory for storing
   trace data.
flush: action to be taken if trace buffer becomes full.
   == 1: write trace buffer to secondary storage and flush.
   == 2: overwrite trace buffer.
   else: stop tracing.
sync: clock synchronization option.
   == 0, no clock synchronization.
   == 1, call clocksync0 to synchronize processor clocks.

Typical setting for ParaGraph is tracenode(100000, 0, 1).

Flushing trace buffer during run is generally very disruptive and
should be avoided by providing adequate trace buffer space.
Synchronization option is usually required to yield meaningful
timestamps for use with ParaGraph.

---------------------------------------------------------------------------
open0 and close0 routines of PICL are unnecessary with MPICL, as their
functionality is provided by MPI_Init and MPI_Finalize, respectively.

Collective communication functions of PICL, such as bcast0 and gsum0,
still work in MPICL, but equivalent functionality is provided by the
collective communication functions of MPI, such as MPI_Bcast and
MPI_Reduce.
```

Appendix C. Example code using MPICL tracing

```
/*
      Example "node code" using MPICL tracing with MPI to generate
      tracefile for viewing with ParaGraph.
*/

#include <mpi.h>
#include <mpiprof.h>
#include <binding.h>

main(argc, argv)
      int argc;
      char **argv;
{
      int k, myid, nprocs, ntasks, work;

      MPI_Init(&argc, &argv);

      MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
      MPI_Comm_rank(MPI_COMM_WORLD, &myid);

      tracefiles("", "tracefile", 0);           /* mandatory */
      tracelevel(1, 1, 0);                      /* mandatory */
      tracenode(100000, 0, 1);                  /* mandatory */

      for (k = 0; k < ntasks; k++) {
            traceevent("entry", k, 0);          /* optional */
            .
            .
            /* code for task k, including various MPI calls */
            .
            .
            work = work done in task k;         /* optional */
            traceevent("exit", k, 1, &work);    /* optional */
      }

      MPI_Finalize();
}
```

Appendix D. MPICL event record formats

Detailed knowledge of event record formats is not needed for normal use of ParaGraph. It is provided here as a reference for users who wish to write optional application-specific displays and thus may need to know the specific information available in each event record. In addition, though there should rarely be a need for a user to read raw tracefiles directly since ParaGraph provides an annotated display for this purpose, the information provided here may be helpful in "debugging" faulty tracefiles that cannot be viewed with ParaGraph.

```
------------------------------------------------------------------------
OPENKEY             -11
CLOCKSYNCKEY        -401
RELPROCSKEY         -502
IDLEKEY             -601
SYSTEMOHKEY         -602
USEROHKEY           -603

EVENTENTRY:   -3 key time node pid 0
EVENTEXIT:    -4 key time node pid 0
------------------------------------------------------------------------
SENDKEY             -21
BSENDKEY            -22
SSENDKEY            -23
RSENDKEY            -24

EVENTENTRY:   -3 key time node pid 3 type lth tag dest
EVENTEXIT:    -4 key time node pid 0
------------------------------------------------------------------------
SENDBEGKEY          -27
BSENDBEGKEY         -33
SSENDBEGKEY         -39
RSENDBEGKEY         -45

EVENTENTRY:   -3 key time node pid 3 type lth tag dest
EVENTEXIT:    -4 key time node pid 1 type req
------------------------------------------------------------------------
SENDDONEKEY         -28
SENDENDKEY          -30
SENDENDBLOCKKEY     -31
BSENDDONEKEY        -34
BSENDENDKEY         -36
BSENDENDBLOCKKEY    -37
SSENDDONEKEY        -40
SSENDENDKEY         -42
SSENDENDBLOCKKEY    -43
RSENDDONEKEY        -46
```

```
RSENDENDKEY         -48
RSENDENDBLOCKKEY    -49

EVENTENTRY:   -3 key time node pid 1 type req
EVENTEXIT:    -4 key time node pid 0
-------------------------------------------------------------------
RECVKEY             -51
RECVBLOCKKEY        -52
WAITBLOCKKEY        -56

EVENTENTRY:   -3 key time node pid 2 type tag src
EVENTEXIT:    -4 key time node pid 3 type lth tag src
-------------------------------------------------------------------
RECVBEGKEY          -57

EVENTENTRY:   -3 key time node pid 2 type tag src
EVENTEXIT:    -4 key time node pid 1 type req
-------------------------------------------------------------------
RECVDONEKEY         -58
RECVENDKEY          -60
RECVENDBLOCKKEY     -61

EVENTENTRY:   -3 key time node pid 1 type req
EVENTEXIT:    -4 key time node pid 3 type lth tag src
-------------------------------------------------------------------
COMMCREATEKEY       -804
COMMDUPKEY          -805
COMMSPLITKEY        -812
INTERCOMMCREATEKEY  -826
INTERCOMMMERGEKEY   -827
CARTCREATEKEY       -841
CARTSUBKEY          -846
GRAPHCREATEKEY      -849

EVENTENTRY:      -3 key time node pid 1 type oldgrpid
EVENTEXIT:       -4 key time node pid 2 type newgrpid grpsize
PROCESSSUBSET: -202 key time node pid grpsize+1 type newgrpid mem1 mem2 ...
-------------------------------------------------------------------
BARRIERKEY          -402

EVENTENTRY:   -3 key time node pid 1 type grpid
EVENTEXIT:    -4 key time node pid 0
-------------------------------------------------------------------
COMMFREEKEY         -806
```

```
EVENTENTRY:   -3 key time node pid 1 type grpid
EVENTEXIT:    -4 key time node pid 0
-------------------------------------------------------------------
ALLGATHERKEY        -780
ALLREDUCEKEY        -782
ALLTOALLKEY         -783
REDUCESCATTERKEY    -791
SCANKEY             -792

EVENTENTRY:   -3 key time node pid 2 type lth grpid
EVENTEXIT:    -4 key time node pid 1 type lth
-------------------------------------------------------------------
BCASTKEY            -785
GATHERKEY           -786
REDUCEKEY           -790
SCATTERKEY          -793

EVENTENTRY:   -3 key time node pid 3 type lth root groupid
EVENTEXIT:    -4 key time node pid 1 lth
-------------------------------------------------------------------
FILEOPENKEY         -201

EVENTENTRY:   -3 key time node pid filename
EVENTEXIT:    -4 key time node pid 1 type channelid
-------------------------------------------------------------------
FILECLOSEKEY        -202

EVENTENTRY:   -3 key time node pid 1 type channelid
EVENTEXIT:    -4 key time node pid 0
-------------------------------------------------------------------
READKEY             -251

EVENTENTRY:   -3 key time node pid 0
EVENTEXIT:    -4 key time node pid 2 type lth channelid
-------------------------------------------------------------------
WRITEKEY            -221

EVENTENTRY:   -3 key time node pid 2 type lth channelid
EVENTEXIT:    -4 key time node pid 0
-------------------------------------------------------------------
GETPROCSKEY         -501

EVENTENTRY:   -3 key time node pid 1 type num_procs
EVENTEXIT:    -4 key time node pid num_procs
-------------------------------------------------------------------
```

```
LOADKEY             -503

EVENTENTRY:   -3 key time node pid filename
EVENTEXIT:    -4 key time node pid 2 type newnode newpid
-------------------------------------------------------------------------
PACKKEY             -756

EVENTENTRY:   -3 key time node pid 1 type lth
EVENTEXIT:    -4 key time node pid 0
-------------------------------------------------------------------------
UNPACKKEY           -773

EVENTENTRY:   -3 key time node pid 0
EVENTEXIT:    -4 key time node pid 1 type lth
-------------------------------------------------------------------------
USEREVENTKEY        user defined, nonnegative number

EVENTMARK:    -2 key time node pid nparams type param1 param2 ...
EVENTENTRY:   -3 key time node pid nparams type param1 param2 ...
EVENTEXIT:    -4 key time node pid nparams type param1 param2 ...
EVENTLABEL:   -5 key time node pid labellth type labelstring
EVENTMESSAGE: -7 key time node pid msglth type msgstring
-------------------------------------------------------------------------
```