

ornl

MARTIN MARIETTA ENERGY SYSTEMS LIBRARIES



3 4456 0354988 1

ORNL/TM-11813

**OAK RIDGE
NATIONAL
LABORATORY**

MARTIN MARIETTA

Visualizing Performance of Parallel Programs

M. T. Heath
J. A. Etheridge

OAK RIDGE NATIONAL LABORATORY

CENTRAL RESEARCH LIBRARY

CIRCULATION SECTION

4500N ROOM 175

LIBRARY LOAN COPY

DO NOT TRANSFER TO ANOTHER PERSON

If you wish someone else to see this
report, send in name with report and
the library will arrange a loan.

UCN-7969 3 9-77

MANAGED BY
MARTIN MARIETTA ENERGY SYSTEMS, INC.
FOR THE UNITED STATES
DEPARTMENT OF ENERGY

This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from the Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831; prices available from (615) 576-8401, FTS 626-8401.

Available to the public from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161.

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Engineering Physics and Mathematics Division

Mathematical Sciences Section

VISUALIZING PERFORMANCE OF PARALLEL PROGRAMS

Michael T. Heath †
Jennifer A. Etheridge ‡

† Center for Supercomputing Research and
Development
305 Talbot Laboratory
University of Illinois
104 South Wright Street
Urbana, IL 61801-2932

‡ Mathematical Sciences Section
Oak Ridge National Laboratory
P.O. Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367

Date Published; May 1991

This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
managed by
Martin Marietta Energy Systems, Inc.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-84OR21400



3 4456 0354988 1

Contents

1	Motivation and Design Philosophy	1
1.1	Graphical Simulation	1
1.2	Design Goals	2
1.2.1	Ease of understanding	3
1.2.2	Ease of use	3
1.2.3	Portability	3
1.3	Previous Work	4
1.4	Relationship to PICL	5
2	Using ParaGraph	7
3	Software Design	9
4	Displays	10
4.1	Utilization Displays	11
4.1.1	Utilization Count (Figure 1)	11
4.1.2	Gantt Chart (Figure 2)	12
4.1.3	Utilization Summary (Figure 3)	12
4.1.4	Utilization Meter (Figure 4)	16
4.1.5	Concurrency Profile (Figure 5)	16
4.1.6	Kiviat Diagram (Figure 6)	16
4.2	Communication Displays	20
4.2.1	Communication Traffic (Figure 7)	20
4.2.2	Spacetime Diagram (Figure 8)	22
4.2.3	Message Queues (Figure 9)	22
4.2.4	Communication Matrix (Figure 10)	25
4.2.5	Communication Meter (Figure 4)	27
4.2.6	Animation (Figure 11)	27
4.2.7	Hypercube (Figures 12 and 13)	29
4.2.8	Node Statistics (Figure 14)	29
4.3	Task Displays	33
4.3.1	Task Count (Figure 15)	34
4.3.2	Task Gantt (Figure 16)	34
4.3.3	Task Status (Figure 17)	37
4.3.4	Task Summary (Figure 18)	37
4.4	Other Displays	37
4.4.1	Phase Portrait (Figure 19)	37
4.4.2	Critical Path (Figure 20)	41
4.4.3	Processor Status (Figure 21)	41
4.4.4	Clock	44
4.4.5	Trace	44
4.4.6	Statistical Summary	45
4.5	Application-Specific Displays	45

5	Options	46
6	Future Work	48
7	Acknowledgements	49
8	References	50

VISUALIZING PERFORMANCE OF PARALLEL PROGRAMS

Michael T. Heath

Jennifer A. Etheridge

Abstract

In this paper we describe a graphical display system for visualizing the behavior and performance of parallel programs on message-passing multiprocessor 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. Several distinct 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 illustrate its usefulness in analyzing parallel programs.

1. Motivation and Design Philosophy

Graphical visualization is a standard technique for facilitating human comprehension of complex phenomena and large volumes of data (see, for example, [12,18]). 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 paper is to describe this visualization tool, outline the motivation and philosophy behind its design, and illustrate its usefulness in analyzing 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.

Algorithm 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 the 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 more than twenty 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 [20,21]), which enables the user to produce such trace data automatically.

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 the MIT distributions of X11R2, X11R3, and X11R4, as well as several vendor-supplied versions of X Windows. Although ParaGraph is most effective in color, it also works on monochrome and gray-scale monitors, and it automatically detects which type of monitor is in use. 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, Ncube, Symult). 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.

1.3. 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 [6,7,8,9,33,56]. Visualization of parallel computations has been the subject of a number of recent Ph.D. theses [11,34,48], technical articles [2,27,31,32,36,38,42,44,47,49,50,55,57], and even a book [53]. Graphical visualization has also been an important component of several environments that have been developed for parallel programming [1,5,16,22,46,54], debugging [25,26,37,60], and monitoring [23,29,39,40], as well as integrated environments that combine several of these components [17,35,52]. Algorithm visualization tools have also been developed for specific applications, such as matrix computations [3,4,13,43,58]. ParaGraph is a general-purpose performance visualization tool that is distinguished from previous efforts in the following ways:

- The sheer multiplicity of displays provided by ParaGraph is unique. Other packages have emphasized the importance of multiple views (e.g., [11,31,36,46]), 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 new standards in the intuitive appeal and aesthetic quality of the displays provided by ParaGraph, including both the new displays we have devised and the displays 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.
- 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 have obtained an early version from Netlib [14] over the Internet during the past year, and have been able to build the program at their locations and use it effectively without the benefit of any documentation beyond a one-page README file.

1.4. Relationship to PICL

PICL is a Portable Instrumented Communication Library [20,21] that runs on a variety of message-passing parallel architectures. As its name implies, it 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.

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.

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 [15]. On some machines, PICL actually provides a higher resolution clock than the one supplied by the system vendor.

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 primary additional cost is due to the clock calls necessary to determine the timestamps for the event records to be placed in the tracefile [20]. These clock calls, 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. Using ParaGraph

ParaGraph supports command line options that specify a hostname for remote display across a network, forced monochrome display mode (useful if black-and-white hardcopies are to be made from a color screen), or a tracefile name. The tracefile can also be specified (or changed) during execution by typing the filename in the appropriate entry of the options menu. ParaGraph preprocesses the input 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.

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 additional menu of miscellaneous displays, and a menu for specifying various options and parameters. 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.

After selecting the desired displays, the user presses `start` to begin the graphical

simulation of the parallel program based on the tracefile specified. The animation then proceeds straight through to the end of the tracefile, but it can be interrupted for detailed study by use of the **pause/resume** button. For even more detailed study, the **step** button provides a single-step mode that processes the tracefile one event 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 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. 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. The speed is also affected by the complexity of the displays and the type and amount of scrolling used. In its initial design, when there were only a few displays available, we included parameterized delay loops to slow the drawing down in case it

moved too quickly for the human eye to follow. However, as we added more displays, this ceased to be a problem and we dispensed with the delay loops, opting instead for the more indirect control over simulation speed mentioned above. We find that now users tend to complain more that the simulation is too slow rather than too fast, since most like to have many displays open at once. Moreover, one can always resort to single-step mode if arbitrarily slow drawing speed is desired for very close study of program behavior.

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 (e.g., initial selection of displays, options, and parameter values) and dynamically (e.g., pause/resume, single-step mode).

ParaGraph is written in C, and the source code contains about 10,000 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.

4. Displays

In this section we describe and illustrate the individual displays provided by ParaGraph. Some of these 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. In illustrating these displays in a printed manuscript, we obviously cannot convey the dynamic movement portrayed by ParaGraph in actual practice, but must content ourselves with snapshots taken during a typical execution. The figures were produced from tracefiles made on an Intel iPSC/2 hypercube.

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 and still-picture illustrations of the displays. For clarity and simplicity, the illustrative examples use only a small number of processors. Many of the displays scale up well to much larger numbers of processors, but a few contain too much detail to scale up well. We will discuss later the number of processors that can be supported effectively and the limitations we see in our approach.

The parallel program illustrated in most of the figures is a common computation in scientific computing, the solution of a large sparse system of linear equations by Cholesky factorization. For details of the parallel algorithm used, see [24]. In the

example, the sparse matrix of the linear system arises from a 15×15 square grid, so that the matrix is of order 225. The nodes of the grid, and hence the rows and columns of the matrix, are ordered by nested dissection, which is a type of domain decomposition that leads to a typical divide-and-conquer parallel algorithm for the factorization. In the example, each of the eight processors initially computes the portion of the factorization corresponding to the interior of its own part of the grid, and can do so independently of the other processors. Eventually, however, the processors reach a point where interprocessor communication is required to supply boundary data from neighboring portions of the grid that are needed before computations can proceed any further. The processors team up in four pairs, then two sets of four, and finally all eight together, as they work their way up the elimination tree and communicate across higher level boundaries.

4.1. 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.

4.1.1. Utilization Count (Figure 1)

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. In the example shown in Figure 1, the all-green portion at the far left depicts the final part of the perfectly parallel phase with which the divide-and-conquer algorithm begins.

4.1.2. Gantt Chart (Figure 2)

This display, which is patterned after graphical charts used in industrial management [19], 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.

4.1.3. Utilization Summary (Figure 3)

Unlike the displays described previously, which show current behavior and change dynamically with time, the Utilization Summary display is defined only at the end of a run. It shows the 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. In the sparse matrix example shown

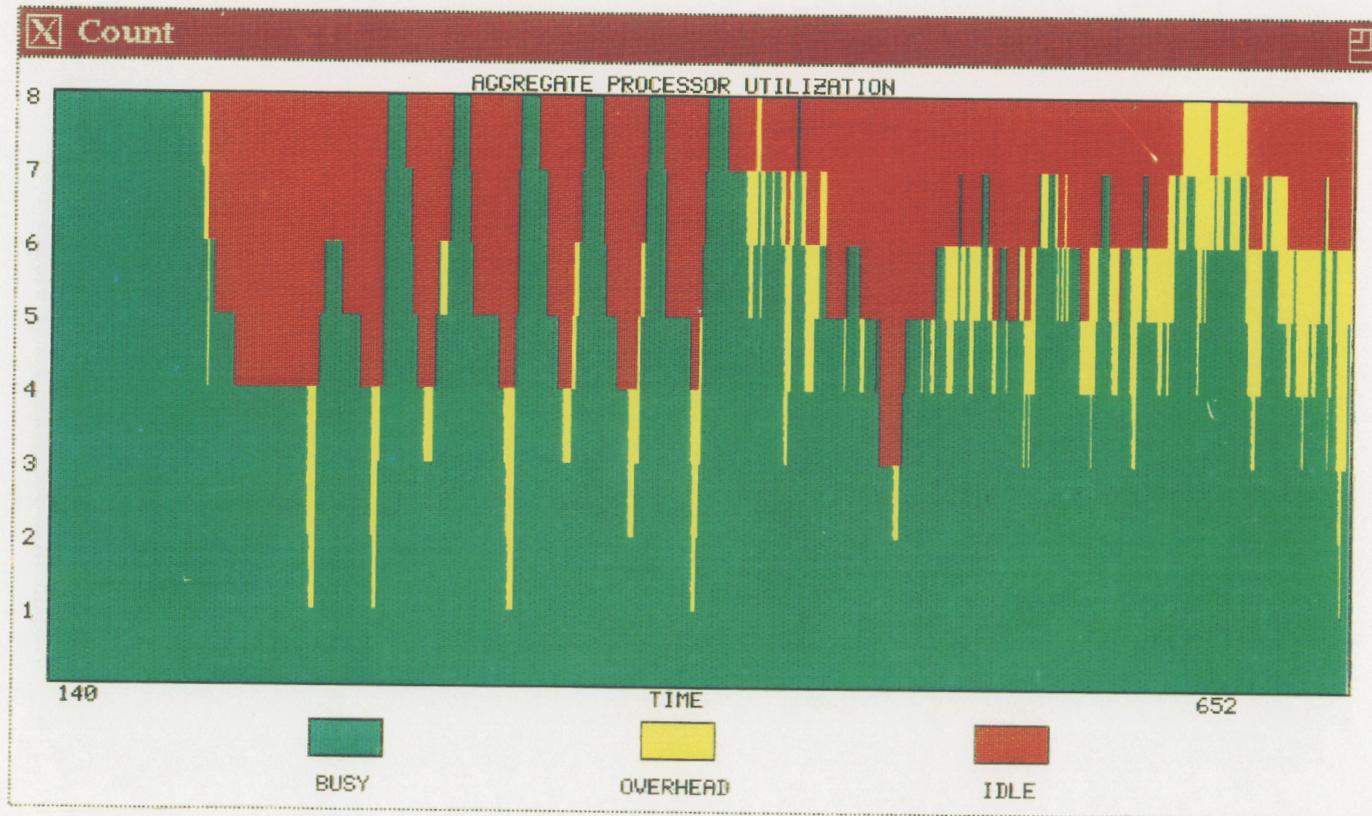


Figure 1. Utilization Count display.

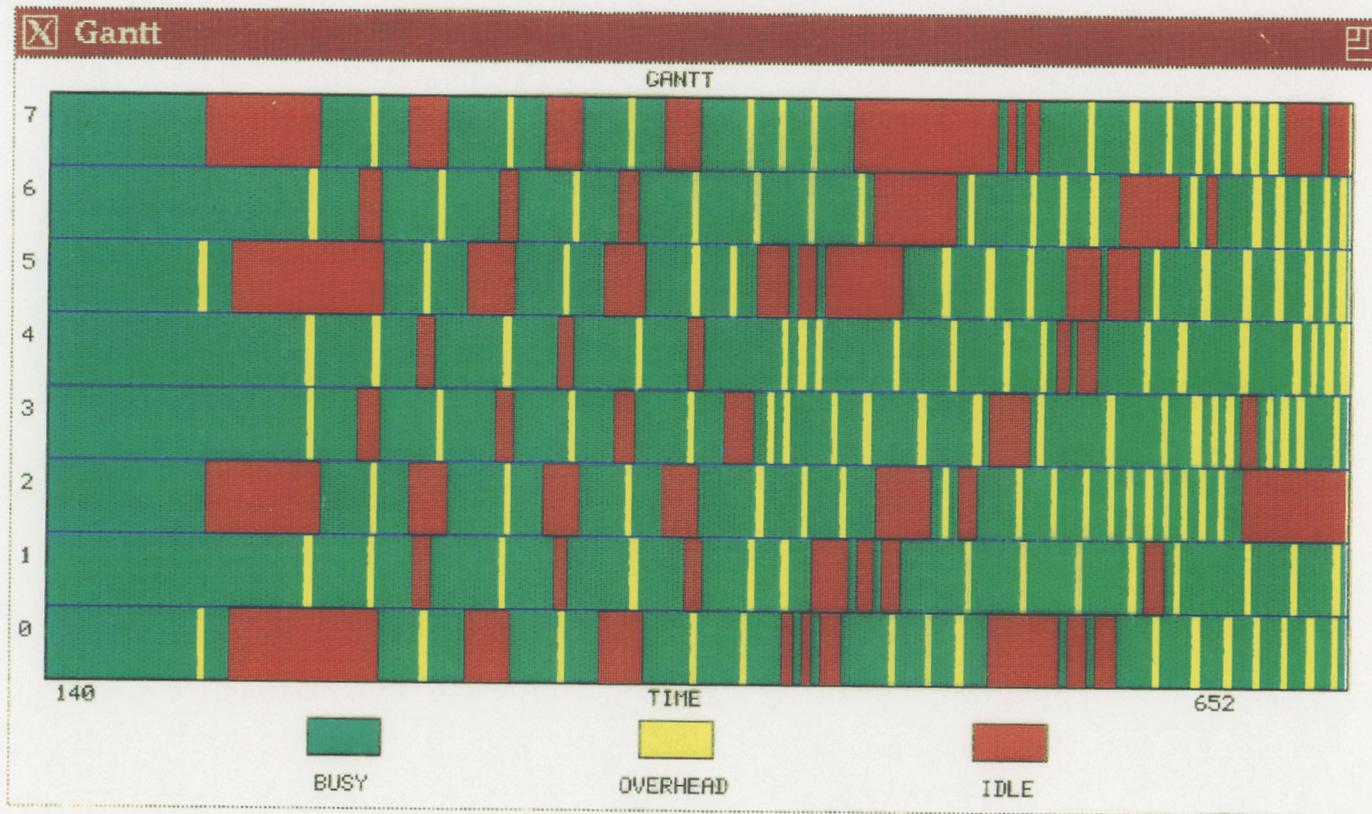


Figure 2. Gantt Chart display.

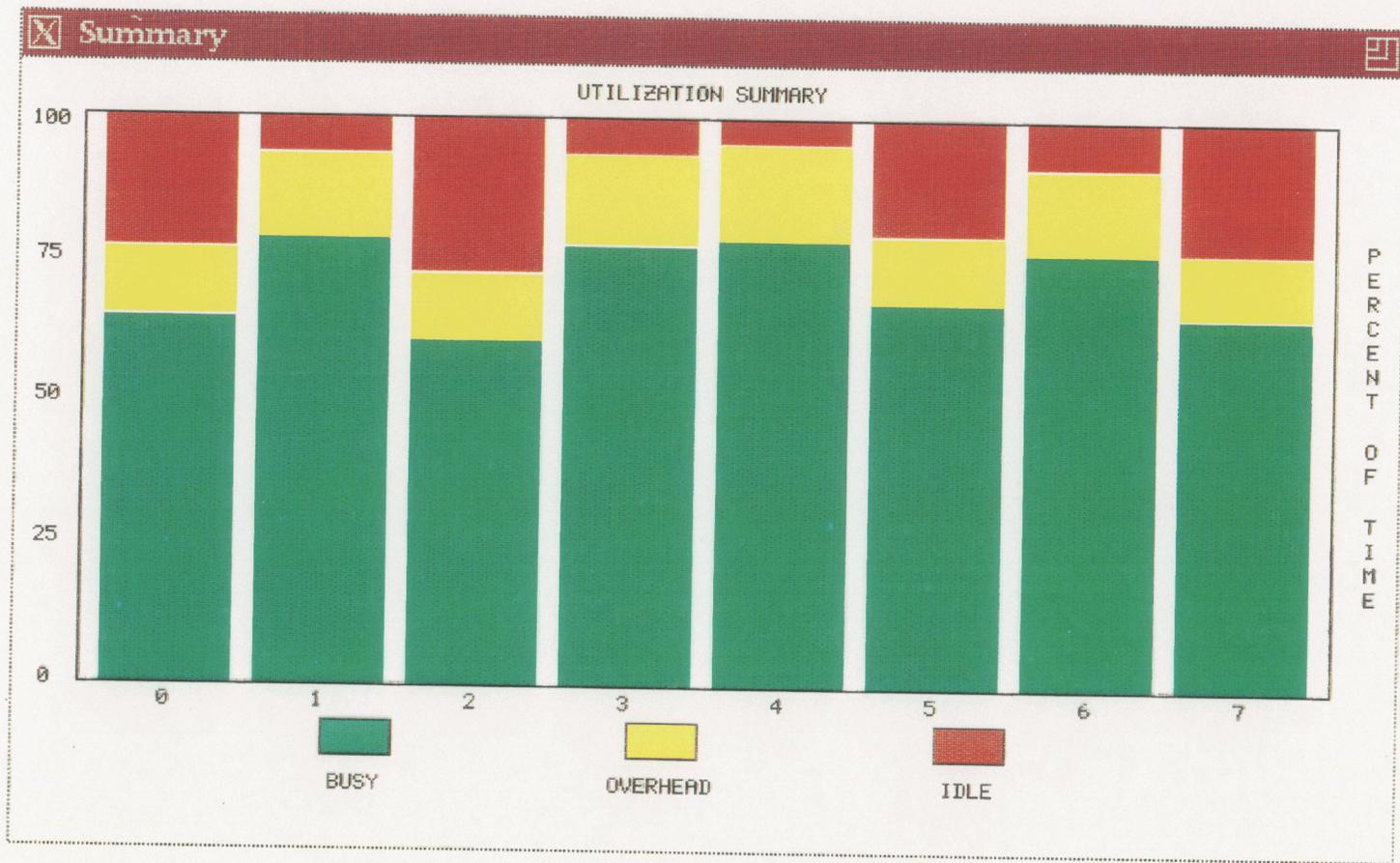


Figure 3. Utilization Summary display.

in Figure 3, four of the processors are assigned the four corners of the grid, while the other four are assigned central portions of the grid, leading to a load imbalance that is clearly visible.

4.1.4. Utilization Meter (Figure 4)

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.

4.1.5. Concurrency Profile (Figure 5)

This is another summary display that becomes defined only at the end of a run. 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. 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). Figure 5 shows the busy and idle profiles for the sparse matrix example; the overhead profile is not shown.

4.1.6. Kiviat Diagram (Figure 6)

This display, which is adapted from related graphs used in other types of performance evaluation [28,41], 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

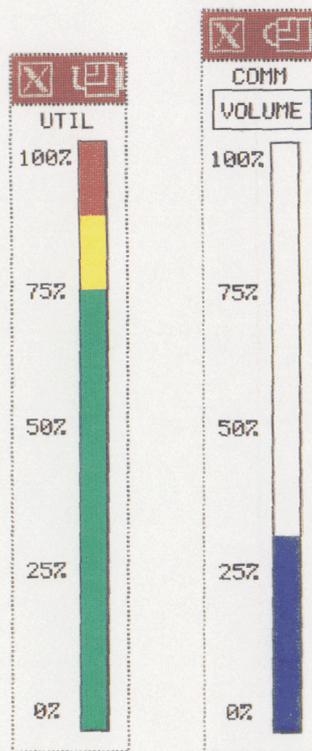


Figure 4. Utilization Meter (left) and Communication Meter (right) displays.

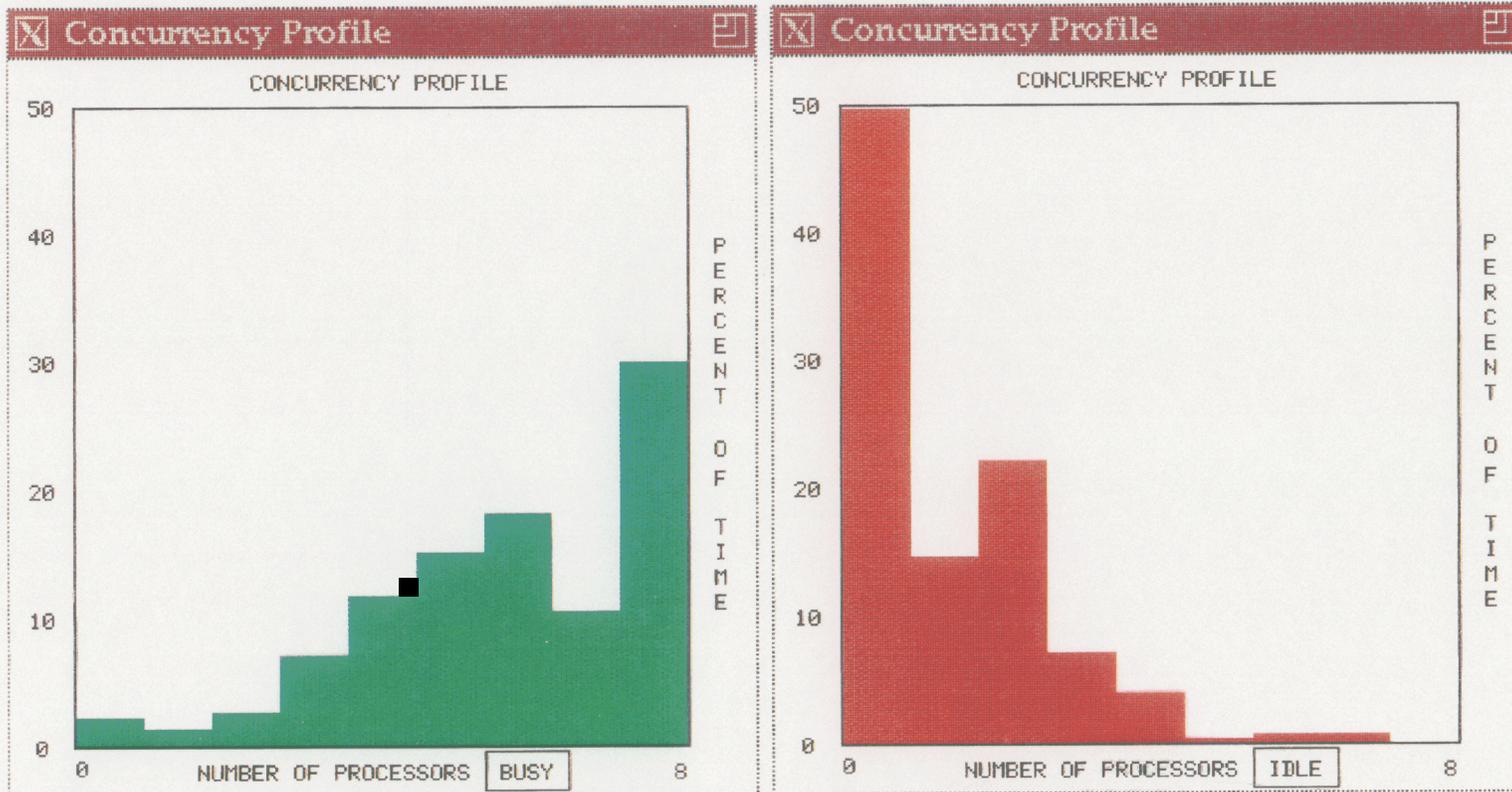


Figure 5. Concurrency Profile display.

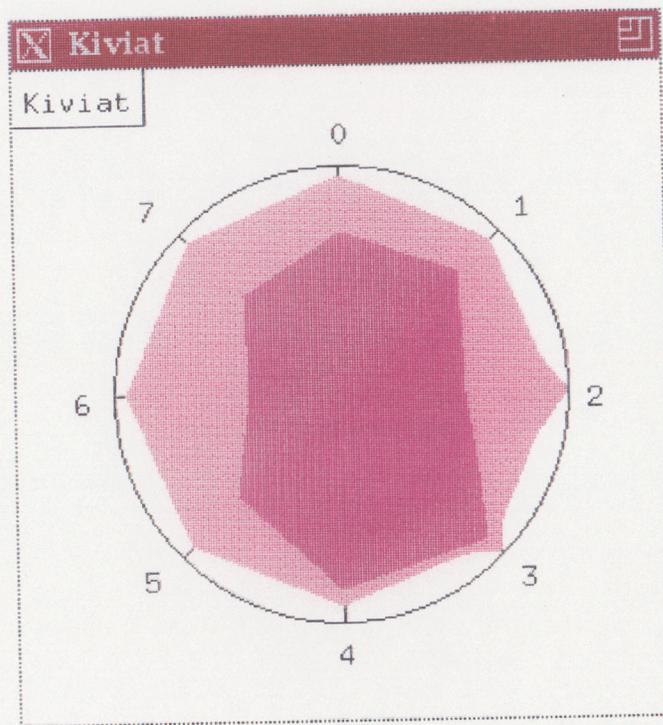


Figure 6. Kiviat Diagram display.

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. The current utilization is shown in dark shading, while the "high water mark" seen thus far is shown in lighter shading. The "current" utilization 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.

4.2. Communication Displays

The displays described in this section are concerned primarily with depicting inter-processor communication. They are helpful in determining the frequency, volume, and overall pattern of communication, and whether there is congestion in the message queues.

4.2.1. Communication Traffic (Figure 7)

This display is a simple plot of the total communication traffic in the interconnection network (including 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. Figure 7 shows the successively higher peaks in communication traffic for the sparse matrix example as higher level grid separators are encountered.

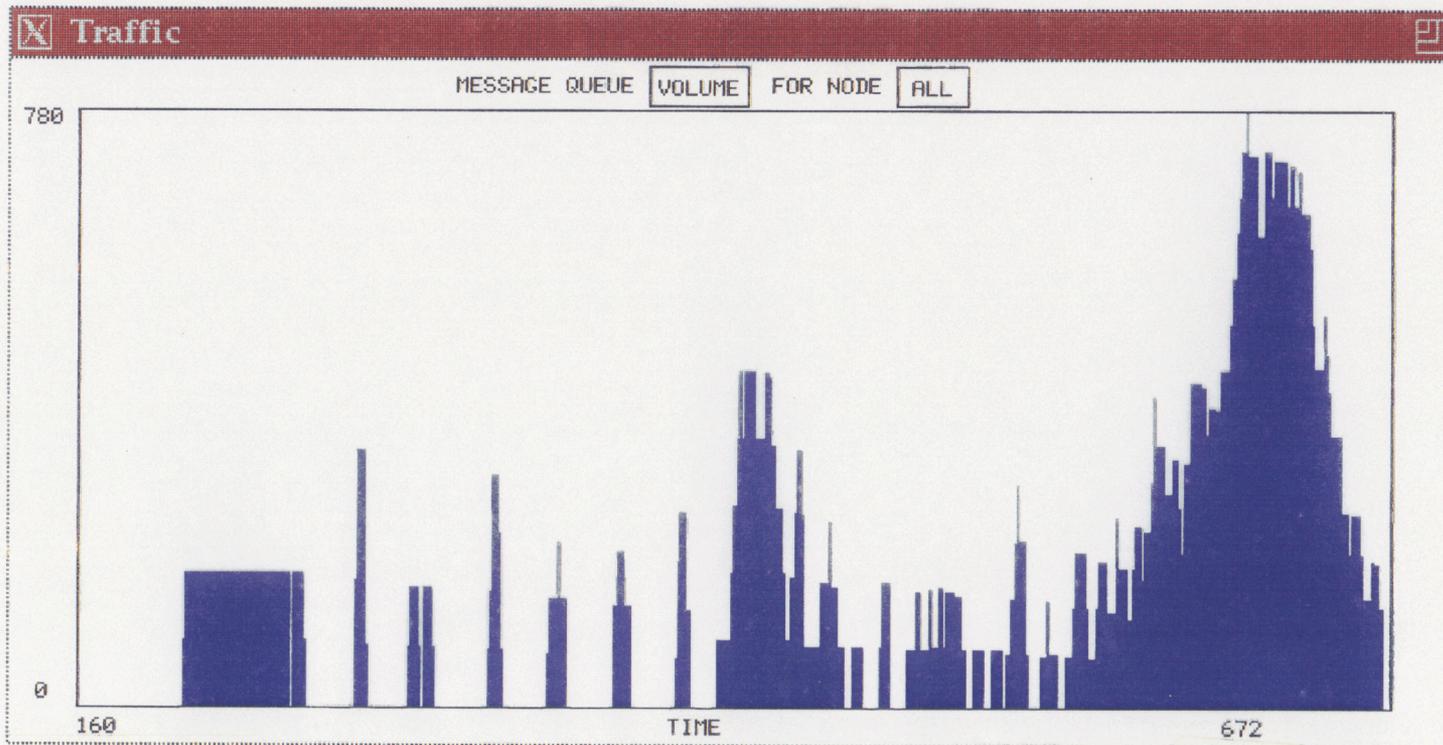


Figure 7. Communication Traffic display.

4.2.2. Spacetime Diagram (Figure 8)

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 [30] 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 [59, page 31]. In our adaptation of the Spacetime Diagram, processor number is on the vertical axis, and time is on the horizontal axis, which scrolls as necessary as time proceeds. Processor activity (busy/idle) is indicated by horizontal lines, one for each processor, with the line drawn solid if the corresponding processor is busy, 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 to indicate the sizes of the messages being transmitted. 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. The divide-and-conquer nature of the sparse matrix example can be clearly seen in Figure 8. The eight processors initially work independently, then combine in successively larger groups as they move up the elimination tree.

4.2.3. Message Queues (Figure 9)

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

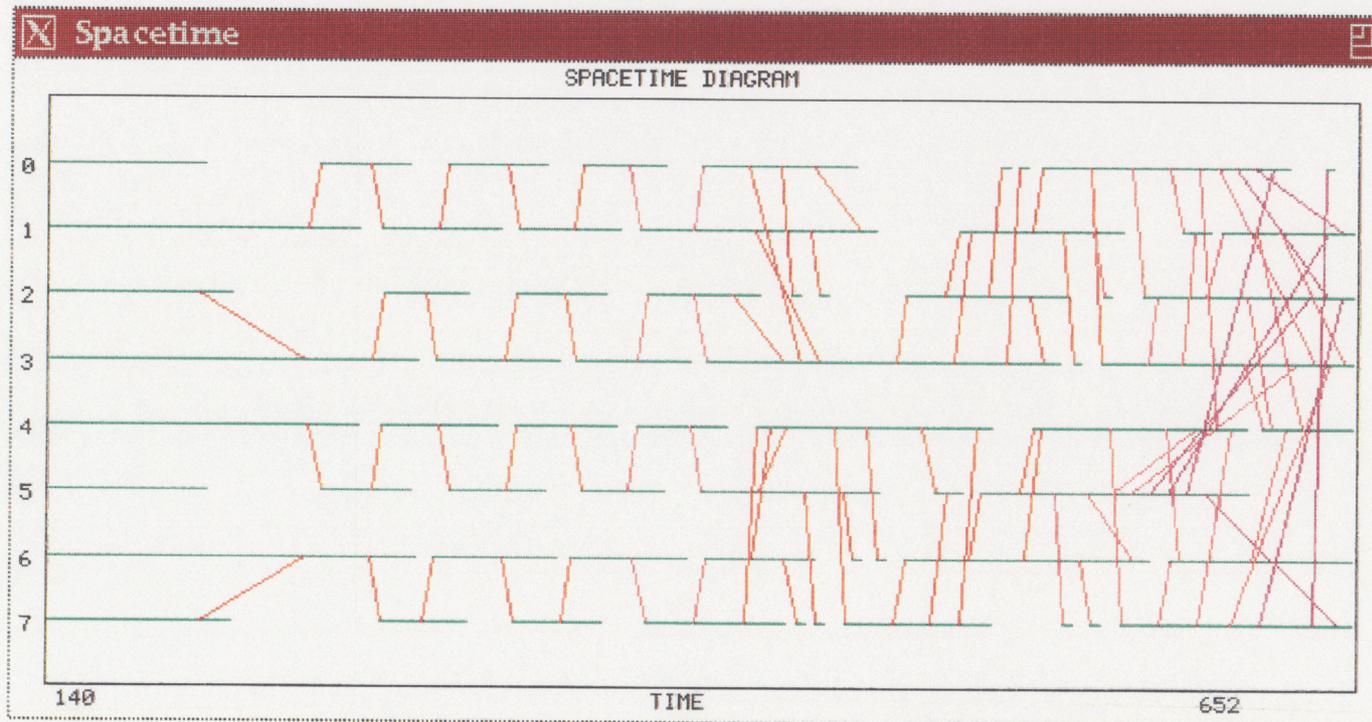


Figure 8. Spacetime Diagram display.

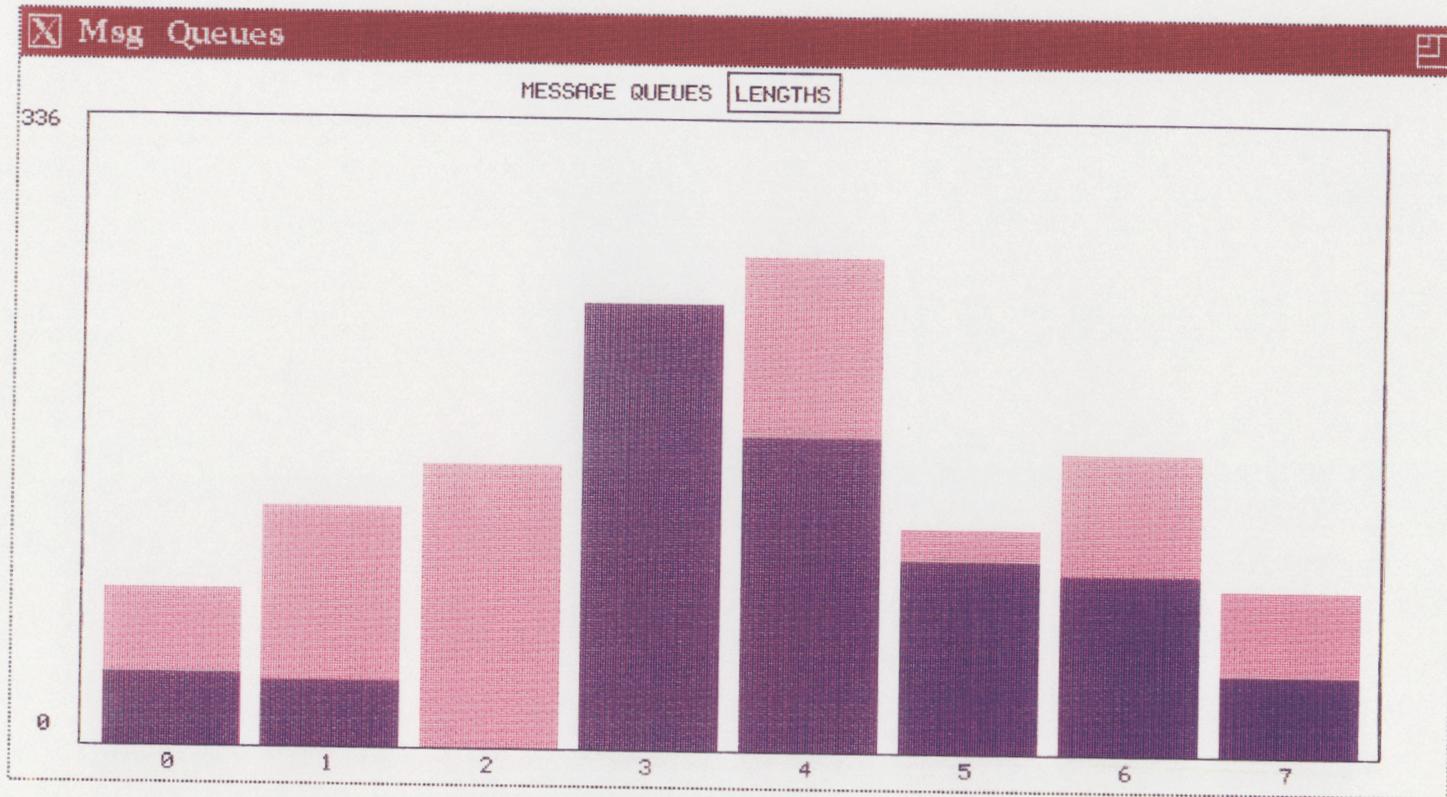


Figure 9. Message Queues display.

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. In the example shown in Figure 9, processor 2 currently has no messages in its input queue; the remaining processors all have messages awaiting receipt by their user processes, but only the queue on processor 3 is at its maximum size seen so far.

4.2.4. Communication Matrix (Figure 10)

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 indicates the size of the message in bytes, as given in the separate Color Code display that can also be selected from the menu. Thus, the sizes, durations, and overall pattern of messages are depicted by this display. The nodes can be ordered along the axes in either natural or Gray code order, and the user's choice may strongly affect the appearance of the communication pattern. At the end of the simulation, the Communication Matrix display shows the cumulative

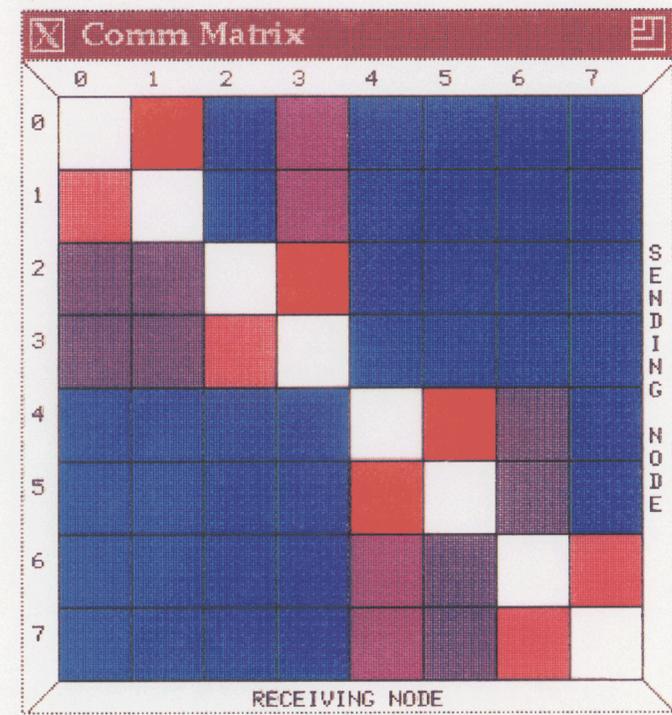
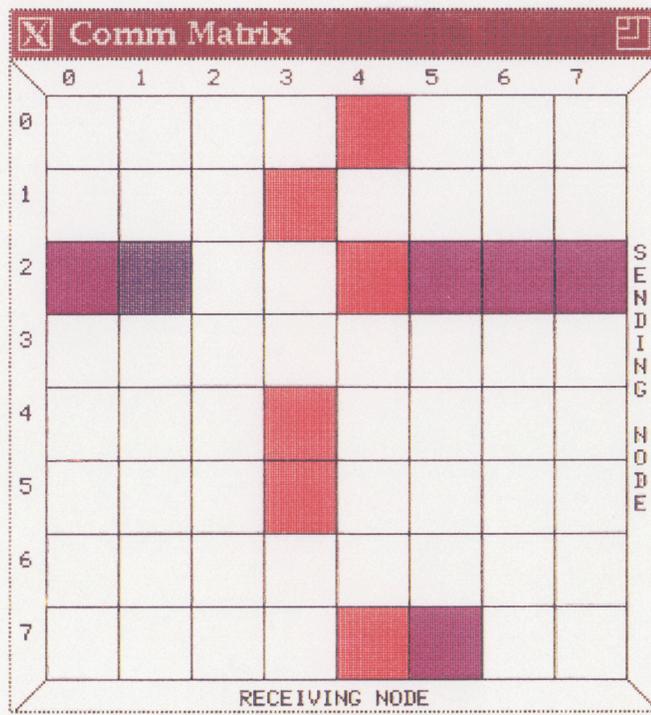


Figure 10. Communication Matrix display during run (left) and concluding summary (right).

communication volume for the entire run between each pair of processors.

4.2.5. Communication Meter (Figure 4)

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.

4.2.6. Animation (Figure 11)

In this display, the multiprocessor is represented by a graph whose nodes (depicted by numbered circles) represent processors, and whose arcs (depicted by lines between the circles) represent communication links. 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 multiprocessor. An arc is drawn between the source and destination processors when a 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 small circles depicting the processors are arranged in a large circle merely for convenience in drawing straight lines between arbitrary pairs of processors without intersecting any other processors; this is not meant to suggest that the underlying architecture is necessarily a ring. The nodes can be ordered around the circle in either natural or Gray code order, and the user's choice may strongly affect the appearance of the communication pattern among processors. The arcs represent the logical, rather than physical, connectivity of the multiprocessor network, and possible routing of messages through intervening nodes is not depicted unless the program being visualized does such forwarding explicitly. In the example shown in Figure 11, a total of four messages are pending receipt. Note that various combinations of states are possible for the sending and receiving processors. For example, both processors 2 and 3 are busy, one having already sent the message and resumed computing, while the other has not yet stopped computing to receive it. Upon conclusion, this display shows

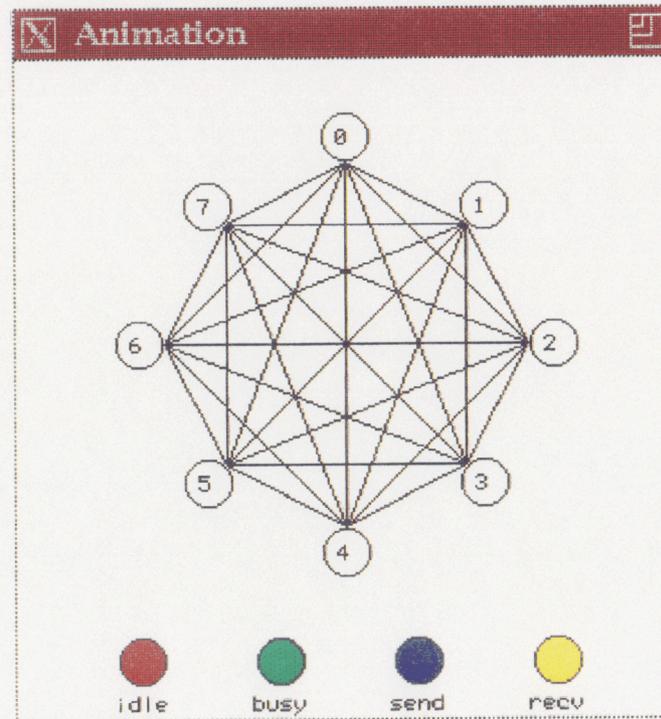
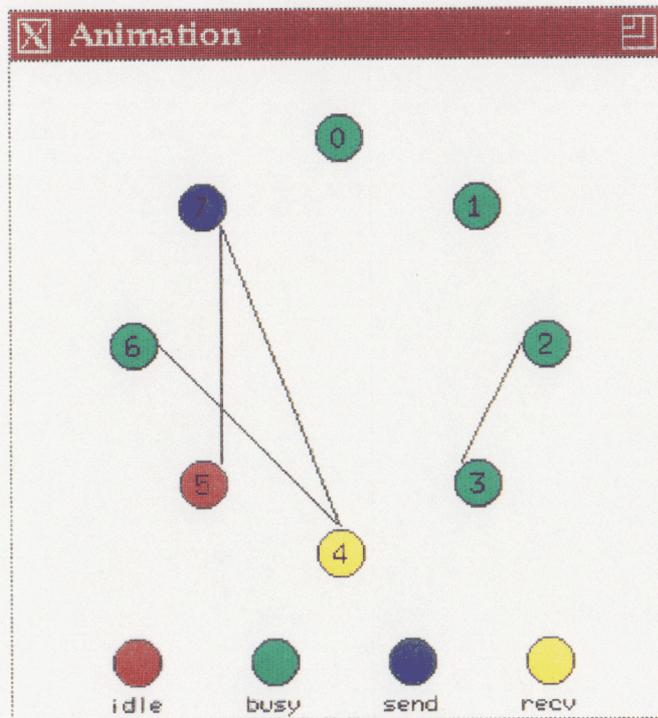


Figure 11. Animation display during run (left) and concluding summary (right).

a summary of all (logical) communication links used throughout the run.

4.2.7. Hypercube (Figures 12 and 13)

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 [10,51]. The layouts provided include ring, ring of rings, web, cube, lateral cubes, nested cubes, mesh, linear, tree, tesseract, and polytope arrangements, some of which are illustrated in Figure 12. 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 intermediate 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 intervening processors. In Figure 13, for example, the message between nodes 0 and 5 must travel over a virtual link by being forwarded through an intermediate processor, whereas the message between nodes 0 and 2 travels directly over the physical link between those two processors. 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.

4.2.8. Node Statistics (Figure 14)

This display provides, in graphical form, detailed communication statistics for a single, user-selected processor. The choices of statistics plotted are the source/destination, type, length, and Hamming distance traveled for all messages sent to or from the chosen processor. Time is on the horizontal axis, and the chosen statistic is on the vertical axis, with incoming and outgoing messages shown in separate windows. This display is helpful in analyzing communication behavior in detail, especially in perceiving trends

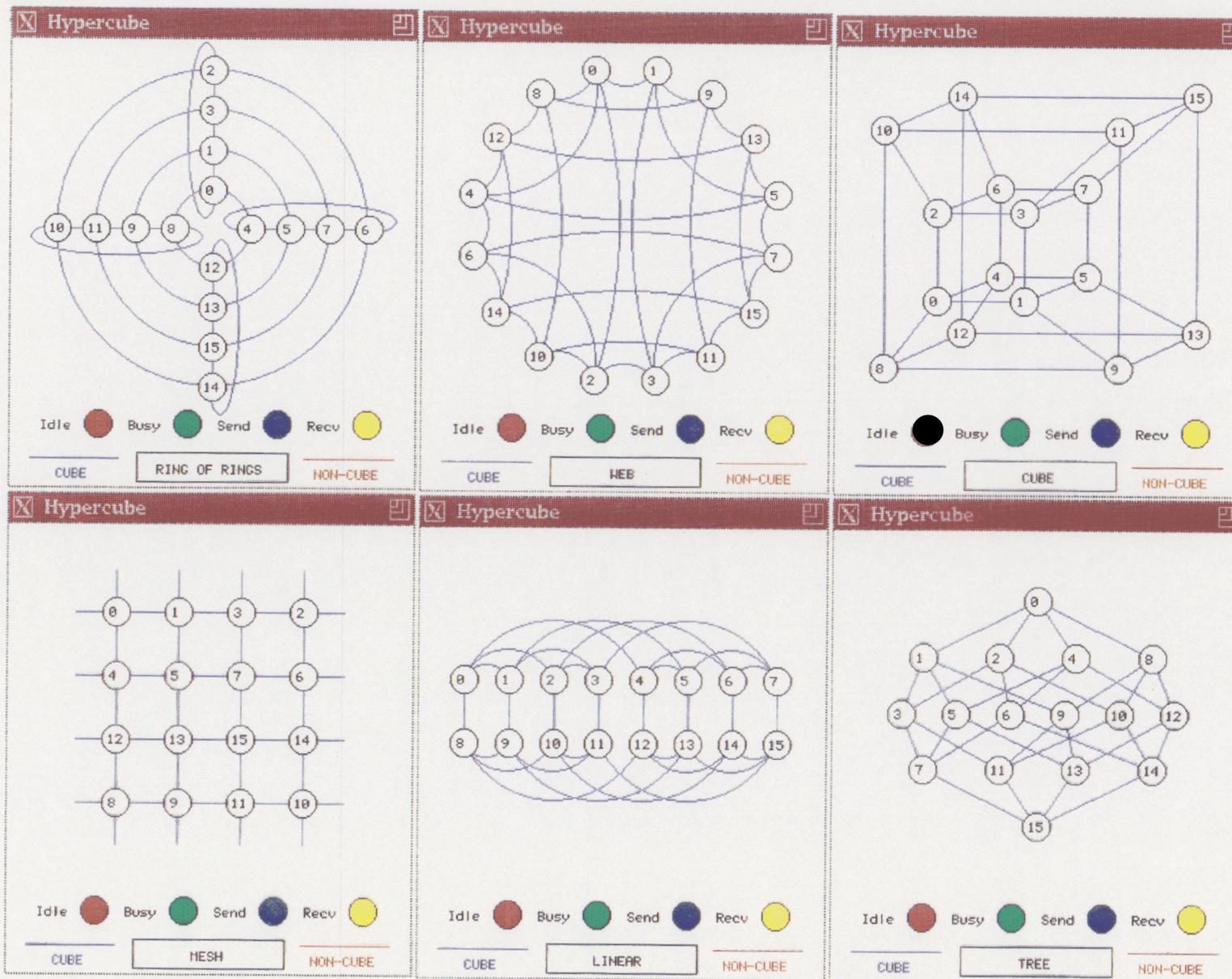


Figure 12. Some of the node layouts available in the Hypercube display.

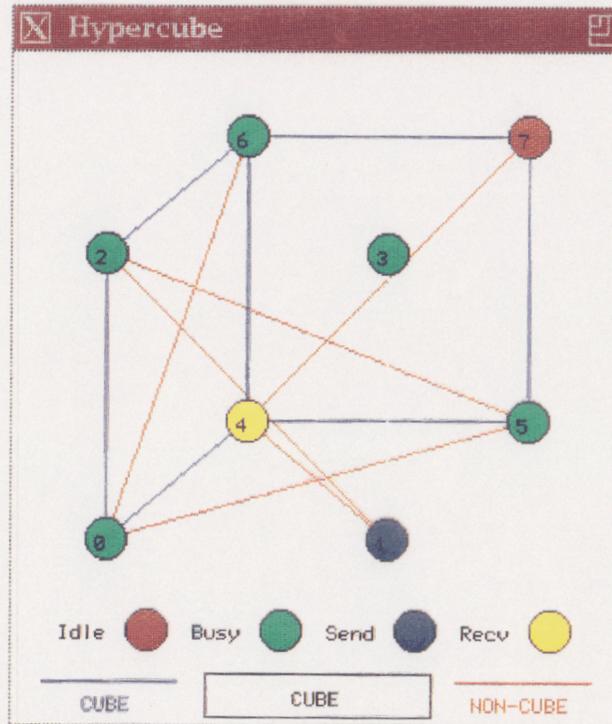


Figure 13. Hypercube display during run using "cube" layout of nodes.

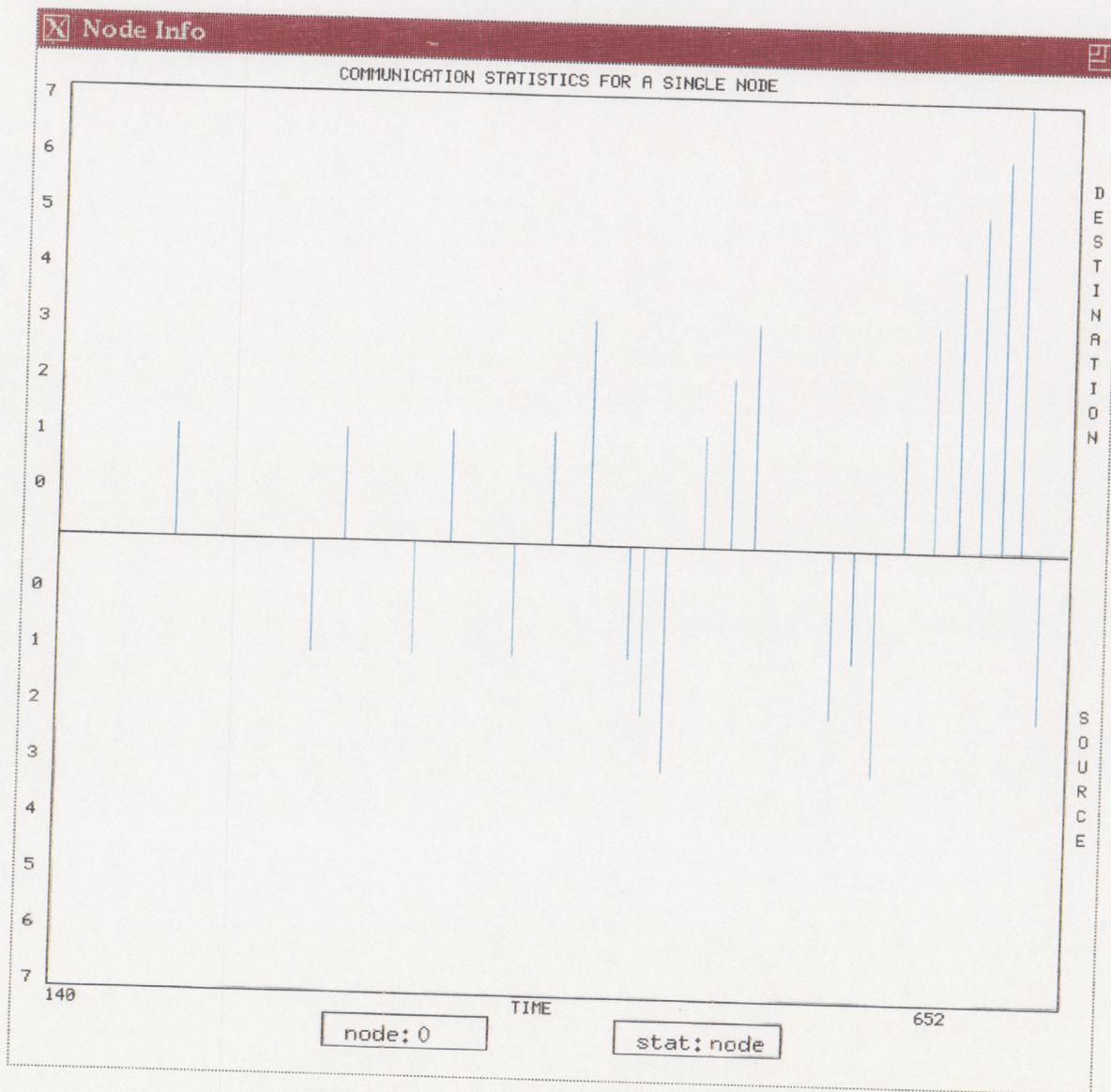


Figure 14. Node Statistics display.

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 [45].

4.3. 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 new “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 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 should 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.

Note that tasks can be nested, one inside another, but if so these should be properly bracketed by matching task begin and end records. Note also that 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 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 is likely to 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 one of six basic colors for indicating each task, with the choice of color given by the task number modulo six. In the sparse matrix example, we defined the computation of each column of the factorization to be a separate task, with the column number as task number, for a total of 225 tasks.

4.3.1. Task Count (Figure 15)

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). In the example shown in Figure 15, four processors are currently working on task 4, two are working on task 3, and one processor each on tasks 1 and 2.

4.3.2. Task Gantt (Figure 16)

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

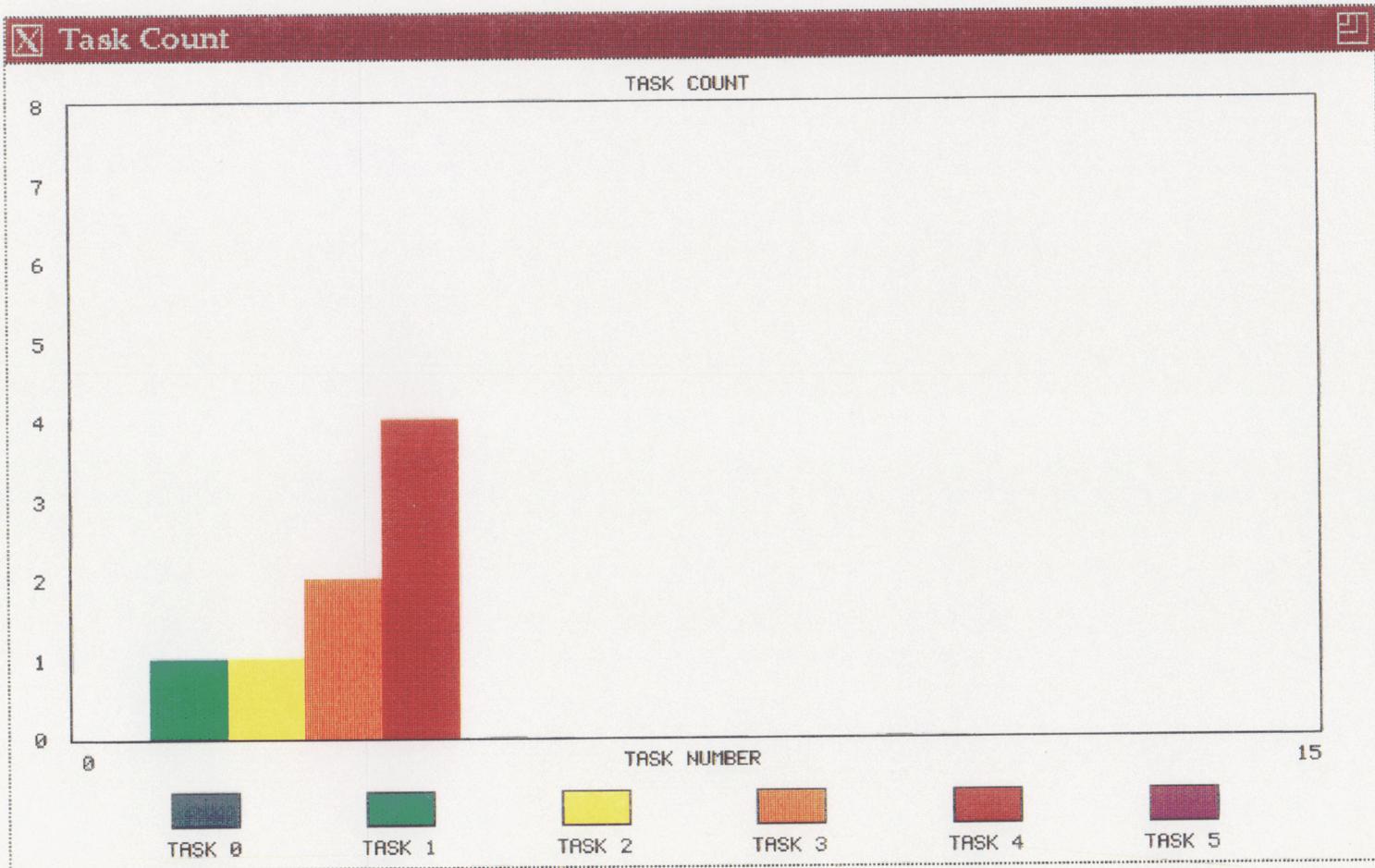


Figure 15. Task Count display.

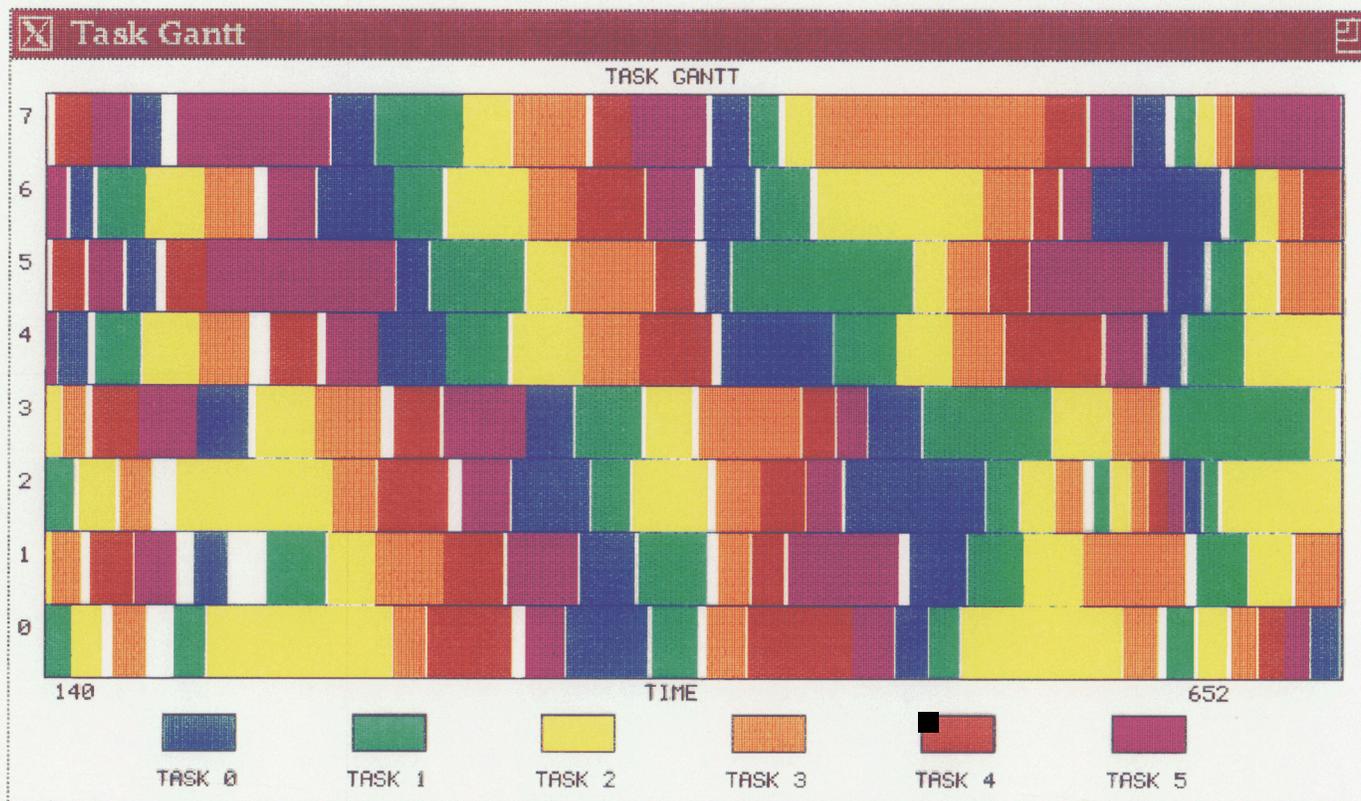


Figure 16. Task Gantt display.

busy/overhead/idle status with the task information. For instance, comparing Figure 16 with Figure 2 shows that for the sparse matrix example the longer tasks tend to be caused by extended idle periods within the task while the processor awaits needed data, rather than by a heavier work load for that processor.

4.3.3. Task Status (Figure 17)

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 that task is now in progress. When a task is subsequently completed, its corresponding square is then darkly shaded. Again, the divide-and-conquer nature of the sparse matrix example is clearly visible in Figure 17, where several factor columns associated with the interiors of the initial eight pieces of the grid have been completed at the instant shown, and precisely eight distinct tasks are currently in progress.

4.3.4. Task Summary (Figure 18)

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. Figure 18 provides another striking depiction of the divide-and-conquer sparse matrix example, with the 8-4-2-1 sequence clearly visible.

4.4. 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.

4.4.1. Phase Portrait (Figure 19)

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

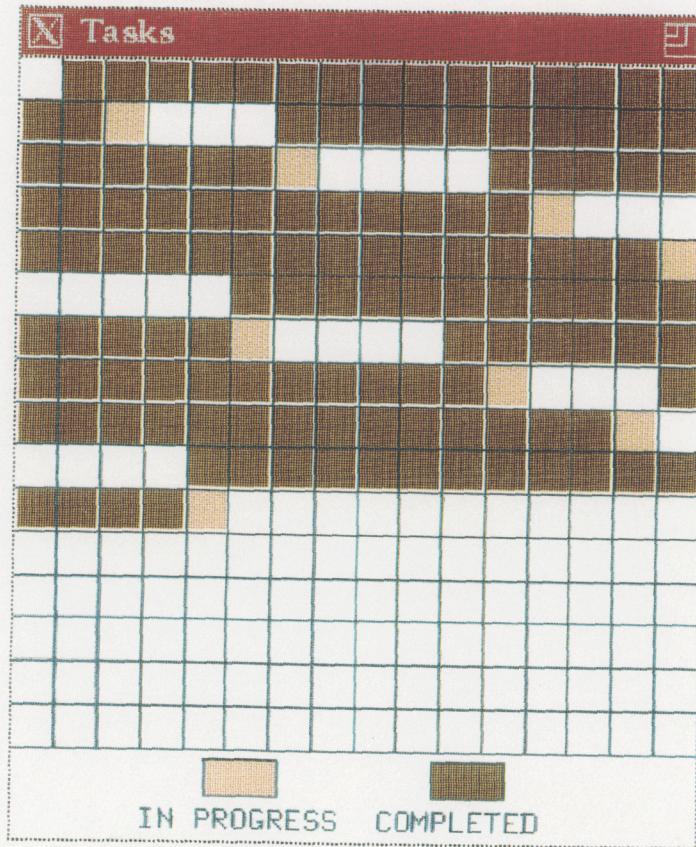


Figure 17. Task Status display.

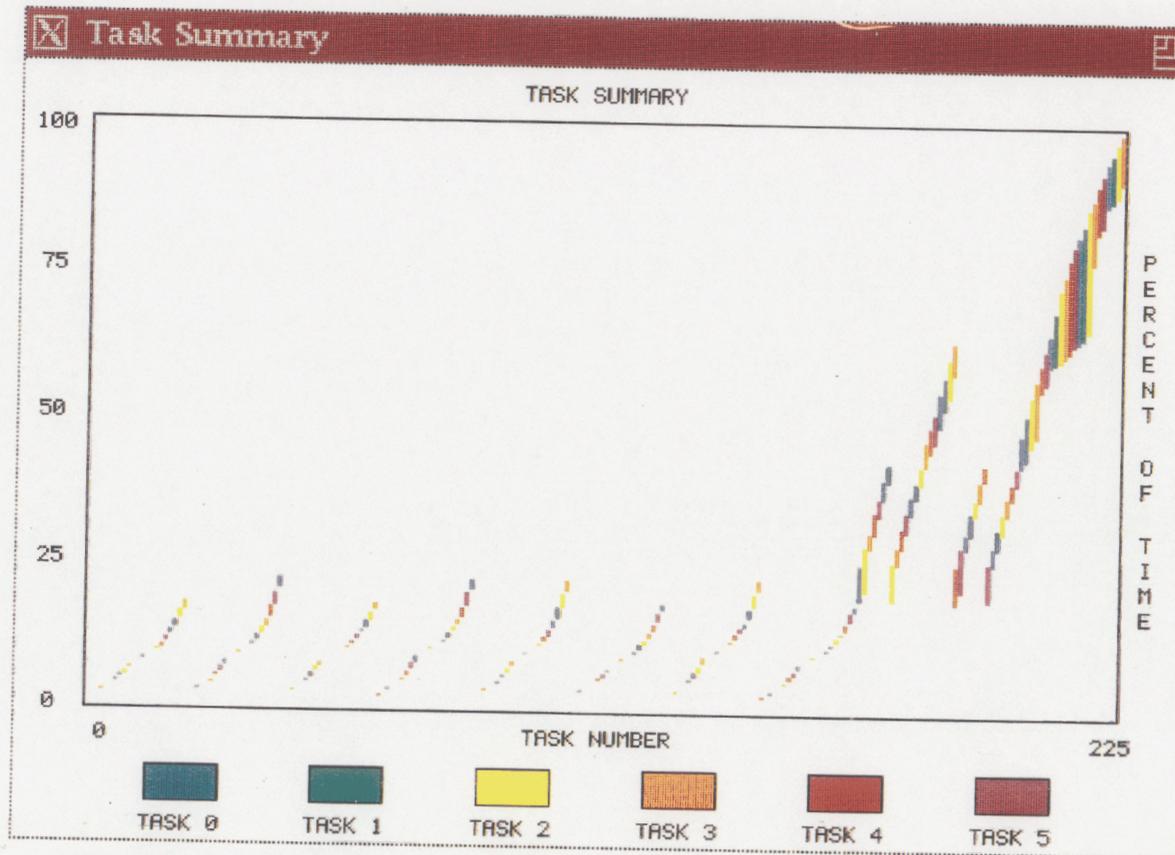


Figure 18. Task Summary display.

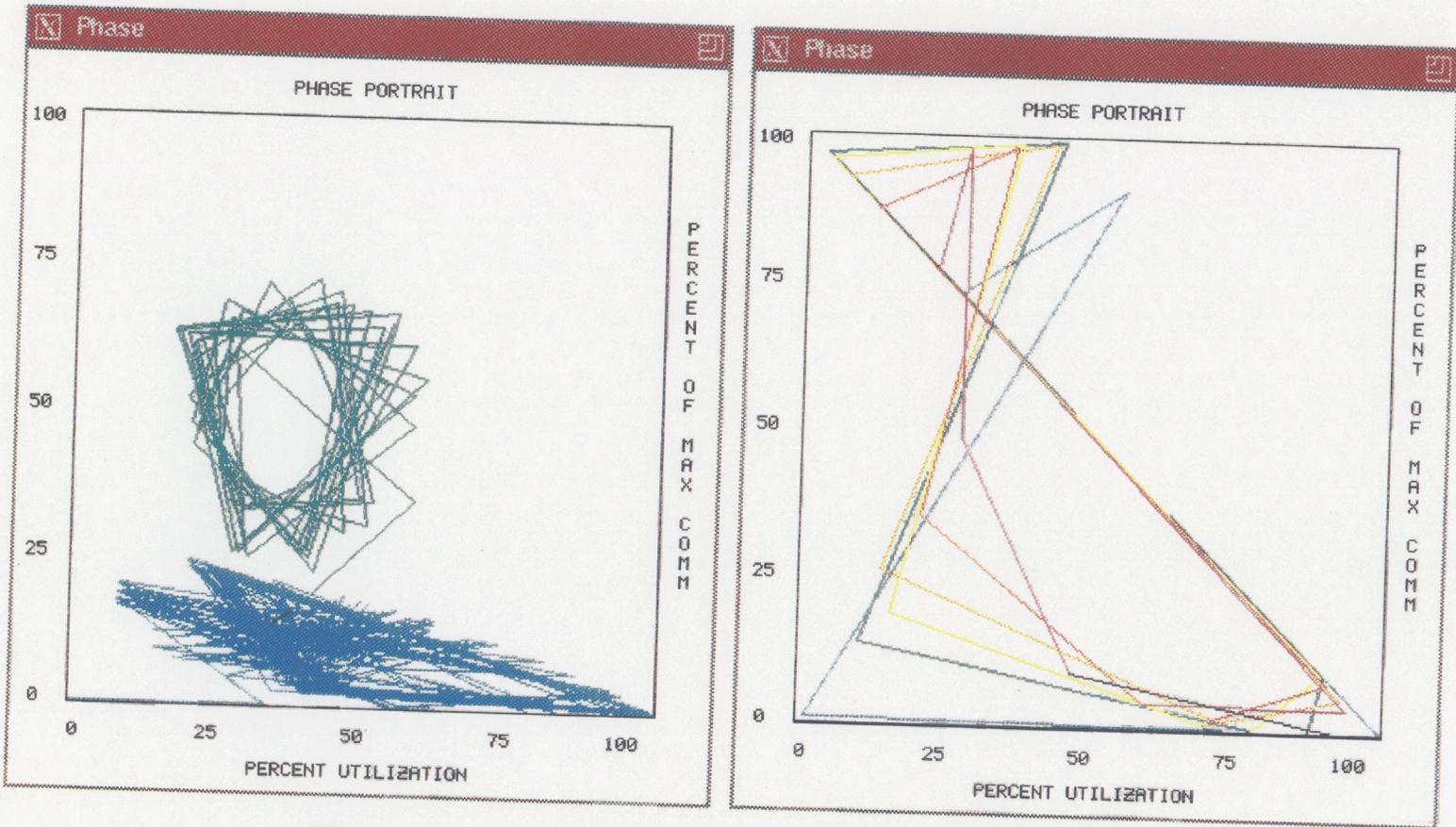


Figure 19. Two examples of the Phase Portrait display.

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. 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. In the example shown on the left in Figure 19, two distinct phases of the computation can be seen, each of which exhibits a high degree of periodic behavior. By setting task numbers appropriately, the user can color code the trajectory to highlight either major phases (Figure 19, left) or individual orbits (Figure 19, right).

4.4.2. Critical Path (Figure 20)

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.

4.4.3. Processor Status (Figure 21)

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

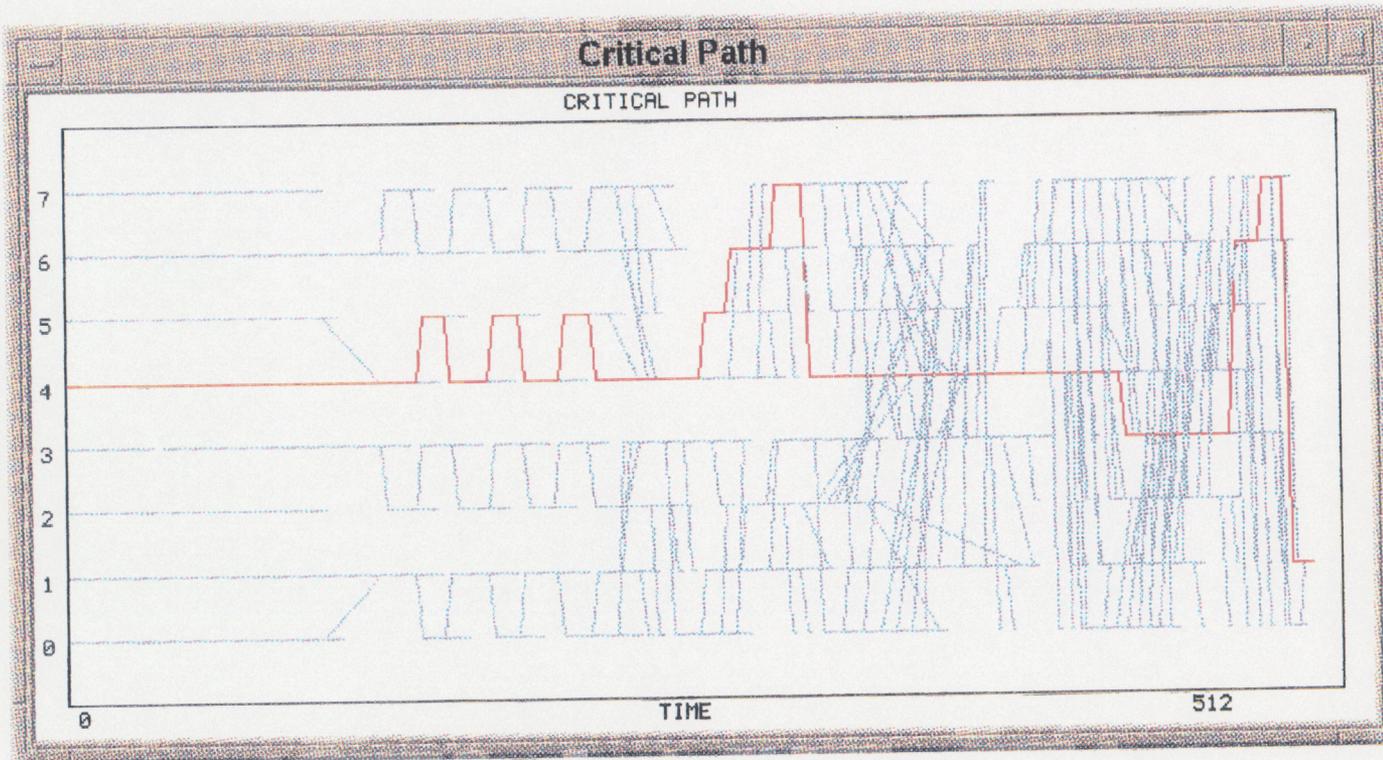


Figure 20. Critical Path display

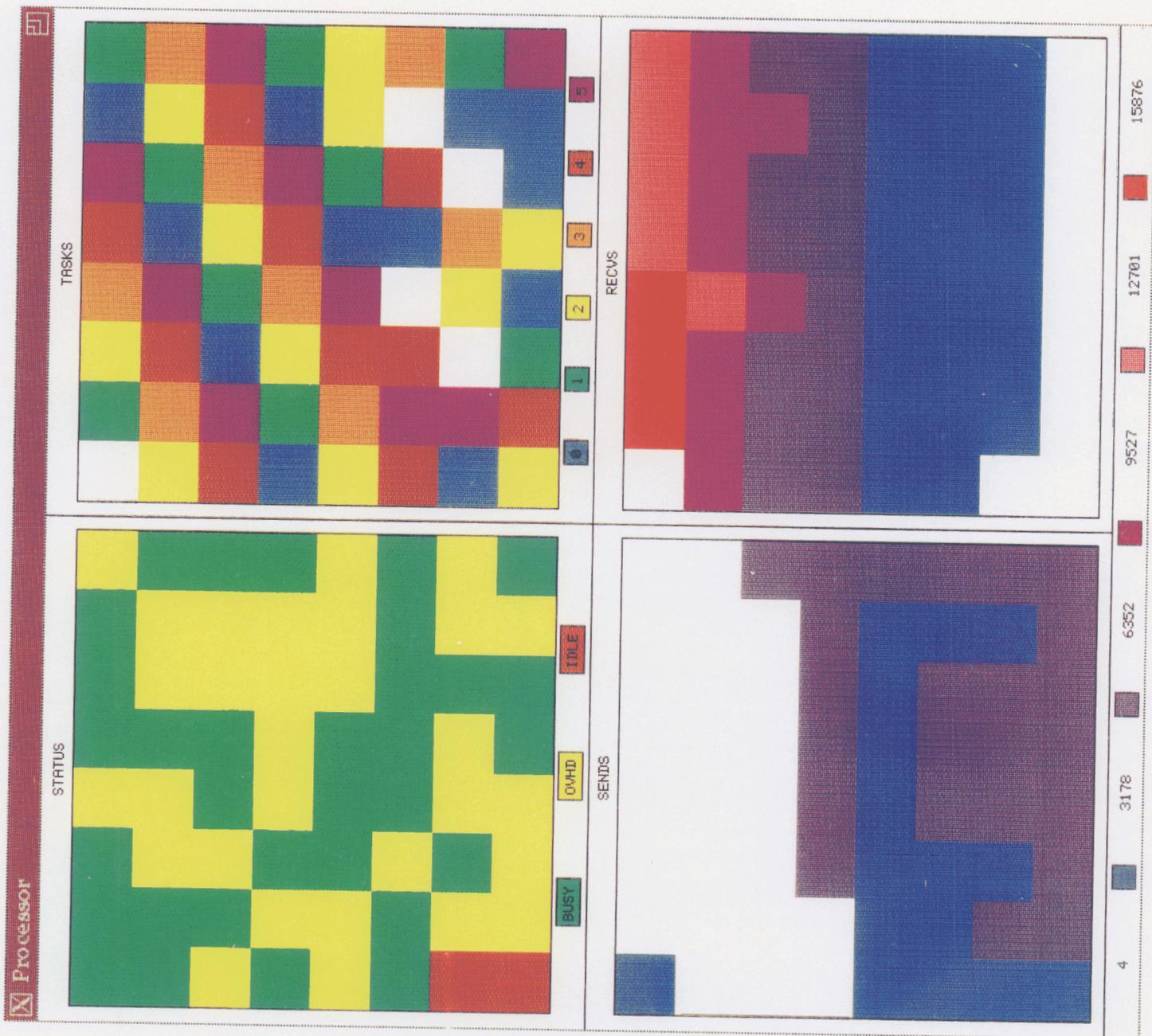


Figure 21. Processor Status display.

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 one of six colors chosen as discussed previously. The lower left subdisplay shows the volume in bytes of messages currently being sent by each processor, and the lower right subdisplay shows the volume in bytes 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 scaling to very large numbers of processors more readily than any of the other displays in ParaGraph. The example shown in Figure 21 illustrates a run with 64 processors.

4.4.4. 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 come from event timestamps.

4.4.5. 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, and the display scrolls vertically as necessary with time.

4.4.6. 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 aggregates over all processors. While this tabular display may yield considerably 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.

4.5. 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 the 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 an application-specific display. If the corresponding routines for such a display are not supplied by the user when the executable module for ParaGraph is built, then dummy "stub" routines are linked into ParaGraph instead, and no user-supplied display selection appears in the menu. When an application-specific display has been linked into ParaGraph and the resulting module is executed, the user-supplied display is given access to all of the event records in the tracefile that ParaGraph reads and can use them in any manner it chooses.

The usual events generated by PICL may suffice for the application-specific display, 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 indices, array indices, memory addresses, 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 an application-specific display 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 two such prototype displays, one for depicting parallel sorting algorithms and one for depicting parallel matrix transposition. These example routines are distributed along with the source code for ParaGraph. Figure 22 illustrates the application-specific display for matrix transposition, which is driven by **tracemarks** event records that indicate which data items are being exchanged among the processors.

5. 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. In this section, we briefly discuss some of the choices available in the Options menu.

- In many of the displays, the user can choose to have the processors arranged in either natural or Gray code order, and the choice will affect the appearance of communication patterns.
- 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.

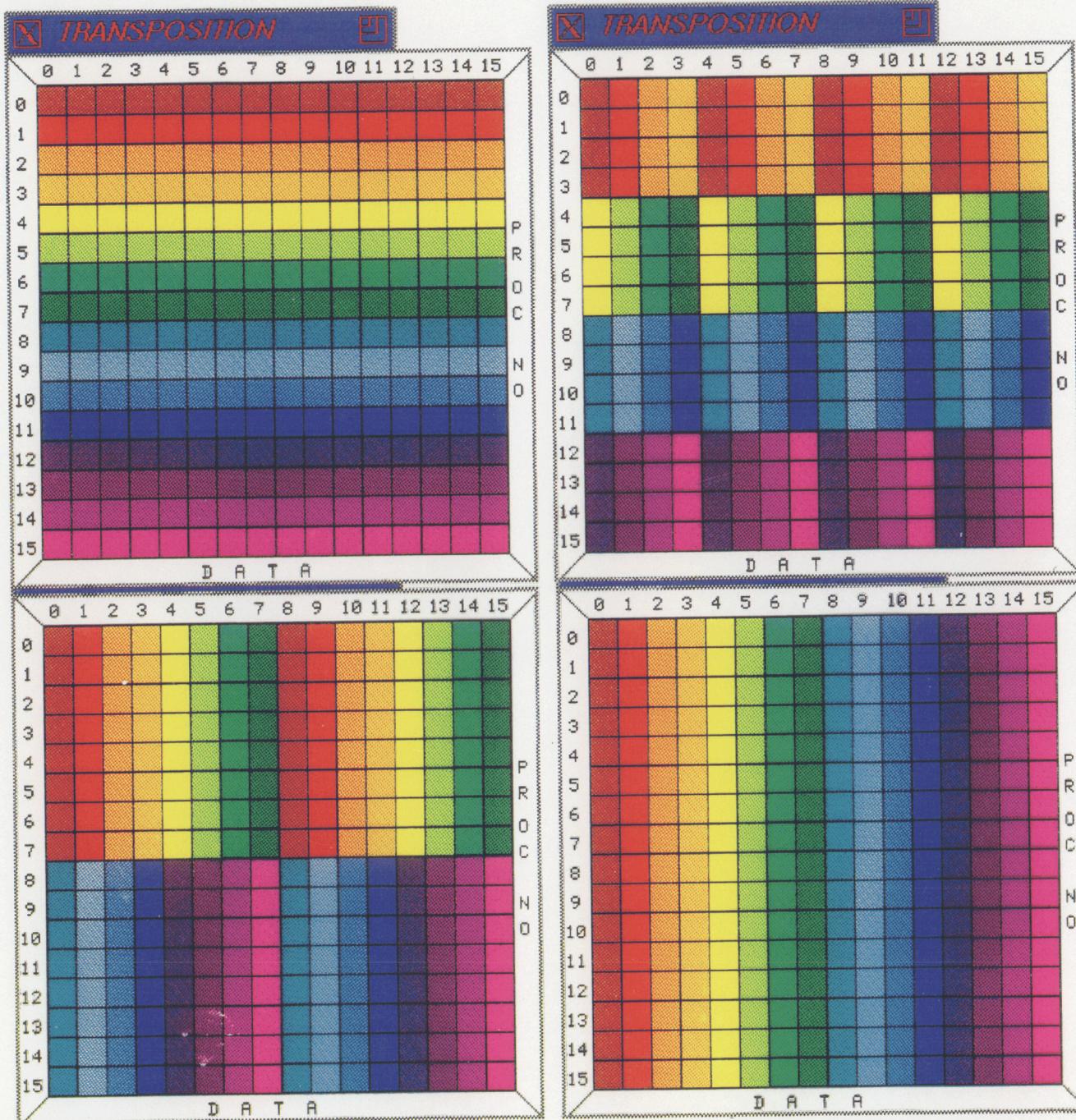


Figure 22. Four snapshots of a typical application-specific display showing several stages of recursive matrix transposition.

- The relationship between simulation time and the timestamps of the trace events is determined by the *time unit* chosen. The user can override the value that ParaGraph heuristically chooses for a given tracefile.
- A related parameter is the *scale width*, which is defined to be the width, in simulation time units, of the horizontal axis for the displays that scroll with time. The scale width chosen implicitly determines the number of pixels on the screen that represent each unit of simulation time. A larger number of pixels per time unit in effect 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. Again, the user can override the value that ParaGraph chooses.
- 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.
- The user can also select the amount of smoothing used in the Kiviat Diagram and Phase Portrait displays to avoid an excessively noisy or jumpy appearance.

6. 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 minor technical points about ParaGraph that could stand improvement. We have already mentioned that it would be nice to have more explicit control over the apparent speed of the simulation. As another example, 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 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 either 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 PICL and 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.

7. Acknowledgements

The detailed implementation of ParaGraph has been 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 one of the authors (Heath). The vast bulk of the programming was done by the other author (Etheridge) while she was an undergraduate student, first at Roanoke College and later at the University of Tennessee. Two other undergraduate

students have also worked on the development of ParaGraph: Loretta Auvil, then of Alderson Broaddus College, 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, computer graphics, workstations, or the X Window System, and within a single term each was contributing to the sophisticated software described in this paper. 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.

8. References

- [1] D.A. Bailey, J.E. Cunny, and C.P. Loomis. ParaGraph: graph editor support for parallel programming environments. *International Journal of Parallel Programming*, 19, 1990. to appear.
- [2] D. Bernstein and K. So. Performance visualization of parallel programs on a shared memory multiprocessor system. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume II, pages 1–10, August 1989.
- [3] M. Berry. The use of matrix visualization in algorithmic design. *Computing Systems in Engineering*, 1990. to appear.
- [4] O. Brewer, J. Dongarra, and D. Sorensen. Tools to aid in the analysis of memory access patterns for Fortran programs. *Parallel Computing*, 9:25–35, 1988.
- [5] G.P. Brown, R.T. Carling, C.F. Herot, D.A. Kramlich, and P. Souza. Program visualization: graphical support for software development. *IEEE Computer*, 18(8):27–35, August 1985.
- [6] M.H. Brown. *Algorithm Animation*. MIT Press, Cambridge, MA, 1988.
- [7] M.H. Brown. Exploring algorithms using Balsa-II. *IEEE Computer*, 21(5):14–36, May 1988.

- [8] M.H. Brown and R. Sedgewick. A system for algorithm animation. *Computer Graphics*, pages 177–186, July 1984.
- [9] M.H. Brown and R. Sedgewick. Techniques for algorithm animation. *IEEE Software*, 2(1):28–39, January 1985.
- [10] D. Clark. Plotting N-space cubes. *Creative Computing*, pages 148–161, July 1982.
- [11] A.L. Couch. Graphical representations of program performance on hypercube message-passing multiprocessors. Technical Report 88-4, Department of Computer Science, Tufts University, Medford, MA, April 1988.
- [12] T.A. Defanti, M.D. Brown, and B.H. McCormick. Visualization: expanding scientific and engineering research opportunities. *IEEE Computer*, 22(8):12–25, August 1989.
- [13] J. Dongarra, O. Brewer, J.A. Kohl, and S. Fineberg. A tool to aid in the design, implementation, and understanding of matrix algorithms for parallel processors. *Journal of Parallel and Distributed Computing*, 9:185–202, 1990.
- [14] J.J. Dongarra and E. Grosse. Distribution of mathematical software via electronic mail. *Communications of the ACM*, 30:403–407, May 1987.
- [15] T.H. Dunigan. Hypercube clock synchronization. Technical Report ORNL/TM-11744, Oak Ridge National Laboratory, Oak Ridge, TN, February 1991.
- [16] H. El-Rewini and T.G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9:138–153, 1990.
- [17] R.J. Fowler, T.J. LeBlanc, and J.M. Mellor-Crummey. An integrated approach to parallel program debugging and performance analysis on large-scale multiprocessors. *SIGPLAN Notices*, 24(1):163–173, January 1989.
- [18] K.A. Frenkel. The art and science of visualizing data. *Communications of the ACM*, 31:110–121, February 1988.
- [19] H.L. Gantt. Organizing for work. *Industrial Management*, 58:89–93, August 1919.

- [20] 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.
- [21] 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.
- [22] V.A. Guarna, D. Gannon, D. Jablonowski, A.D. Malony, and Y. Gaur. Faust: an integrated environment for parallel programming. *IEEE Software*, 6(4):20-27, July 1989.
- [23] D. Haban and D. Wybraniec. A hybrid monitor for behavior and performance analysis of distributed systems. *IEEE Transactions on Software Engineering*, 16(2):197-211, 1990.
- [24] M.T. Heath, E. Ng, and B.W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33(2), June 1991. to appear.
- [25] A.A. Hough and J.E. Cuny. Initial experiences with a pattern-oriented parallel debugger. *SIGPLAN Notices*, 24(1):195-205, January 1989.
- [26] S. Isoda, T. Shimomura, and Y. Ono. VIPS: a visual debugger. *IEEE Software*, 4(3):8-19, May 1987.
- [27] D. Kimelman and T. Ngo. Program visualization for RP3: an overview. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1990.
- [28] K. Kolence and P. Kiviat. Software unit profiles and Kiviat figures. *Performance Evaluation Review*, 2(3):2-12, September 1973.
- [29] M. Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Transactions on Computers*, 37(9):1088-1098, 1988.
- [30] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558-565, July 1978.

- [31] T.J. LeBlanc, J.M. Mellor-Crummey, and R.J. Fowler. Analyzing parallel program executions using multiple views. *Journal of Parallel and Distributed Computing*, 9:203–217, 1990.
- [32] T. Lehr, Z. Segall, D.F. Vrsalovic, E. Caplan, A.L. Chung, and C.E. Fineman. Visualizing performance debugging. *IEEE Computer*, 22(10):38–51, October 1989.
- [33] R.L. London and R.A. Duisberg. Animating programs using Smalltalk. *IEEE Computer*, 18(8):61–71, August 1985.
- [34] A.D. Malony. Performance observability. Technical Report UIUCDCS-R-90-1630, Department of Computer Science, University of Illinois, Urbana-Champaign, IL, October 1990.
- [35] A.D. Malony, J.W. Arendt, R.A. Aydt, D.A. Reed, D. Grabas, and B.K. Totty. An integrated performance data collection, analysis, and visualization system. In J. Gustafson, editor, *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, volume I, pages 229–236, Los Altos, CA, March 1989. Golden Gate Enterprises.
- [36] A.D. Malony and D.A. Reed. Visualizing parallel computer system performance. Technical Report UIUCDCS-R-88-1465, Department of Computer Science, University of Illinois, Urbana-Champaign, IL, September 1988.
- [37] C.E. McDowell and D.P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21:593–622, 1989.
- [38] B. Melamed and R.J.T. Morris. Visual simulation: the performance analysis workstation. *IEEE Computer*, 18(8):87–94, August 1985.
- [39] B.P. Miller. DPM: a measurement system for distributed programs. *IEEE Transactions on Computers*, 37(2):243–248, February 1988.
- [40] B.P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. Lim, and T. Torzewski. IPS-2: the second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 1:206–217, 1990.

- [41] M.F. Morris. Kiviat graphs – conventions and figures of merit. *Performance Evaluation Review*, 3(3):2–8, October 1974.
- [42] K.M. Nichols and J.T. Erdmark. Modeling multicomputer systems with PARET. *IEEE Computer*, 21(5):39–48, May 1988.
- [43] R.F. Paul and D.A. Poplawski. Visualizing the performance of parallel matrix algorithms. In D.W. Walker and Q.F. Stout, editors, *Proceedings of the Fifth Distributed Memory Computing Conference*, volume II, pages 1207–1212, Los Alamitos, CA, April 1990. IEEE Computer Society Press.
- [44] D. Pease, A. Ghafoor, I. Ahmad, D. L. Andrews, K. Foudil-Bey, T.E. Karpinski, M.A. Mikki, and M. Zerrouki. PAWS: A performance evaluation tool for parallel computing systems. *IEEE Computer*, 24(1):18–29, January 1991.
- [45] D.A. Poplawski. Synthetic models of distributed memory parallel programs. Technical Report ORNL/TM-11634, Oak Ridge National Laboratory, Oak Ridge, TN, September 1990.
- [46] S.P. Reiss. Pecan: program development systems that support multiple views. *IEEE Transactions on Software Engineering*, SE-11:276–285, 1985.
- [47] G.-C. Roman and K.C. Cox. A declarative approach to visualizing concurrent computations. *IEEE Computer*, 22(10):25–36, October 1989.
- [48] D.T. Rover. *Visualization of program performance on concurrent computers*. PhD thesis, Iowa State University, Ames, IA, 1989.
- [49] D.T. Rover, G.M. Prabhu, and C.T. Wright. Characterizing the performance of concurrent computers: a picture is worth a thousand numbers. In J. Gustafson, editor, *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, volume I, pages 245–248, Los Altos, CA, March 1989. Golden Gate Enterprises.
- [50] D.T. Rover and C.T. Wright. Pictures of performance: highlighting program activity in time and space. In D.W. Walker and Q.F. Stout, editors, *Proceedings of the Fifth Distributed Memory Computing Conference*, volume II, pages 1228–1233, Los Alamitos, CA, April 1990. IEEE Computer Society Press.

- [51] Y. Saad and M.H. Schultz. Topological properties of hypercubes. *IEEE Transactions on Computers*, 37(7):867–872, 1988.
- [52] Z. Segall and L. Rudolph. PIE: a programming and instrumentation environment for parallel processing. *IEEE Software*, 2(6):22–37, November 1985.
- [53] M. Simmons, R. Koskela, and I. Bucher, editors. *Parallel Computer Systems: Performance Instrumentation and Visualization*. ACM Press, New York, 1990.
- [54] L. Snyder. Parallel programming and the Poker programming environment. *IEEE Computer*, 17(7):27–36, July 1984.
- [55] D. Socha, M.L. Bailey, and D. Notkin. Voyeur: graphical views of parallel programs. *SIGPLAN Notices*, 24(1):206–215, January 1989.
- [56] J.T. Stasko. Tango: a framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, September 1990.
- [57] J.M. Stone. A graphical representation of concurrent processes. *SIGPLAN Notices*, 24(1):226–235, January 1989.
- [58] A. Tuchman and M. Berry. Matrix visualization in the design of numerical algorithms. *ORSA Journal on Computing*, 2(1):84–92, 1990.
- [59] E.R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT, 1983.
- [60] L.D. Wittie. Debugging distributed C programs by real time replay. *SIGPLAN Notices*, 24(1):57–67, January 1989.

Biographies

Michael T. Heath is Professor in the Computer Science Department and Senior Computer Scientist in the Center for Supercomputing Research and Development at the University of Illinois at 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.

Jennifer A. Etheridge 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 December, 1990, with a major in Mathematics. Her current interests are in computer graphics and visualization.

INTERNAL DISTRIBUTION

- | | |
|-----------------------|--------------------------------------|
| 1. B. R. Appleton | 20-24. S. A. Raby |
| 2-3. T. S. Darland | 25. C. H. Romine |
| 4. E. F. D'Azevedo | 26. T. H. Rowan |
| 5. J. J. Dongarra | 27-31. R. F. Sincovec |
| 6. J. B. Drake | 32-36. R. C. Ward |
| 7-11. J. A. Etheridge | 37. P. H. Worley |
| 12. R. E. Flanery | 38. A. Zucker |
| 13. G. A. Geist | 39. Central Research Library |
| 14. E. R. Jessup | 40. ORNL Patent Office |
| 15. M. R. Leuze | 41. K-25 Applied Technology Library |
| 16. V. E. Lynch | 42. Y-12 Technical Library |
| 17. E. G. Ng | 43. Laboratory Records - RC |
| 18. C. E. Oliver | 44-45. Laboratory Records Department |
| 19. B. W. Peyton | |

EXTERNAL DISTRIBUTION

46. John Antonishek, Bldg. 223/B364, National Institute of Standards and Technology, Gaithersburg, MD 20899
47. Donald M. Austin, 6196 EECS Bldg., University of Minnesota, 200 Union St., S.E., Minneapolis, MN 55455
48. Loretta Auvil, Dept. of Computer Science, Virginia Tech University, Blacksburg, VA 24061
49. Edward H. Barsis, Computer Science and Mathematics, P. O. Box 5800, Sandia National Laboratories, Albuquerque, NM 87185
50. Eric Barszcz, NASA Ames Research Center, Moffett Field, CA 94035
51. Donna Bergmark, Cornell National Supercomputer Facility, Engineering and Theory Center Bldg., Ithaca, NY 14853-3801
52. Fran Berman, Dept. of Computer Science, University of California - San Diego, La Jolla, CA 92093
53. Roger W. Brockett (EPMD Advisory Committee), Wang Professor of Electrical Engineering and Computer Science, Division of Applied Sciences, Harvard University, Cambridge, MA 02138
54. Eugene Brooks, P.O. Box 808, LLNL-298, Lawrence Livermore National Laboratory, Livermore, CA 94550
55. Marc Bui, I.N.R.I.A. Rocquencourt, Projet Capran, Domaine de Voluceau - B.P. 105, F-78153 Le Chesnay Cedex, France

56. Rosario Caltabiano, Silicon Graphics, One Cabot Road, Hudson, MA 01749
57. Brian M. Carlson, Dept. of Computer Science, Vanderbilt University, Nashville, TN 37235
58. John Cavallini, Office of Scientific Computing, Office of Energy Research, ER-7, Germantown Bldg., U.S. Dept. of Energy, Washington, DC 20585
59. Tony Chan, Dept. of Mathematics, University of California, Los Angeles, 405 Hilgard Ave., Los Angeles, CA 90024
60. Jagdish Chandra, Army Research Office, P.O. Box 12211, Research Triangle Park, NC 27709
61. Long Chang, SSC Laboratory, MS-1046, 2550 Beckleymeade Ave., Dallas, TX 75237
62. Arun Chatterjee, Experimental Systems, MCC, 3500 West Balcones Center Dr., Austin, TX 78759-6509
63. Doreen Y. Cheng, MS 258-6, NASA Ames Research Center, Moffett Field, CA 94035
64. Thomas Coleman, Dept. of Computer Science, Cornell University, Ithaca, NY 14853
65. Alva Couch, Dept. of Computer Science, Tufts University, Medford, MA 02155
66. Lawrence Cowsar, Dept. of Mathematical Sciences, Rice University, Houston, TX 77251
67. Jan Cuny, Dept. of Computer and Information Sciences, University of Massachusetts, Amherst, MA 01003
68. George Cybenko, Center for Supercomputing Research and Development, 305 Talbot Laboratory, University of Illinois, 104 S. Wright St., Urbana, IL 61801-2932
69. Ivo De Lotto, Dept. of Informatics and Systems, University of Pavia, Via Abbiategrasso, 209-27100 Pavia, Italy
70. James W. Demmel, Computer Science Division, University of California, Berkeley, CA 94720
71. Yuefan Deng, Applied Mathematics Dept., SUNY at Stony Brook, Stony Brook, NY 11794-3600
72. Darrin L. Dimmick, Bldg. 223/B364, National Institute of Standards and Technology, Gaithersburg, MD. 20899
73. J. J. Dorning (EPMD Advisory Committee), Dept. of Nuclear Engineering, and Engineering Physics, Thornton Hall, University of Virginia, Charlottesville, VA 22901
74. Lawrence W. Dowdy, Dept. of Computer Science, Vanderbilt University, Nashville, TN 37235
75. Jean-Marc Fellous, Center For Neural Engineering, University of Southern California, Los Angeles, CA 90089
76. Chuck Fleckentein, Honeywell Inc., MS 736-4A, 13350 Hwy 19 South, Clearwater, FL 34624

77. Geoffrey C. Fox, NPAC, 111 College Place, Syracuse University, Syracuse, NY 13244-4100
78. Joan Francioni, Computer Science Dept., University of Southwestern Louisiana, Lafayette, LA 70504
79. Dennis B. Gannon, Dept. of Computer Science, Indiana University, Bloomington, IN 47405
80. John Garnett, Dept. of Computer Science, University of Texas, Austin, TX 78712
81. W. Morven Gentleman, Div. of Electrical Engineering, National Research Council, Bldg. M-50, Room 344, Montreal Rd., Ottawa, Ontario, Canada K1A 0R8
82. J. Alan George, Vice President, Academic and Provost, Needles Hall, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1
83. Ian Glendinning, Electronics and Computer Science, University of Southampton, England SO9 5NH
84. Gene H. Golub, Dept. of Computer Science, Stanford University, Stanford, CA 94305
85. John Gustafson, Ames Laboratory, 236 Wilhelm Hall, Iowa State University, Ames, IA 50011-3020
- 86-90. Michael T. Heath, Center for Supercomputing Research and Development, 305 Talbot Laboratory, University of Illinois, 104 S. Wright St., Urbana, IL 61801-2932
91. Don E. Heller, Physics and Computer Science Dept., Shell Development Co., P.O. Box 481, Houston, TX 77001
92. Charles S. Henkel, Dept. of Nuclear Engineering, North Carolina State University, Raleigh, NC 27695,
93. Walter Hoffmann, Vakgroep Computersystemen, Universiteit van Amsterdam, Kruislaan 409, 1098 SJ Amsterdam
94. Mary E. Hribar, 23078 Johnston, East Detroit, MI 48021
95. Michelle R. Hribar, 23078 Johnston, East Detroit, MI 48021
96. Lennart Johnsson, Thinking Machines Inc., 245 First St., Cambridge, MA 02142-1214
97. Harry Jordan, Dept. of Electrical and Computer Engineering, University of Colorado, Boulder, CO 80309
98. Jeff Jortner, Sandia National Laboratories, Albuquerque, NM 87185
99. Bo Kagstrom, Institute of Information Processing, University of Umea, S-901 87 Umea, Sweden
100. Malvin H. Kalos, Cornell Theory Center, Engineering and Theory Center Bldg., Cornell University, Ithaca, NY 14853-3901
101. Hans Kaper, Mathematics and Computer Science Div., Argonne National Laboratory, 9700 South Cass Ave., Argonne, IL 60439
102. Nick Karonis, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439

103. Kenneth Kennedy, Dept. of Computer Science, Rice University, P.O. Box 1892, Houston, TX 77005
104. Karl Kesselman, Aerospace Corp., Los Angeles, CA 90053
105. Doug Kimelman, IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598
106. Andrew Kitchen, Dept. of Computer Science, Rochester Institute of Technology, P.O. Box 9887, Rochester, NY 14623-0887
107. Thomas Kitchens, Dept. of Energy, Scientific Computing Staff, Office of Energy Research, ER-7, Office G-236 Germantown Bldg., Washington, DC 20585
108. Jim Kohl, Iowa Computer Aided Engineering Network, University of Iowa, Iowa City, IA 52242
109. Steven Kratzer, Supercomputing Research Center, Institute for Defense Analyses, 17100 Science Dr., Bowie, MD 20715-4300
110. David Kuck, Center for Supercomputing Research and Development, 305 Talbot Laboratory, University of Illinois, 104 S. Wright St., Urbana, IL 61801-2932
111. J. E. Leiss (EPMD Advisory Committee), Rt. 2, Box 142C, Broadway, VA 22815
112. Eric Leu, Ecole Polytechnique Fidirale, Lab. de Syst. d'Exploitation - DI, IN - Ecublens, CH - 1015 Lausanne, France
113. Jim Laurus, Dept. of Computer Science, University of Wisconsin, Madison, WI 53706
114. Ted Lewis, Dept. of Computer Science, Oregon State University, Corvallis, OR 97331
115. Rik Littlefield, Dept. of Computer Science, University of Washington, Seattle, WA 98195
116. Stephen F. Lundstrom, P.O. Box 9535, Palo Alto, CA 94309
117. Ewing Lusk, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439
118. Allen Malony, Center for Supercomputing Research and Development, 305 Talbot Laboratory, University of Illinois, 104 S. Wright St., Urbana, IL 61801-2932
119. James McGraw, Lawrence Livermore National Laboratory, L-306, P.O. Box 808, Livermore, CA 94550
120. John Meissen, Oregon Advanced Computing Institute, 19500 S.W. Gibbs Dr., Suite 110, Beaverton, OR 97006-6907
121. Paul C. Messina, Mail Code 158-79, California Institute of Technology, 1201 E. California Blvd., Pasadena, CA 91125
122. Bart Miller, Dept. of Computer Science, University of Wisconsin, Madison, WI 53706
123. Bryan Miller, Dept. of Computer Science, Oregon State University, Corvallis, OR 97331
124. Stuti Moitra, Mail Stop 125, NASA Langley Research Center, Hampton, VA 23665

125. Cleve Moler, The Mathworks, 325 Linfield Place, Menlo Park, CA 94025
126. N. Moray (EPMD Advisory Committee), Dept. of Mechanical and Industrial Engr., University of Illinois, 1206 W. Green St., Urbana, IL 61801
127. Kathleen M. Nichols, Apple Computer, Inc., 20525 Mariani Ave., M/S 76-3K, Cupertino, CA 95014
128. Paul W. Oman, Computer Science Dept., University of Idaho, Moscow, ID 83843
129. James M. Ortega, Dept. of Applied Mathematics, Thornton Hall, University of Virginia, Charlottesville, VA 22903
130. Susan Ostrouchov, Dept. of Computer Science, University of Tennessee, Knoxville, TN 37996
131. Cherri Pancake, Dept. Computer Science and Engineering, Auburn University, Auburn, AL 36849-5347
132. Merrell Patrick, New Technologies Program, National Science Foundation, 1800 G Street, N.W., Washington, DC 20550
133. Robert J. Plemmons, Depts. of Mathematics and Computer Science, North Carolina State University, Raleigh, NC 27650
134. David Poplawski, Dept. of Mathematical Sciences, Michigan Tech University, Houghton, MI 49931
135. J. Mark Pullen, Tactical Technology Office, Defense Advanced Research Projects Agency, 1400 Wilson Boulevard, Arlington, VA 22209
136. Angela Quealy, Sverdrup Technology, Inc., NASA Lewis Research Center, Cleveland, OH 44135
137. Michael Quinn, Dept. of Computer Science, Oregon State University, Corvallis, OR 97331
138. Justin Rattner, Intel Scientific Computers, 15201 N.W. Greenbrier Pkwy., Beaverton, OR 97006
139. Dan Reed, Center for Supercomputing Research and Development, 305 Talbot Laboratory, University of Illinois, 104 S. Wright St., Urbana, IL 61801-2932
140. Michael D. Rice, Mathematics Dept., Wesleyan University, Middletown, CT 06457
141. Rich Rinehart, NASA/Lewis Research Center, Cleveland, OH 44135
142. Diane Rover, 236 Wilhelm Hall, Arnes Laboratory, Iowa State University, Ames, IA 50011
143. Joel Saltz, ICASE, MS 132C, NASA Langley Research Center, Hampton, VA 23665
144. Ahmed H. Sameh, Center for Supercomputing Research and Development, 305 Talbot Laboratory, University of Illinois, 104 S. Wright St., Urbana, IL 61801-2932
145. Nan C. Schaller, Dept. of Computer Science, Rochester Institute of Technology, P.O. Box 9887, Rochester, NY 14623-0887

146. Bobby Schnabel, Dept. of Computer Science, University of Colorado, Boulder, CO 80309
147. Robert Schreiber, RIACS, Mail Stop 230-5, NASA Ames Research Center, Moffet Field, CA 94035
148. David S. Scott, Intel Scientific Computers, 15201 N.W. Greenbrier Pkwy., Beaverton, OR 97006
149. Charles L. Seitz, Dept. of Computer Science, California Institute of Technology, Pasadena, CA 91125
150. Andrew Sherman, Dept. of Computer Science, Yale University, New Haven, CT 06520
151. Anthony Skjellum, Lawrence Livermore National Laboratory, 7000 East Ave., L-316, PO Box 808, Livermore, CA 94551
152. Lawrence Snyder, Dept. of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195
153. Rick Stevens, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439
154. G. W. Stewart, Computer Science Dept., University of Maryland, College Park, MD 20742
155. Quentin F. Stout, Dept. of Electrical and Computer Engineering, University of Michigan, Ann Arbor, MI 48109
156. Vaidy Sunderam, Dept. of Mathematics and Computer Science, Emory University, Atlanta, GA 30322
157. Dave Swan, Dept. of Computer Science, FR-35, University of Washington, Seattle, WA 98195
158. Valerie Taylor, Dept. of Electrical Engineering, University of California, Berkeley, CA 94720
159. Charles Tong, Dept. of Mathematics, University of California, Los Angeles, 405 Hilgard Ave., Los Angeles, CA 90024
160. Bernard Tourancheau, LIP, ENS-Lyon, 69364 Lyon cedex 07, France
161. Scott Townsend, NASA Lewis Research Center, Cleveland, OH 44135
162. Ray Tuminaro, Sandia National Laboratories, Albuquerque, NM 87185
163. Sue Utter, Cornell National Supercomputer Facility, 737 Engineering and Theory Center Bldg., Ithaca, NY 14853-3801
164. Bradley Vander Zanden, Dept. of Computer Science, University of Tennessee, Knoxville, TN 37996
165. Robert G. Voigt, ICASE, MS 132-C, NASA Langley Research Center, Hampton, VA 23665
166. Charles Vollum, Cogent Research, Inc., 2010 N.E. 25th, Hillsboro, OR 97124
167. Phuong Vu, Cray Research Inc., 655F Lone Oak Dr., Eagan, MN 55121
168. Thomas D. Wagner, Dept. of Computer Science, Vanderbilt University, Nashville, TN 37235

169. Gil Weigand, Defense Advanced Research Projects Agency, 1400 Wilson Boulevard, Arlington, VA 22209
170. Tammy Welcome, P.O. Box 808, LLNL-298, Lawrence Livermore National Laboratory, Livermore, CA 94550
171. M. F. Wheeler (EPMD Advisory Committee), Dept. of Mathematical Sciences, Rice University, P.O. Box 1892, Houston, TX 77251
172. Andrew B. White, Computing Div., Los Alamos National Laboratory, Los Alamos, NM 87545
173. Michael Wolfe, Oregon Graduate Institute, 19500 N.W. Von Neumann Dr., Beaverton, OR 97006-1999
174. Markus Zellner, Computer Science Dept., Australian National University, GPO Box 4, Canberra A.C.T., 2601, Australia
175. Office of Assistant Manager, for Energy Research and Development, U.S. Dept. of Energy, Oak Ridge Operations Office, P.O. Box 2001, Oak Ridge, TN 37831-8600
- 176--185. Office of Scientific, & Technical Information, P.O. Box 62, Oak Ridge, TN 37831