ParaGraph: A Tool for Visualizing Performance of Parallel Programs *

Michael T. Heath, University of Illinois
Jennifer Etheridge Finger, Oak Ridge National Laboratory

September 1, 1994

Abstract

ParaGraph is a graphical display system for visualizing the behavior and performance of parallel programs on message-passing multicomputer architectures. The visual animation is based on execution trace information monitored during an actual run of a parallel program on a message-passing parallel computer. 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. Many different visual perspectives are provided from which to view the same performance data, in an attempt to gain insights that might be missed by any single view. We describe this visualization tool, outline the motivation and philosophy behind its design, and provide details on how to use it.

*This user guide is adapted from a technical report [6], which also appears in a modified form in [7]. The principal modifications to produce this manual were the omission of illustrations and most bibliographic citations, and the addition of a considerable amount of detail to aid in using and understanding ParaGraph. This edition also reflects many new features recently added to ParaGraph, and is up to date as of August 30, 1993.

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 on advanced computer architectures 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. We have developed such a software tool, called ParaGraph, that provides a detailed, dynamic, graphical animation of the behavior of message-passing parallel programs, as well as graphical summaries of their performance. The purpose of this document is to explain how to use the visualization tool to analyze parallel programs.

1.1 Graphical Simulation

For lack of a better term, we will often use the word "simulation" to refer to the graphical animation of a parallel program. The use of this term should not be taken to suggest that there is anything artificial about the programs or their behavior as we portray them. ParaGraph displays the behavior and performance of real parallel programs running on real parallel computers to solve real problems. In effect, ParaGraph simply provides a visual replay of the events that actually occurred when a parallel program was run on a parallel machine.

To date, ParaGraph has been used only in such a "post processing" manner, using a tracefile created during the execution of the parallel program and saved for later study. But the design of the package does not rule out the possibility that the data for the visualization could 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 the current generation of distributed-memory 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 processor and the graphical workstation, as well as the drawing speed of the workstation, are usually inadequate to handle the extremely high data transmission rates that would be required for real-time display. Finally, even if these other limitations were not a factor, human visual perception would be hard pressed to digest a detailed graphical depiction as it flies by in real time. One of the strengths of ParaGraph is the insight that can be gained from repeated replays 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. We see the tracefile as a script to be played out, visually re-enacting 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 the simulation. 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 backward), 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 simulation speed (as will be discussed in greater detail below).

1.2 Design Goals

In designing ParaGraph, our principal goals were:

- ease of understanding,
- ease of use, and
- portability.

We now briefly discuss each of these goals in turn.

1.2.1 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 useful 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 in fact provides some twenty-five different displays or views, all based on the same underlying execution trace data.

1.2.2 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. It certainly seems reasonable that any graphics package should have a graphical user interface. 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. ParaGraph currently takes its input data from execution tracefiles produced by PICL (Portable Instrumented Communication Library [3, 4]), which enables the user to produce such trace data automatically. We have also tried to keep the user's learning curve for ParaGraph very short, even at the expense of limiting 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.

1.2.3 Portability

There are two senses in which portability is important in the present context. One is that the graphics package itself be portable. ParaGraph is based on the X Window System, and thus runs on a wide variety of scientific workstations from many different vendors. 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. ParaGraph has been tested with all MIT distributions of X11 through X11R5, as well as several vendor-supplied versions of X Windows. 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. A second sense in which portability is important is that the package be capable of displaying execution behavior from different parallel architectures and parallel programming paradigms. ParaGraph inherits a high degree of such portability from PICL, which runs on parallel architectures from a number of different vendors (e.g., Cogent, Intel, Meiko, Ncube, Symult, Thinking Machines). On the other hand, many of the displays in ParaGraph are based on a message-passing paradigm, and thus the package does not currently offer support for displaying the behavior of programs based explicitly on shared-memory constructs. Further comments on the programming model supported by ParaGraph are given below.

1.3 Relationship to Previous Work

ParaGraph is certainly not the first software tool to be developed for visualizing parallel programs. Graphical animation techniques for visualizing serial algorithms have received considerable study. Visualization of parallel computations has been the subject of a number of Ph.D. theses, technical articles, and books. Graphical visualization has also been an important component of several environments that have been developed for parallel programming, debugging, and monitoring, as well as integrated environments that combine several of these components. Algorithm visualization tools have also been developed for specific applications, such as matrix computations. See [6] for an extensive list of references on much of this prior work. ParaGraph is a general-purpose performance visualization tool that is distinguished from most previous efforts in the following ways:

- The multiplicity of displays provided by ParaGraph is unique. Other packages have emphasized the importance of multiple views, but ParaGraph provides a substantially greater variety of perspectives than any other package of which we are aware. Some of the displays we have incorporated into ParaGraph appear to be original, while others have been motivated by similar displays found in previous packages.

- Many previous packages for visualizing parallel programs have targeted a particular parallel architecture and/or been based on a proprietary graphical display system. ParaGraph is applicable to any parallel architecture having message passing as its programming paradigm, and ParaGraph itself is based on the X Window System, which is widely available on workstations from many vendors.

- We have tried to attain high standards in the intuitive appeal and aesthetic quality of the displays provided by ParaGraph, including both the new displays we have devised and the display concepts that we have borrowed from previous packages. Of course, the perceived success of this attempt is in the eye of the beholder and can be judged only by users.

- We have also tried to make ParaGraph exceptionally easy to use, both through its interactive, graphical user interface and by relying on an instrumented communication library (PICL) to provide the requisite trace data without requiring the user to instrument explicitly the parallel program under study. We have also emphasized a short learning curve, minimal data manipulation, and views that are already tuned to the specific task at hand.

- Another unusual feature of ParaGraph is its extensibility. ParaGraph provides a mechanism for users to add new displays of their own design that can be viewed along with the other displays already provided. This capability is intended primarily to support special-purpose displays for particular applications, and is described in more detail below.

An indication of our degree of success in making ParaGraph easy to use and easy to understand is the fact that many users obtained an early version from `netlib@ornl.gov` [1] over the Internet and were able to build the program at their locations and use it effectively without the benefit of any documentation beyond a one-page README file. For some case studies of performance tuning with ParaGraph, see [5, 8, 9].

1.4 Relationship to PICL

PICL is a Portable Instrumented Communication Library [3, 4] that is available from `netlib@ornl.gov` and runs on a variety of message-passing parallel architectures. As its name implies, PICL provides both portability and instrumentation for programs that use its communication facilities for passing messages between processors. On request, PICL provides a tracefile that records important events in the execution of the user's parallel program (e.g., sending and receiving messages). The tracefile contains one event record per line, and each event record consists of a set of integers that specify the event type, timestamp, processor number, message length, and other similar information. The format of the tracefile is documented in the PICL reference manual [3]. To obtain further information about PICL, including documentation and source code, send email to `netlib@ornl.gov` containing the message `send index from picl`.

ParaGraph has a producer-consumer relationship with PICL: ParaGraph consumes trace data produced by PICL. By using PICL rather than the "native" parallel programming interface for a particular machine, the user gains portability, instrumentation, and the ability to use ParaGraph in analyzing the behavior and performance of the parallel program. These benefits are essentially "free" in that once the parallel program is implemented using PICL, no further changes are required to the source code to move it to a new machine (provided PICL is available on the target machine), and little or no effort is required to instrument the program for performance analysis. On the other hand, since ParaGraph's dependence on PICL is solely for its input data, ParaGraph could in fact work equally well with any other source of data having the same format and semantics. Thus, other message-passing systems could be instrumented to produce trace data in the format expected by ParaGraph, or else ParaGraph's input routine could be adapted to a different input format. In this manner, ParaGraph can be, and indeed has been, used in conjunction with communication systems other than PICL. Several vendors of parallel computers have instrumented their native messaging systems to produce PICL-compatible tracefiles, which can then be viewed with ParaGraph.

For a meaningful simulation, the timestamps of the events should be as accurate and consistent across processors as possible. This is not necessarily easy to accomplish on a machine in which each processor may have its own clock with its own starting time, 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, PICL goes to considerable lengths to synchronize the processor clocks, and also to adjust for potential clock drift, so that the timestamps will be as consistent and meaningful as possible [2]. On some machines, PICL actually provides a higher resolution clock than the one supplied by the system vendor. Fortunately, the trend is towards more accurate clocks and centralized clock pulses in distributed parallel machines, so clock consistency should be less of a problem in the future.

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. PICL 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 PICL the clock calls necessary to determine the timestamps for the event records, plus other minor overhead, add 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. In general, we believe that this perturbation is small enough that the behavior of parallel programs is not fundamentally altered. It is certainly true that in our experience the lessons we learn from visual study of instrumented runs invariably lead to improved performance of uninstrumented runs.

2 Programming Model

The most fundamental restriction in the parallel programming model currently supported by ParaGraph (and PICL) is the assumption that there is only one user process per processor. Such a programming style is typical of SPMD (single program multiple data) programs for multicomputers and is adequate for the vast majority of applications on these machines. Nevertheless, this restriction is occasionally limiting, and it will probably be relaxed in a future release of ParaGraph.

The interprocessor communication model supported by ParaGraph is typical of interrupt-driven, loosely synchronous message passing systems from a variety of vendors. If a message is sent before it has been requested by the receiving user process, then the message is buffered and queued by the communication system. When the receiving user process subsequently requests the message, its contents are then transferred from a system buffer into an array provided by the user process. If a message is requested before it has arrived, then the receiving user process blocks further execution until the requested message arrives, at which point the message is transferred into the receiving array in the user process and execution resumes. In any case, the sending user process resumes execution immediately after the send and does not block. Currently, ParaGraph does not explicitly support fully synchronous communication (in which the sender blocks until the receive is executed) or fully asynchronous communication (in which neither sender nor receiver blocks, and probes must be done to detect whether messages have arrived). It may still be possible for ParaGraph to visualize programs that use such communication, but the visual simulation and the figures of merit produced may not be very accurate. Explicit support for these and other communication protocols may be added to ParaGraph in the future.

It is assumed that each message has an integer type or tag. The value for the type is assigned by the sending user process and can be used by the receiving user process to search the queue of incoming messages selectively for a particular kind of message. Such tags can play an important role in synchronization and control of the parallel program.

Some message passing systems support "global sends," that is, messages that go from one processor to all other processors as a result of a single send command. Such a global send is usually indicated by a special value, such as -1, for the destination in the usual send function. The actual implementation of such global sends varies from one machine to another, depending on the most efficient way of using a given interconnection network topology (e.g., a spanning tree in a hypercube). ParaGraph handles global sends (i.e., send event records with destination -1) as if there were a whole set of separate messages, one for each possible destination processor, with each message having the same source and contents. This may or may not accurately reflect how the given communication system actually implements global sends, but is about all that ParaGraph can do in representing global sends pictorially, given its lack of knowledge of the underlying interconnection network topology. Note that PICL includes a special routine, `bcast0`, for global broadcasting that is optimized for various particular topologies, and this routine in turn generates individual send and receive event records, which are therefore accurately depicted by ParaGraph. For this reason, as well as for greater portability, the user may prefer to use the explicit broadcasting facility of PICL rather than a simple send with a special destination value.

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 in fact 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. Thus, ParaGraph must alternate between these two queues to provide both a dynamic depiction of the parallel program and responsive interaction with the user. Menu selections 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 18,000 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 the simulation. On the other hand, the trace event queue should be processed as rapidly as possible while the simulation is active, but need not be checked at all if the next possible event must be an X event (e.g., before the simulation starts, after the simulation finishes, when in single-step mode, or when the simulation 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 PICL 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 the simulation is inactive.

The relationship between the apparent simulation speed and the original execution speed of the parallel program is necessarily somewhat imprecise. The speed of the graphical simulation 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 simulation 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 some graphical depiction of that event on the workstation screen have no particular correlation with each other. 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 simulation 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 simulation 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 simulation, since it is already drawing as fast as it can. However, a "slow-motion" slider is provided for precise control over the simulation speed if slower execution is desired without resorting to opening additional displays. Further, one can always use single-step mode if arbitrarily slow drawing speed is desired for very close 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,
- `-e` to specify an environment file (default: `.pgrc`),
- `-f` (or no switch) to specify the path name 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 not normally necessary to specify the display mode (color, grayscale, or monochrome), as ParaGraph by default will detect 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 environment file, if present, defines the initial selection of displays and options with which ParaGraph begins execution. Typically, such an environment 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 path name in the subwindow provided in the `tracefile` menu. If the path name of a tracefile is specified on the command line, then the directory portion of that path name 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 the graphical simulation 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 will issue an error message and terminate before going into the graphical simulation. To produce the necessary trace records for optimal use in ParaGraph, tracing in PICL should be done with `tracelevel(4,4,0)`. For graphical animation with ParaGraph, the tracefile needs to be sorted into time order. Since the tracing option of PICL produces a tracefile with records in node order, the necessary reordering can be accomplished with the Unix sort command:

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

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 the appropriate subwindow button. 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 environment file for use in subsequent ParaGraph sessions, as explained below.

The `tracefile` menu provides a graphical means for browsing a directory from which to select a desired tracefile. If a directory path name is supplied on the command line when invoking ParaGraph, then it will appear in the `path name` subwindow; otherwise, the current working directory when ParaGraph is invoked is taken as the default directory. A new path name can be typed into the `path name` subwindow of the `tracefile` menu at any time. The filenames in the given directory are displayed, and the user selects the desired tracefile (or another directory) by clicking the mouse pointer 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 the graphical simulation 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 the simulation. 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 ending of the tracefile), or the simulation can be optionally stopped each time a user-specified event occurs in the tracefile. The entire animation can be restarted at any time (whether in the middle or at the end of the tracefile) simply 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 the simulation speed. Clicking or dragging the mouse cursor along the slider slows down execution as much as desired. The position of the slider can be altered 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 environment 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 environment 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 environment 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 [6, 7]. Some of the displays change in place dynamically as events occur, with execution time in the original run represented by simulation 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 simulation 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 work is distributed across the processors.

5.1.1 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 the simulation 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, we show green at the bottom, yellow in the middle, and red at the top along the vertical axis. At any given time, ParaGraph categorizes each processor as *idle* if it has suspended execution awaiting a message that has not yet arrived (or if it has ceased execution at the end of the run), *overhead* if it is executing in the communication subsystem (but not awaiting a message), and *busy* if it is executing some portion of the program other than the communication subsystem. Since the three categories are mutually exclusive and exhaustive, the total height of the composite is always equal to the total number of processors.

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 overhead associated with communication. However, we find that the definitions we have adopted based on the data provided by PICL are quite adequate in practice to convey the effectiveness of parallel programs pictorially.

5.1.2 Gantt Chart

This display, which is patterned after graphical charts used in industrial management, 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 the simulation 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.

5.1.3 Kiviat Diagram

This display, which is adapted from related graphs used in other types of performance evaluation, 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).

5.1.4 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 the run, 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.

5.1.5 Utilization Summary

This display shows the cumulative percentage of time, over the entire run, that each processor 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.

5.1.6 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.

5.1.7 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 the mouse on the appropriate subwindow. This display is defined only at the end of the run. 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, where 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.

5.2.1 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.

5.2.2 Spacetime Diagram

This display is patterned after the diagrams used in physics, particularly in relativity theory, to depict interactions between particles through space and time. This type of diagram has been used by Lamport 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. 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 slopes of the resulting lines give a visual indication of how soon a given piece of data produced by one processor was needed by the receiving processor.

The communication lines are color coded according to the Color Code display (see below). Each message line is drawn when 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 processor that was previously blocked 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.

5.2.3 Message Queues

This display depicts the size of the queue of incoming messages for each processor by a vertical bar whose height varies with time as messages are sent, buffered, and received. The processor number is shown on the horizontal axis. At the user's option, the queue size can be measured either by the number of messages or by their total length in bytes. The input queue size for a given processor is incremented each time a message is sent to that processor, and decremented each time the user process on that processor receives a message.

On most message-passing parallel 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 the destination processor's input queue. Of course, depending on message types, the messages may not be received in the same order in which they arrive for queuing, 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 in a parallel program (i.e., whether messages are accumulating in the input queue), or the messages are being consumed at about the same rate as they arrive. Of course, it is best if messages arrive slightly before they are actually needed, so that the receiving processor does not become idle awaiting a message. But a large backlog of incoming messages can consume excessive buffer space, so a happy medium (analogous to "just in time" manufacturing) is desirable.

5.2.4 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 the simulation, 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 the simulation, the Communication Matrix display shows the cumulative statistics (e.g., communication volume) for the entire run between each pair of processors, depending on the particular choice of color code.

5.2.5 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.

5.2.6 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, idle, sending, receiving) is indicated by its color, so that the circles can be thought of as the "front-panel lights" of the parallel computer. A line is drawn between the source and destination processors when each message is sent, and erased when the message is received. Thus, both the colors of the nodes and the connectivity of the graph change dynamically as the simulation proceeds. The lines represent the logical communication structure of the parallel program and do not necessarily reflect the actual interconnectivity of the underlying physical network. 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.

5.2.7 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 the user of a hypercube to determine if the network's physical connectivity is correctly honored by the communication in the parallel program, message arcs corresponding to genuine physical hypercube links are drawn in a different color from message arcs along "virtual" links that do not exist in a hypercube and therefore entail indirect 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.

5.2.8 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, 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. 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 always 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 Intel Paragon (mesh) can see a visual simulation of the behavior his program might have on a Thinking Machines CM-5 (quadtree), or vice versa.

The node numbers of the peripheral nodes are always shown, but to avoid excessive clutter amid the message lines, by default the the interior node numbers are omitted in the multistage and mesh networks. 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.

5.2.9 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, type, 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.

5.2.10 Color Code

This display permits the user to select which statistic will determine the color 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 type. 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. To remedy this situation, we considered a number of automated approaches to providing such information (e.g., picking up line numbers in the source code from the compiler), but all of these encounter nasty practical difficulties (such as dealing with multiple source files). Thus, we reluctantly made an exception to our rule that the user need do nothing to instrument the parallel program under study in order to use ParaGraph.

We developed a number of "task" displays that use information provided by the user, with the help of PICL, to depict the portion of the user's parallel program that is executing at any given time. Specifically, the user defines "tasks" within the program by using special PICL routines 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, a loop, an entire subroutine, or any other unit of work 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 defined simply by calling PICL's `traceblockbegin` and `traceblockend` routines, with the desired task number as argument, immediately before and after the selected section of code. This causes PICL to produce event records that are interpreted appropriately by ParaGraph to depict the given task, using displays to be described in this section. We emphasize that 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.

Tasks can be nested, one inside another, but if so these should be properly bracketed by matching task begin and end 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); indeed, the model we have in mind is that all processors collaborate on each task, rather than that each task is assigned to a single processor. In many contexts, such as the matrix factorization example mentioned above, there is a natural ordering and corresponding numbering of the tasks in a parallel program.

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.

5.3.1 Task Count

During the simulation, 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 the run, 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).

5.3.2 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 the simulation proceeds. This display can be compared with the Utilization Gantt chart to correlate busy/overhead/idle status with the task information.

5.3.3 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.

5.3.4 Task Summary

This display, which is defined only at the end of the simulation run, 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.

5.4 Other Displays

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

5.4.1 Clock

This display provides both digital and analog clock readings during the graphical simulation of the parallel program. The current simulation time is shown as a numerical reading, and the proportion of the full 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.

5.4.2 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 records are drawn in this display one at a time, space is allowed to show several consecutive trace records 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.

5.4.3 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 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), transit time, and overhead incurred for both incoming and outgoing messages. While this tabular display may yield less insight than the graphical displays provided by ParaGraph, exact numerical quantities are occasionally useful in preparing tables and graphs for printed reports, or for analytical performance modeling. Due to limited space on the screen, this display shows data for at most 16 processors at a time, but the subset of processors shown can be varied by clicking the mouse on the subwindow provided, which enables one to browse the entire data array.

5.4.4 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.

5.4.5 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 computation. 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.

5.4.6 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.

5.4.7 Coordinate Information

The Info display is a non-graphical display used to write information produced by mouse clicks on the other displays. Many of the displays respond to mouse clicks by printing in the 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 a mouse button with the cursor placed on one of the nodes in the Animation display causes the following information to be printed in the Info display: node number, task number (if any), number of incoming messages pending, 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 we obviously could not provide such application-specific displays as part of ParaGraph, we instead made ParaGraph 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 we use 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. Thus, the user-supplied displays can receive input interactively via the mouse or keyboard.

The usual events generated by PICL 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 `tracemarks` event of PICL is perhaps the most useful for this purpose, as it allows the user to insert into the tracefile timestamped records containing arbitrary lists of integers, which might be used to provide loop counters, array indices, memory addresses, identifiers of 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 System programming. But at least the potential user of this capability can concentrate on only those portions of the graphics programming that are relevant to his application, taking advantage of the supporting infrastructure of ParaGraph to provide all of the other necessary facilities to drive the overall graphical simulation. As an aid to users who may wish to develop application-specific displays to add to ParaGraph, we have developed several prototype displays for depicting such applications as parallel sorting algorithms, recursive matrix transposition, general matrix computations, and graph algorithms, and for displaying operation counts (e.g., flops, particles, transactions). These example routines are distributed along with the source code for ParaGraph.

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. Hitting 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 simulation time advances. Smooth scrolling provides an appealing sense of visual continuity, but results in a slower drawing speed.

- Time Unit: The relationship between simulation time and the timestamps of the trace events is determined by the `time unit` chosen. By convention, PICL provides event timestamps in microseconds. Thus, for example, a value of 100 for the time unit in ParaGraph means that each "tick" of the simulation clock corresponds to 100 microseconds in the original execution of the parallel program (i.e., the timestamps in the tracefile are divided by 100). 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 simulation 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 simulation 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 simulation in its entirety. Once the specified stopping time is reached, the simulation 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, 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 simulation time units and it can be changed simply by typing a new value.

- Pause on Tracemark/msg: Another way to stop the simulation 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 provided by PICL 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 simulation 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 the simulation is paused if an error is detected, such as a mismatched send/receive pair or incorrectly nested blocks. Again, the simulation 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 Record 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 the simulation. 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 path name 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 the simulation run, 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 can quickly consume large amounts of disk space and will require a great deal of time for ParaGraph to preprocess and then animate visually. Fortunately, in our experience, basic algorithm behavior and most 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 PICL's tracing commands only for the portion of immediate interest.

On some machines, a more insidious problem with large numbers of processors and/or long run times is the increasing probability of "tachyons" in the tracefile as the number of processors increases and as individual processor clocks drift apart with time. In creating tracefiles for viewing with ParaGraph, be sure to use the highest resolution clock and sharpest clock synchronization that PICL offers. On machines with independent node clocks, PICL will try to compensate for clock skew and drift by monitoring clock behavior during a brief interval before the program begins execution. To increase the accuracy of this computation, you might try calling `clocksync0` at the end of your program, just before calling `close0` on each node. This will cause the entire duration of your run to be used in determining the appropriate adjustment for clock drift, improving its statistical significance. Tachyons may cause unpredictable behavior in ParaGraph, possibly including outright failure. Therefore, before reporting "bugs" in ParaGraph, check the tracefile for tachyons using the awk script `tachyon.awk` supplied in the ParaGraph distribution, which will print any tachyons it finds and otherwise remain silent. 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.

Another way to reduce the size of the tracefile is to refrain from tracing on the host. ParaGraph ignores all events involving the host anyway, so tracing on the host pointlessly clutters the tracefile with data that are irrelevant to the visualization. The decision to ignore the host in ParaGraph was based on a number of factors, including the difficulty of representing a host pictorially without spoiling the symmetry of many of the displays, the difficulty of obtaining accurate and reliable timestamps on a time-shared multiuser host, the fact that most parallel programs do not use the host for any substantive computations anyway, and the fact that many vendors support multiple hosts or are eliminating the need for separate hosts in their systems.

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 the simulation runs for too long. So, 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 simulation 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 simulation 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 hitting `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. These parameters are found in the `defines.h` file. The parameters most likely to require modification are as follows:

`ALL` integer destination value used to indicate global sends (default -1),

`HOST` integer identifier for the host processor (default -32768, consistent with PICL),

`MAXP` maximum number of node processors allowed (default 128, maximum possible 512).

The usual default for the maximum number of nodes allowed is set at 128 in order to conserve memory. `MAXP` can be increased up to 512 to accommodate larger systems, but this may cause sluggish performance and should be avoided unless necessary.

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 the simulation from the beginning, was a deliberate design decision based on a desire to conserve the substantial amount of 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 a simulation run (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 or 1024 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.

10 Acknowledgements

The original implementation of ParaGraph was done almost entirely by undergraduate students during research internships at Oak Ridge National Laboratory. The overall structure of the software and the conceptual designs of the individual displays were developed by Michael Heath. Most of the programming for the initial release of ParaGraph was done by Jennifer Finger (then Etheridge) while she was an undergraduate student, first at Roanoke College and later at the University of Tennessee. Heath and Finger have continued joint development of ParaGraph since he moved from ORNL to the University of Illinois and she became a regular staff member at ORNL. Two other undergraduate students also contributed to the development of ParaGraph: Loretta Auvil, then of Alderson Broaddus College, originally developed the Hypercube display, and Michelle Hribar, then of Albion College, developed the first two application-specific displays (to illustrate parallel sorting and matrix transposition) as extensions to ParaGraph. In each case these undergraduates began their work on ParaGraph without any prior knowledge of Unix, C, workstations, computer graphics, or the X Window System, and within a single term each was contributing to the relatively sophisticated software tool described in this manual. Thus, the development of ParaGraph has been an interesting educational experiment that has provided a useful tool for the performance analysis of parallel programs.

This research was supported by the Applied Mathematical Sciences Research Program, Office of Energy Research, U.S. Department of Energy under contract DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc.

References

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

[2] T. H. Dunigan. Hypercube clock synchronization. Technical Report ORNL/TM-11744, Oak Ridge National Laboratory, Oak Ridge, TN, February 1991.

[3] 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.

[4] 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.

[5] 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, Los Alamitos, CA, April 1990. IEEE Computer Society Press.

[6] 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.

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

[8] 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, Amsterdam, The Netherlands, 1993. Elsevier Science Publishers.

[9] G. Tomas and C. W. Ueberhuber. *Visualization of scientific parallel programs*. Technical University of Vienna. April 1993.

Biographies

Michael T. Heath is Professor in the Department of Computer Science and Research Scientist at the National Center for Supercomputing Applications at the University of Illinois in Urbana-Champaign. Previously he was a Senior Research Staff Member and Computer Science Group Leader in the Mathematical Sciences Section at Oak Ridge National Laboratory. He received a Ph.D. in Computer Science from Stanford University in 1978. His current research interests are in large-scale scientific computing on parallel computers, numerical linear algebra, and performance visualization. He can be contacted by email at the following address: heath@ncsa.uiuc.edu.

Jennifer E. Finger is a Technical Research Associate in the Mathematical Sciences Section at Oak Ridge National Laboratory. She received a B.S. degree from the University of Tennessee, Knoxville, in 1990, with a major in Mathematics. Her current interests are in computer graphics and visualization. She can be contacted by email at the following address: jenn@msr.epm.ornl.gov.

NAME

ParaGraph - performance visualization of parallel programs

SYNOPSIS

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

DESCRIPTION

ParaGraph is a graphical display system for visualizing the behavior and performance of parallel programs on message-passing parallel computers. It takes as input execution trace data provided by PICL (Portable Instrumented Communication Library), developed at Oak Ridge National Laboratory and available from netlib. PICL optionally produces an execution trace during an actual run of a parallel program on a message-passing machine, and the resulting trace data can then be replayed pictorially with ParaGraph to display a dynamic, graphical depiction of the behavior of the parallel program. ParaGraph provides several distinct visual perspectives from which to view processor utilization, communication traffic, and other performance data in an attempt to gain insights that might be missed by any single view.

ParaGraph is based on the X Window System and runs on a wide variety of graphical workstations. It uses no X toolkit and requires only Xlib. Although ParaGraph is 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 PICL. 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 currently provides about 25 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 PICL, which is also available from netlib. The tracing option of PICL produces a tracefile with records in node order. For graphical animation with ParaGraph, the tracefile needs to be sorted into time order, which can be accomplished with the Unix sort command:

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

When using PICL to produce tracefiles for viewing with ParaGraph, set tracelevel(4,4,0) to produce the trace events required by ParaGraph. You may also want to define tasks using the traceblockbegin and traceblockend commands of PICL 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 "host" processor graphically and ignores all trace events involving the host, so tracing on the host is not encouraged when the tracefile is to be viewed using ParaGraph.

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 environment file (default: .pgrc).
- -f (or no switch) to specify a tracefile directory path or filename.
- -g to force grayscale display mode.
- -l to specify an animation layout file (default: .pganim).
- -m to force monochrome display mode.
- -n to specify a name for the base window (default: ParaGraph).
- -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	defines an optional order or alternative names for the processors.
.pgcolors	defines the color scheme to be used for identifying tasks.
.pganim	defines an animation layout file.

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.tex	a user guide in Latex 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 machine-readable manual for ParaGraph, in Latex format, is provided 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 PICL is available from netlib and in the technical reports ORNL/TM-11130 and ORNL/TM-11616.

BUGS

Some of the displays are not repaired 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