Visualizing the Performance of Parallel Programs

MICHAEL T. HEATH, *University of Illinois*
JENNIFER A. ETHERIDGE, *Oak Ridge National Laboratory*

◆ *ParaGraph animates trace information from actual runs to depict behavior and provides graphical performance summaries. It provides 25 perspectives on the same data, lending insight that might otherwise be missed.*

Graphical visualization aids human comprehension of complex phenomena and large volumes of data. The behavior of parallel programs on advanced architectures is often extremely complex, and monitoring the performance of such programs can generate vast quantities of data. So it seems natural to use visualization to gain insight into the behavior of parallel programs so we can better understand them and improve their performance.

We have developed ParaGraph, a software tool that provides a detailed, dynamic, graphical animation of the behavior of message-passing parallel programs and graphical summaries of their performance.

GRAPHICAL SIMULATION

For lack of a better term, we use "simulation" to mean graphical animation. By "simulation," we do not mean to suggest that there is anything artificial about the programs or their behavior as ParaGraph portrays them. ParaGraph displays the behavior and performance of real parallel programs running on real parallel computers to solve real problems. In effect, ParaGraph 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 for post-processing trace files created during execution and saved for later study. But its design does not rule out the possibility that data could arrive at the graphical workstation as the program executes.

However, there are major impediments to genuine real-time performance visualization. With the current generation of distributed-memory parallel architec-

0740-7459/91/0900/0029/$01.00 © IEEE

tures, it is difficult to extract performance data from the processors and send it to the outside world during execution without significantly perturbing the program being monitored. Also, the network bandwidth between the processors and the workstation and the drawing speed of the workstation are usually inadequate to handle the very high data-transmission rates that a real-time display requires. Finally, humans would be hard pressed to digest a detailed graphical depiction unfolding in real time. One of ParaGraph's strengths is that it lets you replay the same execution trace data repeatedly.

Some performance-visualization packages treat the trace file of events saved after a program executes as a static, immutable object to be studied by various analytical or statistical means. Such packages provide graphical tools designed for visual browsing of the performance data from various perspectives using scroll bars and the like.

ParaGraph adopts a more dynamic approach whose conceptual basis is algorithm animation. We see the trace file as a script to be played out, visually reenacting the original live action to provide insight into a program's dynamic behavior.

Both the static and dynamic approaches have advantages and disadvantages. Algorithm animation is good at capturing a sense of motion and change, but it is difficult to control the simulation's apparent speed. The static browser approach gives the user fine control over the speed at which the data are viewed (time can even move backward), but it does not provide an intuitive feeling for dynamic behavior.

Design goals. We wanted ParaGraph to be easy to understand, easy to use, and portable.

Easy to understand. The whole point of visualization is to aid understanding, so it is imperative that the visual displays be as intuitively meaningful as possible. The charts and diagrams should be aesthetically appealing and the information they convey should be self-evident. A diagram is not likely to be useful if it requires an extensive explanation, so the information it conveys should either be immediately obvious or easily remembered once learned.

The display's colors should reinforce the meaning of graphical objects and be consistent across views. Above all, the system must provide many visual perspectives, because no single view is likely to provide full insight into the behavior and data associated with parallel-program execution. ParaGraph provides more than 20 displays or views based on the same underlying trace data.

Easy to use. Software tools should relieve tedium, not promote it. We use color and animation to make ParaGraph painless, even entertaining, to use. ParaGraph has an interactive, mouse- and menu-oriented user interface so its features are easily invoked and customized.

Another important factor in ease of use is that the object under study (the parallel program) need not be modified extensively to obtain the visualization data. ParaGraph's input are trace files produced by the Portable Instrumented Communication Library,[1] which lets users produce trace data automatically.

The whole point of visualization is to aid understanding, so it is imperative that the visual displays be as intuitively meaningful as possible.

Portability. Portability is important in two senses. First, the graphics package itself should be portable. ParaGraph is based on X Windows, and thus runs on many vendors' workstations. Although it is most effective in color, it also works on monochrome and grayscale monitors — it detects automatically which monitor type is in use.

Second, the package must be able to display 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 many different vendors (including Cogent, Intel, N-Cube, and Symult).

On the other hand, many of ParaGraph's displays are based on the message-passing paradigm, so it does not support programs explicitly based on shared-memory constructs.

PARAGRAPH FEATURES

ParaGraph is distinguished from other visualization systems[2] in:

♦ The number of displays it provides. While other packages provide multiple views, none we know of provides the variety of perspectives ParaGraph does. Some of ParaGraph's displays are original; others have been inspired by similar displays in other packages.

♦ Its portability among architectures and displays. ParaGraph is applicable to any parallel architecture having message passing as its programming paradigm, and ParaGraph itself is based on X Windows.

♦ The intuitive appeal and aesthetic quality of its displays, which we hope reach a new, higher standard. Of course, how successful we've been is in the eye of the beholder.

♦ Its ease of use, attributable both to its interactive, graphical interface and to its use of PICL to provide the trace data without requiring the user to instrument the program under study.

♦ Its extensibility. ParaGraph lets users add new displays of their own design to the views already provided.

An indication of how successful we've been in making ParaGraph easy to use and understand is the fact that many users have obtained an early version from Netlib on Internet during the last year, built the program, and used it effectively without the benefit of any documentation except a one-page Readme file.[3]

Relationship to PICL. PICL runs on several message-passing parallel architectures.[1] As its name implies, it provides both portability and instrumentation for programs that use its communication facilities to pass messages among processors.

On request, PICL provides a trace file that records important execution events (like sending and receiving messages). The trace file contains one event record per line, and each event record comprises an integer set that specifies the event type, time stamp, processor number, message length, and other similar information.

ParaGraph has a producer-consumer relationship with PICL: ParaGraph consumes the trace data PICL produces. Using PICL instead of a machine's native parallel-programming interface gives the user portability, instrumentation, and the ability to use ParaGraph to analyze behavior and performance.

These benefits are essentially "free" in that once you've implemented a parallel program using PICL, you don't have to change the source code to move it to a new machine (provided PICL is available on the new machine), and little or no effort is required to instrument the program for performance analysis.

On the other hand, because ParaGraph's dependence on PICL is solely for input data, ParaGraph could work equally well with any data source that has the same format and semantics. So other message-passing systems can be instrumented to produce trace data in ParaGraph's format, and ParaGraph's input routine can be adapted to different input formats. Indeed, ParaGraph has been used with communication systems other than PICL.

Clock synchronization. For meaningful simulation, the event's time stamps should be as accurate and consistent across processors as possible. This is not necessarily easy when each processor has its own clock with its own starting time and runs at its own rate. Also, the clock's resolution may be inadequate to resolve events precisely.

Poor clock resolution and/or synchronization can lead to what we call *tachyons* in the trace file — messages that appear to be received before they are sent (a tachyon is a hypothetical particle that travels faster than light). Tachyons confuse ParaGraph, because much of its logic depends on pairing sends and receives.

Because this possibility will invalidate some ParaGraph displays, PICL goes to considerable lengths to synchronize processor clocks and adjust for potential clock drift, so time stamps are as consistent and meaningful as possible. On some machines, PICL actually provides higher clock resolution than the system was supplied with.

Overhead. Collecting trace data can add to overhead. PICL tries to minimize tracing perturbation by saving trace data in each processor's local memory, downloading them to disk only after the program has finished execution. Nevertheless, monitoring inevitably introduces extra overhead: In PICL, the clock calls necessary to determine the time stamps for the event records, plus other minor overhead, add a fixed amount (independent of message size) to the cost of sending each message.

Because ParaGraph's dependence on PICL is solely for input data, ParaGraph could work equally well with any data source that has the same format and semantics.

Thus, the overhead added is a function of the frequency and volume of communication traffic; it also varies from machine to machine. In general, we believe that this perturbation is small enough that the program's behavior is not altered fundamentally. In our experience, the lessons we learn from the visual study of instrumented runs always improve the performance of uninstrumented runs.

USING PARAGRAPH

ParaGraph supports command-line options that specify a host name for remote display across a network, forced monochrome display mode (useful if black-and-white hard copies are to be made from a color screen), or a trace-file name. You can also specify (or change) the trace-file name during execution by typing the file name in an options menu. ParaGraph preprocesses the input trace file to determine some parameters (like time scale and number of processors) automatically, before the simulation begins; the user can override most of these values.

Interface. The initial ParaGraph display is a main menu of buttons with which you control the execution and select submenus. Submenus include those for three display types, or families (utilization, communication, and tasks), for miscellaneous displays, and for options and parameters.

You can select as many displays as will fit on the screen, and you can resize displays within reason. It is difficult to pay close attention to many displays at once, but it is useful to have several simultaneous displays for comparison and selective scrutiny.

After selecting the displays, you press start to begin the graphical simulation of the parallel program based on the trace file you specified. The animation proceeds to the end of the trace file, but you can interrupt it with a pause/resume button. For even more detailed study, the step button provides a single-step mode that processes the trace file one event at a time.

You can also single out a time interval by specifying a starting and stopping time (the defaults are the beginning and ending of the trace file), or you can have the simulation stop at each occurrence of some event. And you can restart the entire animation at any time by simply pressing the start button.

Some ParaGraph displays change in place as events occur, representing execution time from the original run with simulation time in the replay. Other displays represent execution time in the original run with one space dimension on the screen, scrolling (by a user-controllable amount) as simulation time progresses, in effect providing a moving window for viewing a static picture. No matter which time representation is used, all displays are updated simultaneously and synchronized with each other.

Speed. The relationship between simulation speed and execution speed is necessarily 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 to make the simulation speed uniformly proportional to the original execution speed.

For the most part, ParaGraph simply processes the event records and draws the resulting displays as fast as it can. If there are gaps between consecutive time stamps, ParaGraph fills them in with a spin loop so there is at least a rough (if not uniform) correspondence between simulation time and execution time. Fortunately, close correspondence does not seem to be critical in visual performance analysis. What's most important is that the graphical replay preserve the correct relative order of events. Moreover, the figures of merit ParaGraph produces are based on actual time stamps, not simulation speed.

Because ParaGraph's speed is determined primarily by the workstation's drawing speed, the number of displays you select can speed it up or slow it down. ParaGraph's speed is also affected by the displays' complexity and the type and amount of scrolling used.

When ParaGraph provided only a few displays, it included parameterized delay loops to slow the drawing in case it moved too quickly for the eye to follow. But as we added more displays this ceased to be a problem, so we dispensed with the delay loops. Now users sometimes complain that the simulation is too slow rather than too fast, since most like to have many displays open. You can always resort to single-step mode if you want to study program behavior very closely.

With one noted exception, all of the views shown can display at least 128 processors and most of them can display up to 512 processors.

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 each event's nature. Menu selections determine ParaGraph's execution behavior, both statically (initial display selection, options, and parameter values) and dynamically (pause/resume, single-step mode).

ParaGraph has two event queues: a queue of X events produced by the user (mouse clicks, key presses, window exposures) and a queue of trace events produced by the program under study. ParaGraph must alternate between these queues to provide both a dynamic depiction of the program and responsive user interaction.

The X-event queue must be checked frequently enough to provide good responsiveness, but not so frequently as to degrade drawing speed. The trace-event queue must be processed as rapidly as possible while the simulation is active, but it need not be checked at all if the next possible event must be an X event (as happens before a simulation starts, after it finishes, when it is in single-step mode, or when it has been paused and can be resumed only by the user).

So ParaGraph's alternation between queues is not strict. Because not all event records PICL produces are of interest to ParaGraph, it fast-forwards through such uninteresting records before it rechecks the X-event queue. Also, ParaGraph checks the X-event queue with both blocking and nonblocking calls, depending on the circumstances, so workstation resources are not consumed unnecessarily when the simulation is inactive.

DISPLAYS

Printed illustrations obviously cannot convey ParaGraph's dynamism, but we must be content with snapshots. Due to space limitations, we cannot illustrate all 25 displays, so we selected the most useful and interesting ones from a typical ParaGraph session. For clarity and simplicity, the examples use only a few processors. With one noted exception, all of the views shown can display at least 128 processors and most of them can display up to 512 processors. The figures reproduced here were produced from trace files made on an Intel iPSC/2 hypercube.

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. (See our technical report for details of the parallel algorithm used here.[4])

In the example, the sparse matrix of the linear system arises from a 15×15 square grid, so 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 first computes the portion of the factorization corresponding to the interior of its own part of the grid, which it can do independent of the other processors. However, eventually the processors reach a point where interprocessor communication is required to supply boundary data from neighboring grid portions before computations can proceed. The processors team up in four pairs, then in two sets of four, and finally all eight together, as they work their way up the elimination tree and communicate across higher level boundaries.

Most of the displays fall into one of three categories — utilization, communication, and task information — although some contain more than one type of information and a few do not fit these categories at all.

Utilization displays. Figure 1 shows processor-utilization displays, which show how effectively processors are used and how evenly the computational work is distributed among them.

Processor count. The utilization-count display (top-left window in Figure 1) 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 is borrowed from traffic signals: green (go) for busy, yellow (caution) for overhead, and red (stop) for idle. Along the vertical axis we show green at the bottom, yellow in the middle, and red at the top.

ParaGraph categorizes a 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.

Because these 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. So busy time may well include redundant work or other work that would not be necessary in a serial program, because our monitor detects only that overhead associated with communication.

However, we find that in practice these definitions are quite adequate to illustrate the effectiveness of parallel programs. In Figure 1, the perfectly parallel initial phase of the divide-and-conquer algorithm for sparse-matrix factorization corresponds to the all-green portion at the far left, before subsequent communication causes processor use to decline.

Gantt chart. This display (bottom-left window in Figure 1), patterned after graphical charts used in industrial management,[5] depicts the activity of individual processors with 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; time on the horizontal axis, which scrolls as necessary as the simulation proceeds. The Gantt chart provides the same information as the utilization-count display, but by individual processor instead of by aggregate. As Figure 1 shows, 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 in between.

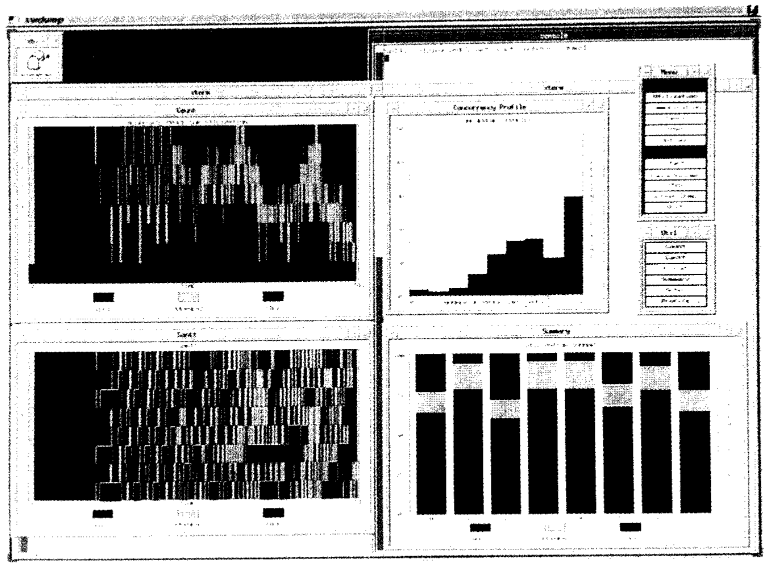


Figure 1. Processor-utilization displays. Top row, from left: count of processors in each of three states; percentage of time a certain number of processors were in a given state. Bottom row, from left: activity of individual processors over time; percentage of time each processor spent in each state.

Summary. The utlization-summary display (bottom-right window in Figure 1) shows the percent of time over the entire run that each processor spent in each of the three states (busy, overhead, and idle). The percentage is shown on the vertical axis; the processor number on the horizontal axis. This display also uses the traffic-light color scheme.

In addition to a visual impression of overall program efficiency, this display gives a visual indication of the load balance across processors. In the sparse-matrix example shown in Figure 1, 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.

Concurrency profile. This display (top-right window in Figure 1) shows the percentage of time that a certain number of processors were in a given state. The percentage of time is shown on the vertical axis; the number of processors on the horizontal axis. The profile for each possible state is shown separately; the user can cycle through the three states by clicking on a 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, which assumes that at any given time the computational work is either strictly serial or fully parallel. Figure 1 shows the busy profile for the sparse-matrix example; the

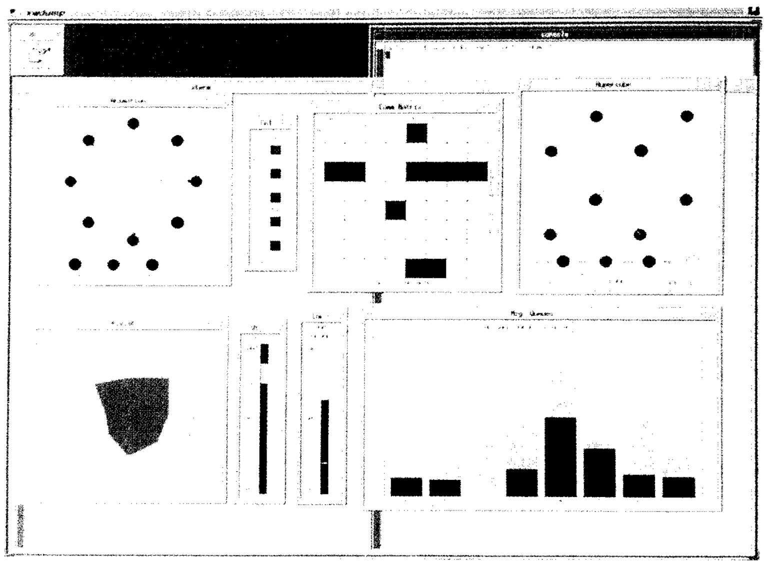


Figure 2. Communication and utilization displays. Top row, from left: logical connectivity of multiprocessors; message length; matrix of message size, duration, and pattern; node layouts that correspond to networks embedded in a hypercube. Bottom row, from left: geometric depiction of an individual processor's use and overall load balance; percentage of maximum number of processors in each of three states; current communication volume as a percentage of the maximum; size of each processor's incoming message queue.

idle and overhead profiles are not shown.

Utilization meter. This display (bottom-center window in Figure 2) uses a colored vertical bar, with the traffic-light color scheme, to indicate the percentage of the maximum number of processors that are currently in each of the three states.

The visual effect, shown in Figure 2, is similar to a thermometer. This display provides essentially the same information as the utilization-count display, but saves screen space because it changes in place rather than scrolling with time.

Kiviat diagram. This display (bottom-left window in Figure 2), adapted from related graphs used in other types of performance evaluation,[6-7] gives a geometric depiction of individual processor's usage and the overall load balance across processors.

As Figure 2 shows, each processor is represented as a spoke on a wheel. The recent average fractional usage of each processor determines a point on its spoke, with the hub of the wheel representing zero (completely idle) and the outer rim representing one (completely busy).

Taken together, the points for all the processors determine the vertices of a polygon whose size and shape illustrate both processor use and load balance. Low usage concentrates the polygon near the center, while high usage causes the polygon to lie near the perimeter. Poor load balance across processors causes it to be strongly skewed or asymmetric. Any change in load balance is clearly shown: With many ring-oriented algorithms, for example, the moving polygon has the appearance of a rotating camshaft as the heavier workload moves around the ring.

In Figure 2, current usage is shown in dark shading, and the "high-water mark" thus far has a lighter shading. The current usage is a moving average over a user-specified time, because instantaneous usage would always be either zero or one for each processor.

Communication displays. Interprocessor-communication displays are helpful in determining communication frequency, volume, and overall pattern, and whether there is congestion in the message queues.

Message queues. This display (bottom-right window in Figure 2) depicts the size of the each processor's incoming-message queue with a vertical bar whose height varies with time as messages are sent, buffered, and received. The processor number is shown on the horizontal axis.

You can set the queue size to be measured either by the number of messages or by their total length in bytes. A processor's input queue size is incremented each time a message is sent to that processor, and decremented each time the user process on that processor receives a message. On most message-passing parallel systems, transmission time between processors is negligible compared to the overhead in handling messages, so the time between send and receive events is a reasonable approximation of the time a message actually spends in the destination processor's input queue.

Depending on their types, messages may not be received in the same order in which they arrive for queuing, so queues may grow and shrink in complicated ways. As before, dark shading depicts the current queue size on each processor, and lighter shading the high-water mark.

This display gives a pictorial indication of whether there is communication congestion (if messages are accumulating in the input queue) or if 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 the receiving processor does not become idle. But a large backlog of incoming messages can consume excessive buffer space, so a happy medium is desirable. In the example shown in Figure 2, processor 2 has no messages in its input queue, the other processors all have messages awaiting receipt by their user processes, and no queue is at its maximum size so far.

Communication matrix. In this display (top-center window in Figure 2), messages are represented by squares in a two-dimensional array whose rows and columns correspond to each message's sending and receiving processor. During simulation, each message is depicted by coloring the appropriate square when the message is sent and erasing it when the message is received.

The square's color indicates message size in bytes, as given in a separate color-code display. Thus, this display shows message size, duration, and overall pattern. The nodes can be ordered along the axes in either natural or Gray code order; your choice will strongly affect the communication pattern. At the end of the simulation, this display shows the cumulative communication volume for the entire run between each pair of processors.

Animation. In this display (top-left window in Figure 2), the multiprocessor is represented by a graph whose nodes (numbered circles) represent processors, and whose arcs (lines between circles) represent communication links.

Each node's status (busy, idle, sending, and receiving) is indicated by its color, so the circles are like the multiprocessor's front-panel lights. An arc is drawn between the source and destination processors when a message is sent, and erased when the message is received. So both node color and graph connectivity change as simulation proceeds.

The small circles are arranged in a large circle merely for convenience in drawing straight lines between processor pairs without intersecting any other processors; this arrangement is not meant to suggest that the underlying architecture is a ring. The nodes can be ordered around the circle in either natural or Gray code order; again, your choice will strongly affect the communication pattern.

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.

Various combinations of states are possible for the sending and receiving processors. As Figure 2 shows, the processors on both ends of a message line can be busy, one having already sent the message and resumed computing, while the other has not yet stopped computing to receive it. At the end of a simulation, this display shows a summary of all logical communication links used in the run.

Hypercube. This display (top-right window in Figure 2) is similar to the animation display, except it provides additional node layouts to exhibit more clearly communication patterns that correspond to the various networks that can be embedded in a hypercube. The layouts provided include ring, ring of rings, web, cube, lateral cubes, nested cubes, mesh, linear, tree, tesseract (four-dimensional cube), and polytope arrangements.

This display does not require that a machine's interconnection network be a hypercube — it highlights the hypercube structure merely as a matter of interest. The scheme for coloring nodes and drawing arcs is the same as the animation display, except that curved arcs are often used to avoid, as much as possible, intersecting intermediate nodes.

The layouts provided include ring, ring of rings, web, cube, lateral cubes, nested cubes, mesh, linear, tree, tesseract (four-dimensional cube), and polytope arrangements.

To help the user of a hypercube determine if the network's physical connectivity is honored by the program's communication, 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 2, the message between nodes 2 and 7 must travel over a virtual link by being forwarded through an intermediate processor, whereas the message between nodes 2 and 6 travels directly over the physical link between those two processors. As in the animation display, at the conclusion of the simulation a summary of all logical communication links used during the run is shown. Unfortunately, the method used to draw the hypercube display does not scale up to large numbers of processors, and it is limited to displaying at most 16 processors.

Communication meter. This display (bottom-center window in Figure 2) uses a vertical bar to indicate the current communication volume as a percentage of the maximum. This display provides essentially the same information as the communication traffic display, but saves screen space by changing in place rather than scrolling with time. Conceptually, this thermometer-like display is similar to the utilization-meter display, and the two are interesting to observe side by side.

Communication traffic. This display (top-left window in Figure 3) 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 length or number of messages sent but not yet received. It can be expressed by message count or by volume in bytes.

The communication traffic can also be either the aggregate over all processors or just the messages pending for any individual processor. Message volume or count is shown on the vertical axis; time on the horizontal axis, which scrolls as necessary. Figure 3 shows the successively higher peaks in communication traffic for the sparse-matrix example as the program encounters higher level grid separators.

Space-time diagram. Patterned after diagrams used in relativity theory, this display (top-right window in Figure 3) depicts interactions among processors through space and time. This diagram has been used by Leslie Lamport to describe the order of events in a distributed system.[8] The same pictorial concept was used over a century

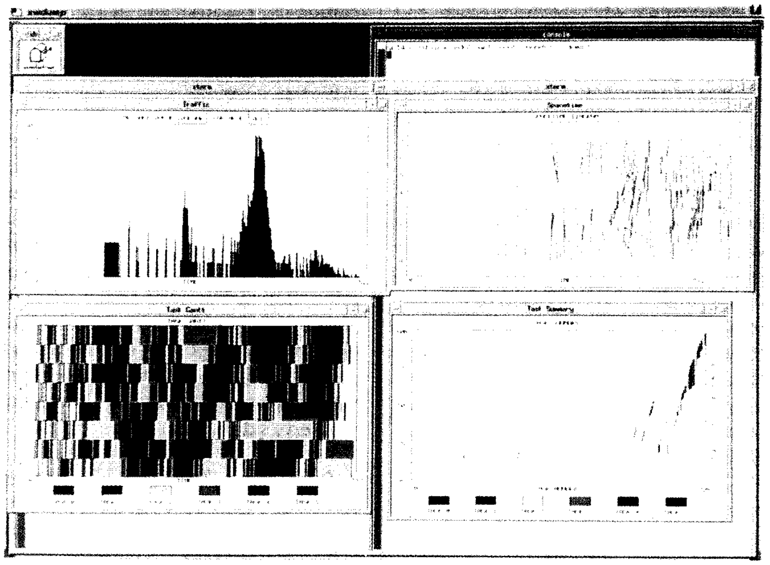


Figure 3. *Communication and task displays. Top row, from left: plot of total communication traffic over time; interactions among processors through space and time. Bottom row, from left: each processor's task activity over time; duration of each task as a percentage of total execution time.*

ago to prepare graphical railway schedules.[9]

In our adaptation of the space-time diagram, processor number is on the vertical axis, time on the horizontal, which scrolls as necessary. Processor activity (busy or idle) is indicated by horizontal lines, one for each processor, with the line drawn solid if the corresponding processor is busy (or doing overhead), and blank if the processor is idle.

Messages between processors are depicted by slanted lines between the sending and receiving processor activity lines, indicating the times at which each message was sent and received. These sending and receiving times are from user process to user process (not simply the physical transmission time), and hence the slopes of the resulting lines give a visual indication of how soon a piece of data produced by one processor is needed by the receiving processor. The communication lines are color-coded to indicate the sizes of the messages being transmitted.

The space-time diagram shown in Figure 3 clearly shows the divide-and-conquer nature of the sparse-matrix example. The eight processors initially work independently, then combine in successively larger groups as they move up the elimination tree. The space-time diagram is one of the most informative of all the displays, because it depicts both individual processor utilization and all message traffic in full detail. For example, it can easily be seen which message wakes up an idle processor that was blocked. Unfortunately, this fine level of detail does not scale up beyond about 128 processors, as the diagram becomes extremely cluttered.

Task displays. The displays thus far have depicted a number of important aspects that help detect performance bottlenecks. However, they contain no information about where in the parallel program the behavior occurs.

To remedy this situation, we considered several automated approaches to providing such information (such as picking up line numbers in the source code from the compiler), but all of these involved nasty practical difficulties. So we reluctantly made an exception to our rule that the ParaGraph user need do nothing to instrument the program under study.

We developed several new task displays that use information provided by the user and PICL to depict the portion of the program that is executing. The user defines tasks within a program by using special PICL routines to mark the beginning and end of each task and assign it a 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. For example, in matrix factorization you might define the computation of each column to be a task and assign the column number as the task number. You need to define tasks only if you want to view the task displays. If the trace file contains no event records that define tasks, the task displays will simply be blank.

Tasks can be nested, but if they are they 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 here, task numbers are color-coded. Because the number of tasks is likely to be larger than the number of colors that can be easily distinguished, ParaGraph recycles 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 factorization computation of each column to be a separate task, with the column number as task number, for a total of 225 tasks.

Task Gantt. This display (bottom-left window in Figure 3) depicts the task activity of individual processors with 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; time on the horizontal axis, which scrolls as necessary.

You can compare this display with the utilization Gantt chart to correlate a processor's busy-overhead-idle status with its task. Comparing Figure 3 with Figure 1 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.

Task summary. This display (bottom-right window in Figure 3) indicates the duration of each task (from earliest beginning to last completion by any processor) as a percentage of the overall execution time and also places the duration interval of each task within the overall execution interval of the program. The percentage of the total execution time is shown on the vertical axis, the task number on the horizontal axis. As Figure 3 shows, this display provides another striking depiction of the divide-and-conquer sparse-matrix example, with the 8-4-2-1 sequence clearly visible.

Other task displays, not shown here, include task count, which shows the number of processors that are executing a given task at a given time, and task status, which indicates whether a task has yet to begin, is currently in progress, or has been completed.

Other displays. Some displays either do not fit into a category or cut across more than one category.

Critical path. Similar to the space-time diagram, this display (top window in Figure 4) uses a different color coding to highlight the longest serial thread in a parallel computation.

As Figure 4 shows, the processor and message lines along the critical path are shown in red, while all other processor and message lines are in light blue. This display helps identify performance bottlenecks and tune the parallel program by focusing on the part of the computation that is limiting performance. Any improvement in performance must necessarily shorten the longest serial thread running through the computation, so this is the first place to look for potential algorithm improvements.

Phase portrait. Patterned after phase portraits used in differential equations and classical mechanics to depict the relationship between two variables that depend on some independent variable, these displays (bottom-left and bottom-right windows in Figure 4) illustrate the relationship over time between communication and processor use.

At any point in time, the percentage of processors in the busy state and the percentage of the maximum volume of communication in transit together define a single point in a two-dimensional plane. This point changes with time as communication and processor use vary, thereby tracing out a trajectory in the plane that is plotted in this display, with communication and use on the two axes.

Because the overhead and potential idleness due to communication inhibit processor use, you expect communication and use generally to have an inverse relationship. Thus the phase trajectory should tend to lie along a diagonal. This display reveals repetitive or periodic behavior, which tends to show up in the phase portrait as an orbit pattern.

The bottom-left window of Figure 4 shows two distinct computation phases, each of which exhibits a high degree of periodic behavior. By setting task numbers, you can color-code the trajectory to highlight either major phases in a computation or individual orbits. For example, the two phases shown in Figure 4 are matrix factorization (blue) and triangular solution (green) in solving a system of linear equations, and in the bottom-right window each separately colored orbit is a different dimension of a fast Fourier transform.

Other displays not shown here include processor status, which is a comprehensive display that attempts to capture detailed information about processor use, communication, and tasks in a compact format that scales up well to very large numbers of processors; clock, which provides both digital and analog clock readings during simulation; trace, a textual display of an annotated version of each trace event as it is read from the trace file (useful primarily in single-step mode for debug-

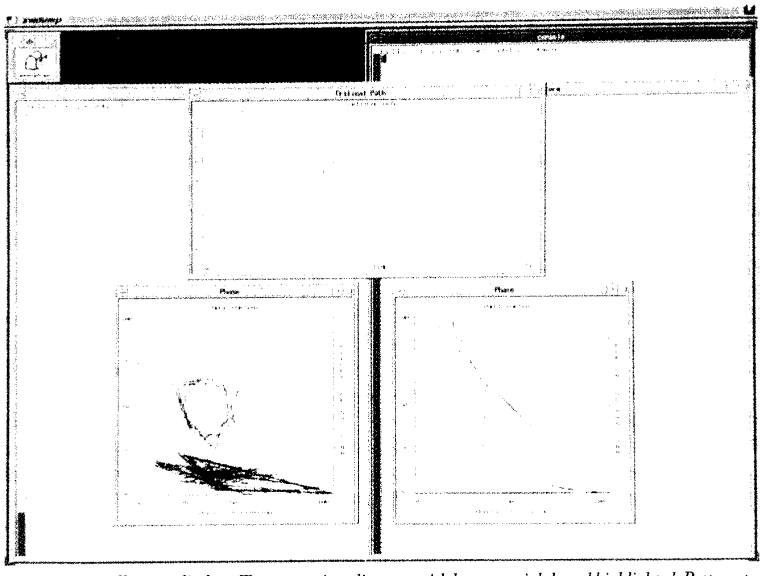


Figure 4. Miscellaneous displays. Top: space-time diagram with longest serial thread highlighted. Bottom: two phase portraits that illustrate the relationship between communication and processor use over time.

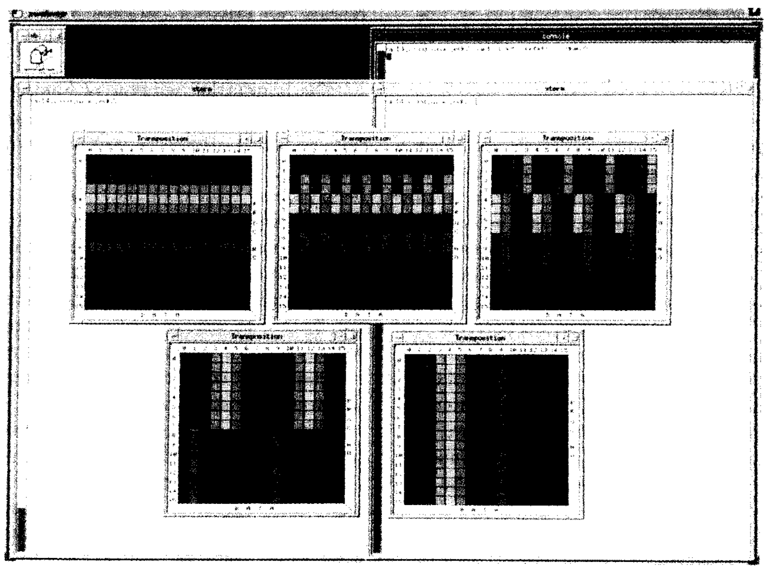


Figure 5. Successive snapshots of an application-specific display showing several stages of recursive matrix transposition.

ging or other detailed study on an event-by-event basis); and statistical summary, a textual display of numerical values for various statistics summarizing processor use and communication, both for individual processors and processor aggregates (useful in preparing tables and graphs that require exact numerical quantities for printed reports or for analytical performance modeling).

Application-specific displays. All the displays shown so far are generic — they are applicable to any parallel program based on message-passing. In general, this wide applicability is a virtue, but knowledge of the application often lets you design a special-purpose display that reveals greater detail or insight than generic displays would permit.

For example, if you are studying a parallel-sorting algorithm, generic displays can show which processors communicate and the communication volume, but not which specific data items are being exchanged among processors. Obviously, ParaGraph cannot provide such application-specific displays, but it is extensible so users can add application-specific displays of their own design that they can add to the menu and view along with the generic displays.

ParaGraph contains calls at appropriate points to application-specific routines for the initialization, data input, event handling, and drawing of displays. If the user does not supply these routines, dummy "stub" routines are instead linked into ParaGraph when the executable module is built. 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 trace file that ParaGraph reads and can use them in any manner it chooses.

The events generated by PICL may suffice for the application-specific display. Or you can insert additional events during execution to supply additional data for the application-specific display.

PICL's tracemarks event is perhaps the most useful for this purpose, because it lets users insert into the trace file time-stamped records containing arbitrary lists of integers, which might be used to provide loop indices, array indices, memory addresses, or other information that would let the application-specific display convey more fully and precisely the program's activity 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 Windows programming. But at least the user of this capability can concentrate on only those portions of the graphics programming that are relevant to his application, taking advantage of ParaGraph's supporting infrastructure to provide all of the other necessary facilities to drive and control the overall graphical simulation and provide a meaningful context in which to view the application-specific information.

To help users who want to develop application-specific displays, we have developed several prototype displays to depict parallel-sorting algorithms, matrix transposition, and various other matrix computations. These example routines are distributed along with the source code for ParaGraph. Figure 5 shows successive snapshots of an application-specific display for matrix transposition that is driven by event records that indicate which data items are being exchanged among the processors.

FUTURE WORK

ParaGraph is a reasonably mature 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. It would be nice to have more explicit control over simulation speed. The contents of many 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 1,024 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 that have thousands of processors, we must either devise new displays that scale up to this level, or adapt the existing displays by aggregating or selecting information. For example, the current displays could depict processor clusters or subsets (cross sections).

It is fairly easy to imagine how graphics technology might be adapted to meet the needs of visualizing massively parallel computations, but it is much less obvious how to handle the vast volume of trace data that would result from monitoring thousands of processors. It is already difficult to store and process the trace data collected from long runs even with the modest numbers of processors currently supported by PICL and ParaGraph. To go beyond the present level will almost certainly require some degree of behavior abstraction, 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 ParaGraph's displays 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 the monitoring facility will produce data that can be visually displayed in a meaningful way.

Source code for ParaGraph, as well as sample trace files for demonstrating its use, are available for free over Internet's Netlib software-distribution service.[3] To receive detailed instructions for obtaining ParaGraph, send an electronic mail message to netlib@ornl.gov containing the text "send index from paragraph." ♦

ACKNOWLEDGMENTS

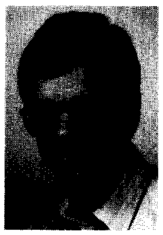
The detailed implementation of ParaGraph has been done almost entirely by undergraduate students on research internships at Oak Ridge National Laboratory. The software's overall structure and the conceptual designs of the individual displays were developed by Heath. The bulk of the programming was done by Etheridge while she was an undergraduate student, first at Roanoke College and later at the University of Tennessee.

Loretta Auvil, then an undergraduate at Alderson Broaddus College, developed the Hypercube display, and Michelle Hribar, then an undergraduate at 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 X Windows, and within a single term each was contributing to the sophisticated software described here. ParaGraph's development 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, US Energy Dept. under contract DE-AC05-84OR21400 with Martin Marietta Energy Systems.

REFERENCES

1. G.A. Geist et al., *PICL: A Portable Instrumented Communication Library, C Reference Manual,* Tech. Report ORNL/TM-11130, Oak Ridge Nat'l Lab., Oak Ridge, Tenn., 1990.
2. M.T. Heath and J.A. Etheridge, *Visualizing Performance of Parallel Programs,* Tech. Report ORNL/TM-11813, Oak Ridge Nat'l Lab., Oak Ridge, Tenn., 1991.
3. J.J. Dongarra and E. Grosse. "Distribution of Mathematical Software via Electronic Mail," *Comm. ACM,* May 1987, pp. 403-407.
4. M.T. Heath, E. Ng, and B.W. Peyton, "Parallel Algorithms for Sparse Linear Systems," *SIAM Review,* Sept. 1991, pp. 420-460.
5. H.L. Gantt, "Organizing for Work," *Industrial Management,* Aug. 1919, pp. 89-93.
6. K. Kolence and P. Kiviat, "Software Unit Profiles and Kiviat Figures," *Performance Evaluation Rev.,* Sept. 1973, pp. 2-12.
7. M.F. Morris, "Kiviat Graphs: Conventions and Figures of Merit," *Performance Evaluation Rev.,* Oct. 1974, pp. 2-8.
8. L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System, *Comm. ACM,* July 1978, pp. 558-565.
9. E.R. Tufte, *The Visual Display of Quantitative Information,* Graphics Press, Cheshire, Conn., 1983.



Michael T. Heath is a computer science professor and research scientist at the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign. He was a senior research-staff member and computer science group leader in the Mathematical Sciences Section at Oak Ridge National Laboratory. His research interests are in large-scale scientific computing on parallel computers, numerical linear algebra, and performance visualization.

Heath received a PhD in computer science from Stanford University.

Jennifer A. Etheridge is a technical research associate in the mathematical sciences section at Oak Ridge National Laboratory. Her current interests are in computer graphics and visualization.

Etheridge received a BS in mathematics from the University of Tennessee at Knoxville.

Address questions about this article to Heath at the National Center for Supercomputing Applications, 4157 Beckman Institute, University of Illinois, 405 Mathews Ave., Urbana, IL 61801-2300; Internet heath@ncsa.uiuc.edu.