Report No. UIUCDCS-R-88-1465

UILU-ENG-88-1771

TAPESTRY
Technical Report No. TTR88-8

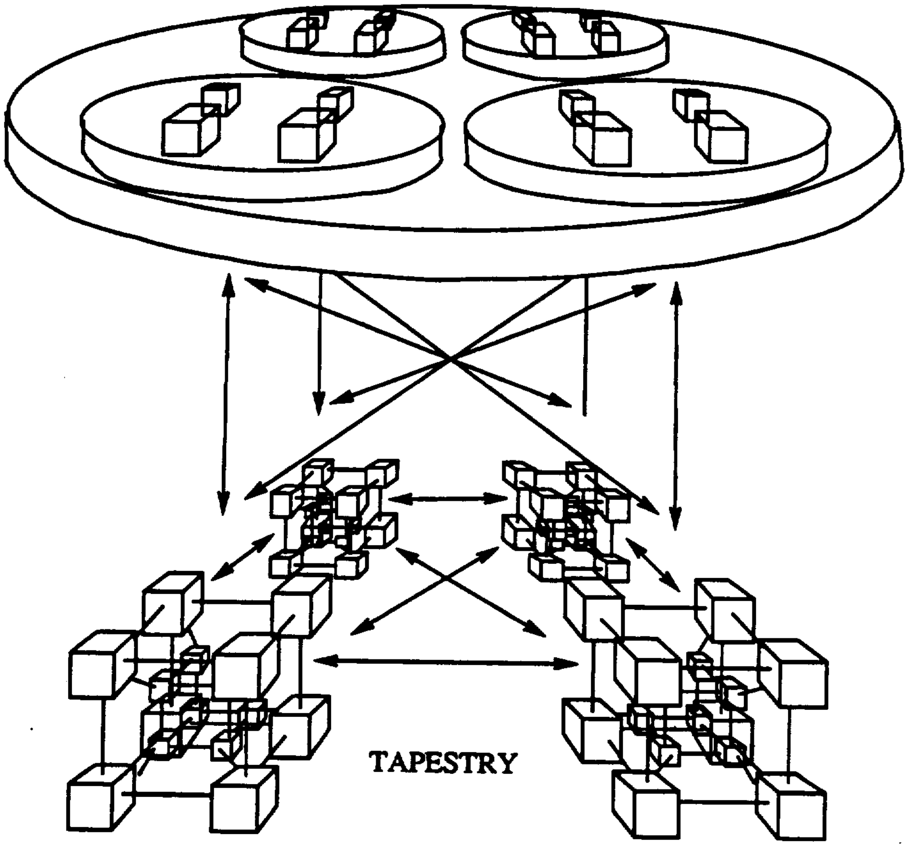
Principal Investigators Roy Campbell
and Daniel Reed

Visualizing Parallel Computer System Performance

by

Allen D. Malony and Daniel A. Reed

September 31, 1988



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS 61801-2987

Visualizing Parallel Computer System Performance

Allen D. Malony [*]   Daniel A. Reed [†]
Department of Computer Science
Center for Supercomputing Research and Development
University of Illinois
Urbana, Illinois 61801

Abstract

Parallel computer systems are among the most complex of man's creations, making satisfactory performance characterisation difficult. Despite this complexity, there are strong, indeed, almost irresistible, incentives to quantify parallel system performance using a single metric. The fallacy lies in succumbing to such temptations. A complete performance characterisation requires not only an analysis of the system's constituent levels, it also requires both *static* and *dynamic* characterisations. Static or average behavior analysis may mask transients that dramatically alter system performance.

Although the human visual system is remarkedly adept at interpreting and identifying anomalies in false color data, the importance of dynamic, visual scientific data presentation has only recently been recognised. Large, complex parallel systems pose equally vexing *performance interpretation* problems. Data from hardware and software performance monitors must be presented in ways that emphasise important events while eliding irrelevant details. Design approaches and tools for performance visualisation are the subject of this paper.

[*] Supported in part by the National Science Foundation under grants NSF MIP-8410110 and NSF DCR 84-06916, by the Department of Energy under grant DOE DE-FG02-85ER25001, and by the Air Force Office of Scientific Research under grant AFOSR-F49620-86-C-0136 (URI).

[†] Supported in part by the National Science Foundation under grants NSF ANTI TAPESTRY 1-5-30035, NSF DCR 84-17948, and NSF CCR86-57696, by the National Aeronautics and Space Administration under NASA Contract Number NAG-1-613, and by the Air Force Office of Scientific Research under grant AFOSR-F49620-86-C-0136 (URI).

The purpose of computing is insight, not numbers.

Richard Hamming

1 Introduction

The appearance of any new computer system raises many questions about its performance, both in absolute terms and in comparison to other machines of its class; parallel computer systems are no exception. Unfortunately, parallel computer systems are among the most complex of man's creations, making satisfactory performance characterization difficult. Despite this complexity, there are strong, indeed, almost irresistible, incentives to quantify parallel system performance using a single metric. The fallacy lies in succumbing to such temptations. Just as it now is widely recognized that human intelligence is not subsumed by the spatial and verbal abilities measured by standard intelligence tests, complete characterization of parallel computer system performance encompasses more than operations executed per second.

Peak performance ratings in MIPS (millions of instructions per second) or MFLOPS (millions of floating point operations per second) obscure the importance of interacting *performance levels* and *dynamic equilibrium*. Repeated studies have shown that a system's performance is maximized when the components are balanced (i.e., there is no single system bottleneck) [5]. As an example, optimizing the performance of message passing systems [17] requires a judicious combination of node computation speed, message transmission latency, and operating system software. High speed processors connected by high latency communication links restrict the classes of algorithms that can be efficiently supported.

A complete performance characterization requires not only an analysis of the system's constituent levels, it also requires both *static* and *dynamic* characterizations. Static or average behavior analysis may mask transients that dramatically alter system performance. By analogy, biological researchers have long recognized the importance of both *in vitro* and *in vivo* measurements. Laboratory measurements of isolated cells or biological molecules often differ from similar measurements in natural environments.

The history of virtual memory research offers a classic example of transient behavior and its importance. The slow drift model [4] predicted that program reference locality changed slowly. Later, more detailed measurements showed that reference localities change swiftly and catastrophically. Most page faults and associated overhead occur in small time intervals, and a phase-transition model more accurately reflects observed behavior.

Performance measurements of high-speed computing systems can quickly generate vast quantities of numerical data. Indeed, recognition of the importance of virtual memory phase transitions was hampered by the volume of data

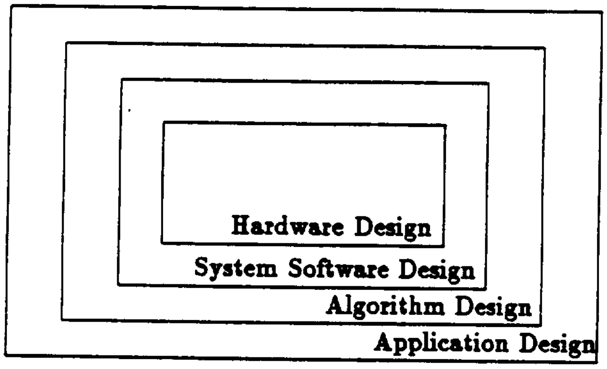


Figure 1: Performance Levels

generated during simulation and measurement; post-measurement data compression yielded page fault rates, a static performance measure. However, phases and transitions can be seen only by examining significant portions of the reference trace; this is best done via dynamic graphic displays.

Although the human visual system is remarkedly adept at interpreting and identifying anomalies in false color data, the importance of visual scientific data presentation has only recently been recognised [7]. Large, complex parallel systems pose equally vexing *performance interpretation* problems. Data from hardware and software performance monitors must be presented in ways that emphasise important events while eliding irrelevant details.

In collaboration with the Center for Supercomputing Research and Development at the University of Illinois, we are developing a suite of performance visualization tools. These tools and our design approach are the subject of this paper. In §2 we examine the importance of performance levels and formalize the empirical performance evaluation process. In §3 we discuss HyperView, a prototype that dynamically displays performance data obtained from hardware measurement and simulation of message passing systems. Techniques for visualizing application performance are the subject of §4; linear programming [6] [19] is used as a test problem. Finally, §5 summarises our experience and development plans.

2 Experimental Performance Analysis

As Figure 1 illustrates, there are four levels in the hierarchy of performance measurements. The answer to the oft-asked question, "How fast is it?" depends on the intended use of the performance data. At the lowest level lies the performance of the hardware design. Determining this performance provides

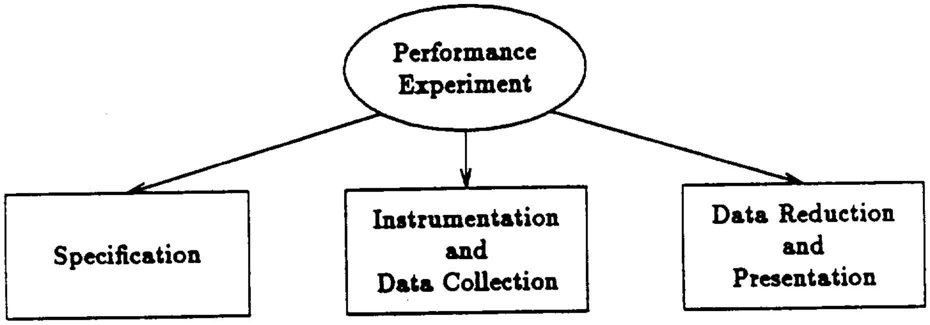

Performance Experiment
Specification
Instrumentation and Data Collection
Data Reduction and Presentation

Figure 2: Performance Analysis Phases

both a design validation and directives for system software design. Only by understanding the strengths and weaknesses of the hardware can system software designers develop an implementation and user interface that maximizes the fraction of the raw hardware performance available to the end user. As an example, consider a hypothetical hypercube operating system that provides dynamic task migration to balance workloads. To meet these goals, it must be possible to rapidly transmit small status messages. It is fruitless to design such a system if the underlying hardware provides only high-latency message transmission. Given some characterization of the balance between processing power and interprocessor communication resulting from the system software, users can develop algorithms that are best suited to the parallel system. Finally, the best mix of key algorithms will maximise the performance of user applications.

Regardless of the system level, performance characterisation requires specification of the desired measurements, instrumentation and data collection mechanisms, and data reduction and display; see Figure 2.[1] Although it is clear that a parallel computer system is a *gestalt* whose performance is inextricably tied to the performance of its constituent hardware and software levels, it is less clear that performance instrumentation and data collection techniques for one level, or even one system, are rarely applicable to other systems or other levels. As an example, Table 1 shows a subset of the important performance measurements for three levels — hardware, system software, and algorithm and three systems — the Cray X/MP, the University of Illinois Cedar system [13], and the Intel iPSC hypercube [16].

The diversity of underlying technology and system architecture makes it impossible to develop a single set of performance instrumentation techniques. Memory bank conflicts on the Cray X/MP have no analog on the distributed memory Intel iPSC. Moreover, the event time scales differ by six orders of

[1] By analogy with news reporting, "What do I want?" "How do I get it?" and "How can I see it?"

Level	*Cray X/MP*	*Illinois Cedar*	*Intel iPSC*
Hardware	vector startup memory conflicts	network contention vector/cache interaction	processor speed communication latency and bandwidth
Software	compiler	compiler	OS support
Algorithm	vectorization	shared memory access	communication pattern

Table 1: Performance Level Comparison

magnitude. Similarly, the shared memory access patterns of Cedar application algorithms may cause interconnection network conflicts, but these patterns are not predictors of performance degradation due to network contention. Although it is impossible to develop a single performance instrumentation mechanism applicable to all levels, *mechanisms* for specification of noteworthy performance events and their presentation are largely system independent.[2]

At all performance levels there exists a minimal set of required events (e.g., counts and times). Capture of these events should be enabled by signals to a hardware monitor, operating system calls, or flags to a compiler preprocessor. In addition to standard events, certain others must be enabled selectively, either to minimize the performance perturbations of instrumentation or to reduce the data volume to tractable levels. Ideally, a standard user interface should permit event specification regardless of the event type or the performance level.

Despite the diverse instrumentation events of differing levels and systems, the performance measures can be presented using a small number of display types (e.g., bar and strip charts, three-dimensional plots, and state transition diagrams). These graphical displays are the subject of the remainder of this paper.

3 HyperView: A Hypercube Visualization Tool

In collaboration with the Center for Supercomputing Research and Development, we have designed and implemented HyperView, a prototype performance visualization tool for distributed memory parallel processors configured as hypercubes. HyperView dynamically displays architectural and system activity via a multiplicity of system views. Detailed performance measurements also are

[2] The *events* vary but the specification and display *mechanism* need not.

provided via standard statistical displays.

HyperView was inspired by *Seecube* [3], a hypercube visualization system built for the SunView[3] window environment. Although many of the HyperView displays were borrowed from *Seecube*, the implementation is based on the X window environment [18] and the user interface libraries provided by the Faust parallel programming environment being developed at the University of Illinois Center for Supercomputing Research and Development [10] [11] [12]. The portability provided by X permits use of HyperView in a variety of workstation environments. Because X supports a client-server paradigm, the data analysis and display portions of HyperView are decoupled, potentially executing on different systems. This decoupling not only makes the visualization portions independent of message passing hardware and system software, it also is crucial if real-time performance display and dynamic system reconfiguration are to be supported. Thus, HyperView contains three cooperating modules — data capture, state analysis, and visualization.

3.1 Data Capture

The HyperView visualization component accepts event traces generated by the processors of a message passing system. Because the data capture is decoupled from visualization, the event trace can be generated via simulation, permitting study of new message passing architectures, or from program execution. At present, the HyperView visualization is driven by data obtained from simulation of communication hardware for different message passing paradigms [9], including store-and-forward message switching, circuit switching, staged circuit switching, and wormhole circuit switching. Our experience has shown that visual comparison of system dynamics quickly reveals differences in communication paradigms.

When an event is detected by the performance instrumentation, an event identifier, a timestamp, and any additional event data are written to a trace buffer. For our message passing simulations, we instrumented the simulator to record the following information about message events at each hypercube node. The following events suffice to display message passing activity for fixed-path and adaptive variants of both circuit and store-and-forward message switching.

```
i <time> <nodes> <string>              Initial message
m <time> <msg id> <from> <to> <size>   Create message
q <time> <msg id> <at>                 Enqueue message
Q <time> <msg id> <at>                 Dequeue message
e <time> <msg id> <at>                 Circuit establishment
v <time> <msg id> <from> <to> <link>   Visit node via link
t <time> <msg id>                      Begin message transmission
T <time> <msg id>                      End message transmission
```

[3] SunView is a trademark of Sun Microsystems.

```
V <time> <msg id> <from> <to> <link> Delete link between nodes
E <time> <msg id> <from> <to>        Circuit termination
M <time> <msg id>                    Message delivery
```

For both circuit and packet switching, messages may require several transmissions and may cross multiple communication links to reach their final destination. Hence, the events recorded by a single hypercube node are insufficient to reconstruct the history of a message. Thus, the *from* and *to* arguments in the message creation event represent the point of message origination and the final destination. Because we are studying routing paradigms that can choose one of many paths to the destination node, *link* traversal information must be saved to reconstruct the routing path.

Although the instrumentation events just described suffice to display communication traffic and queueing delays, other events are needed to display system software and application behavior. Thus, we are developing software and hardware instrumentation for an Intel iPSC/2 hypercube that will permit near real-time data capture of user, system and hardware events, including support for local event buffering, global timestamp synchronisation, and trace processing; see §5 for additional details.

3.2 State Analysis

In a distributed memory parallel system such as a hypercube, each node must record events based only on local knowledge; the absence of global memory precludes data sharing with the granularity necessary to dynamically maintain a consistent, global state. Moreover, the nodes of many distributed memory systems are individually clocked, the clocks often are not synchronized, and the clocks may tick at different rates. Thus, the event trace at best defines a partial time order, and the timestamps may be logically inconsistent with the logical order of events.

To recover global state during trace analysis, the trace timestamps must be reconciled and enough event data must be saved to correlate distributed events. The analysis requires interpreting each event in sequence and incrementally modifying the current system state; for complete details see [3].

3.3 Performance Visualization

The `HyperView` user interface permits simultaneous display of the dynamic system state via a variety of differing *views*. Each view emphasises certain system aspects (e.g., the network topology, the multiplicity of partially overlapping paths from a source to a destination node, or queues of waiting messages). Each view provides a different insight; collectively they convey system dynamics.

Although Hamming's dictum applies, numbers are often necessary and important. In addition to graphical displays, `HyperView` provides statistical dis-

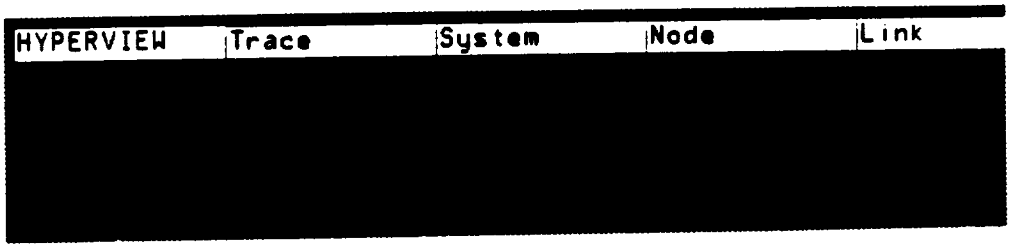


Figure 3: HyperView Top-Level Display

plays at both macroscopic levels (e.g., number of messages transmitted) and microscopic levels (e.g., link utilization). Finally, HyperView permits *selective* display of message traffic and statistics, permitting the performance analyst to isolate anomalous behavior for further study.

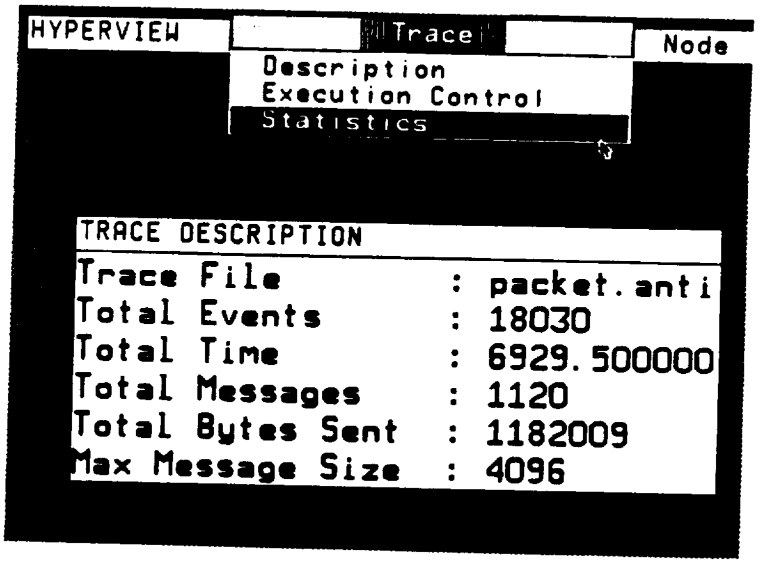
Because HyperView is a *dynamic* performance visualization system, much is lost in description of static, monochrome images. Despite these limitations, we discuss HyperView as a performance analyst might encounter it, beginning with the top-level user interface shown in Figure 3. User menus are shown at the top of the screen. Pulling down the **Trace** menu lists the *Description*, *Execution Control*, and *Statistics* items shown in Figure 4.

3.3.1 Trace Description

In the *Trace Description* window, a performance analyst can select, by clicking the mouse on the *Trace File* item, a trace file that contains the event information captured during system execution. A dialogue box (not shown) will pop up requesting the user to enter the trace file name. After reading the trace file, HyperView begins the state analysis needed to recover the time varying global state of the message passing system. During state analysis, HyperView computes the number of events, messages and bytes transmitted. Throughout the visualization session, these statistics can be viewed by selecting the *Trace Description* menu.

3.3.2 Execution Control

As the name suggests, the *Execution Control* window controls updates to the graphical and statistical displays. During state analysis, HyperView identifies



TRACE DESCRIPTION

Trace File : packet.anti
Total Events : 18030
Total Time : 6929.500000
Total Messages : 1120
Total Bytes Sent : 1182009
Max Message Size : 4096

Figure 4: HyperView Trace Window

a series of globally consistent display points. During updates of the trace display, HyperView moves between these display points. The current point can be marked by a position in time and/or location in the event trace. Thus, the performance analyst can select a current display time either by clicking the mouse on *Current Frame Time* and entering a time, or by clicking the mouse somewhere within the *Frame Time* slider bar. Event trace positions are selected similarly. When a new time or event is selected, HyperView moves to the next consistent system state and its corresponding display. Because the event trace is processed *a posteriori*, the performance analyst can move both forward and backward in time.[4]

A *Frame* in HyperView corresponds to a displayed system state. The user can change three aspects of frame display — mode, rate, and state differential. Frames can be displayed either in single-step mode or continuously. If the mouse is clicked on the *SINGLE STEP* button, the user must explicitly request display of the next frame. Conversely, *CONTINUOUS* mode automatically advances to the next frame specified by the frame rate and differential controls.

Via the *Frame Rate* control, the performance analyst can adjust the interval between display of new frames. The third aspect of frame control is the change in system state, in events or time, between successive, displayed frames. This state difference is the minimum of the specified number of *Events per Frame* and

[4] §5 discusses both the advantages and disadvantages of time independent browsing.

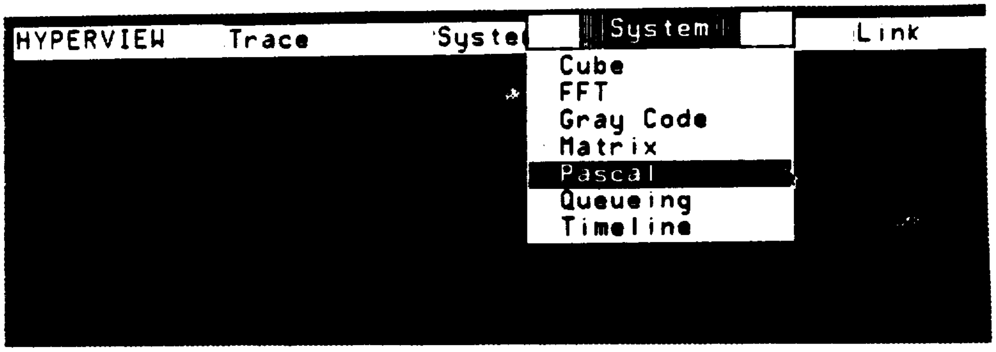


Figure 5: HyperView System Menu

the number of *Clock Ticks per Frame*. By adjusting the display mode, frame rate, and state differential, the performance analyst can study gross behavior, examining a small subset of all states, or examine the trace event by event.

3.3.3 Global Statistics

The *Statistics* window shows the global system state, both cumulative message statistics and *current* node and link activity. Because the performance analyst can browse the trace, cumulative statistics are not monotonic — they reflect performance data relative to the current trace state.

3.3.4 System Displays

Figure 5 shows the menu of dynamic system views provided by HyperView. Figures 6 and 7 show the *CUBE*, *FFT*, *PASCAL*, and *QUEUE* views. Each display gives a different view of the hypercube that shows current system activity as highlighted nodes and links. Each view emphasises certain system aspects (e.g., the network topology, the multiplicity of partially overlapping paths from a source to a destination node, or queues of waiting messages). For example, the *CUBE* view is the "natural" multi-dimensional representation of a hypercube. In contrast, the *FFT* view emphasises message routing paths. The *GRAY CODE* view, not shown, emphasises subcube communication — communication links connecting the two $D-1$-dimensional subcubes of a D-dimensional hypercube appear as parallel lines [3]. The *PASCAL* view reflects the logarithmic combining (e.g., global minimisation) when logical trees are embedded in the hypercube topology [19]. The *QUEUE* view in Figure 7 shows the instantaneous state of the message queues at each node. Each message awaiting transmission is shown as a small box. Communication transients appear as bursts of enqueued messages. Similarly, the effects of differing communication paradigms (e.g., store-and-forward message switching and circuit switching) appear as differences in mean queue size.

In all views, colors emphasize activity — links change color when messages are sent, nodes flash when processing messages. Moreover, each system view supports pull-down menus for inquiries about nodes, links, messages, and circuits. In each topological view (i.e., *CUBE*, *FFT*, and *PASCAL*), unwanted detail can be elided via the *Node* and *Link* menus. For example, display of any combination of transmitting, active, or receiving nodes and links can be disabled. Figure 8 shows the *Link* menu; the *Node* menu is similar. *All*, *Active* and *Transmitting* select the displayed link states.

3.3.5 Message and Circuit Tracking

In addition to elision of unwanted node and link details, HyperView supports *message tracking* and *circuit tracking.*[5] After identifying source and destination nodes, *only* those messages in transit between the specified nodes are displayed. Figure 9 illustrates message and circuit tracking in a system with circuit switched communication. In the figure, nodes 0 and 20 have been selected for circuit tracking and message tracking, respectively. Node 0 is transmitting a message to node 15 along the path shown. The intermediate nodes on the path are not active because only circuit connections have been established there. Concurrently, node 0 is sending a message to node 20, and node 20 is sending a message to node 29.

Message and circuit tracking have proven invaluable when comparing communication paradigms. By eliding extraneous detail, the dynamics of circuit establishment in both fixed and adaptive routing paradigms can be easily compared.

4 Application Performance Displays

Performance visualization at both the hardware and system software levels provides important insight for *system* design and analysis. And because system performance is manifest in the application software executed during performance analysis, system level performance visualization indirectly provides application performance insight. However, insight from visualization of system performance must be coupled with insight from application performance visualization to understand the interactions of different performance levels. To illustrate these interactions and the importance of integrated visualization tools, we use a parallel implementation of the simplex linear optimization algorithm [6] [19] as an example. Like many parallel algorithms, the performance of the simplex

[5] The choice and semantics depend on the underlying hardware communication paradigm.

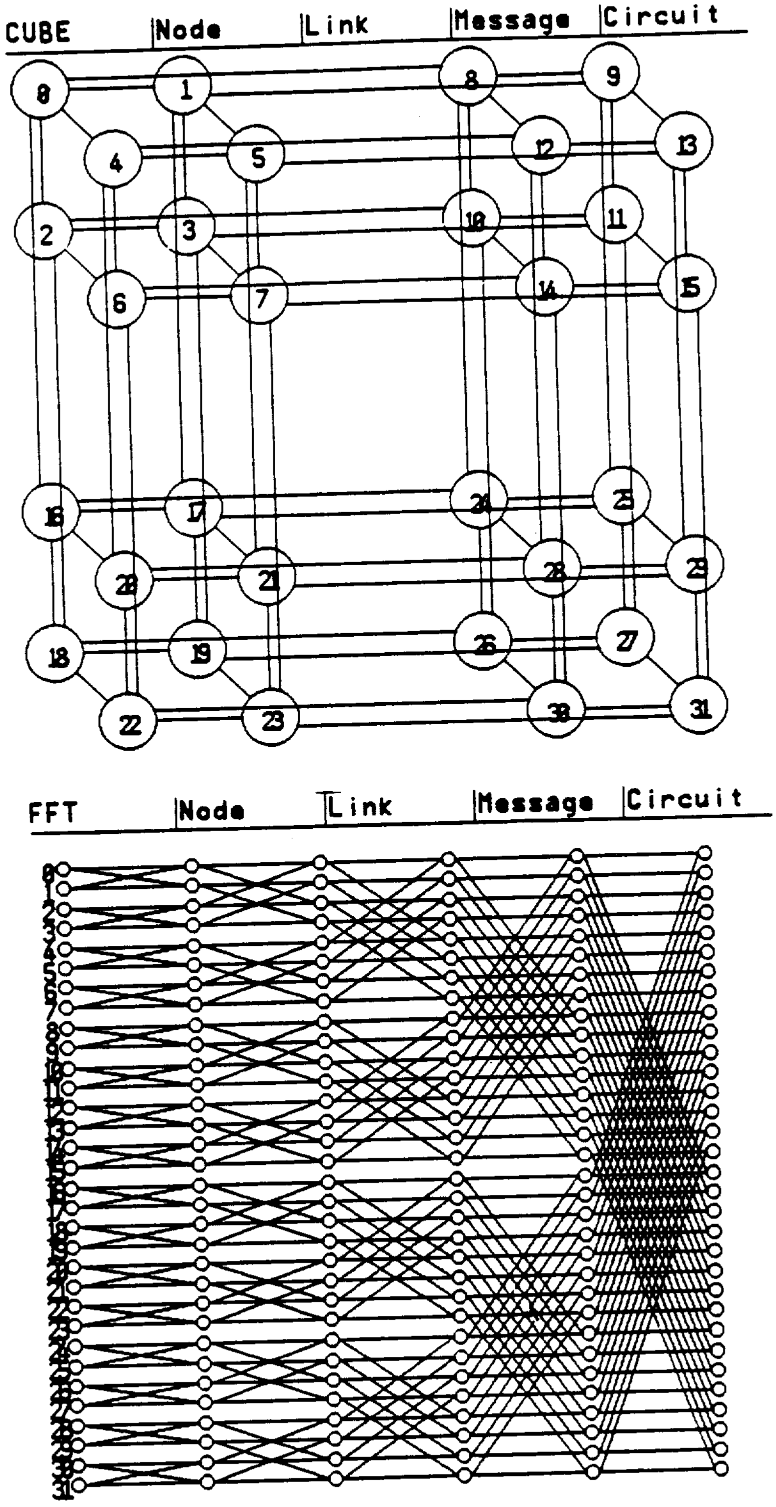


Figure 6: *CUBE* and *FFT* Displays

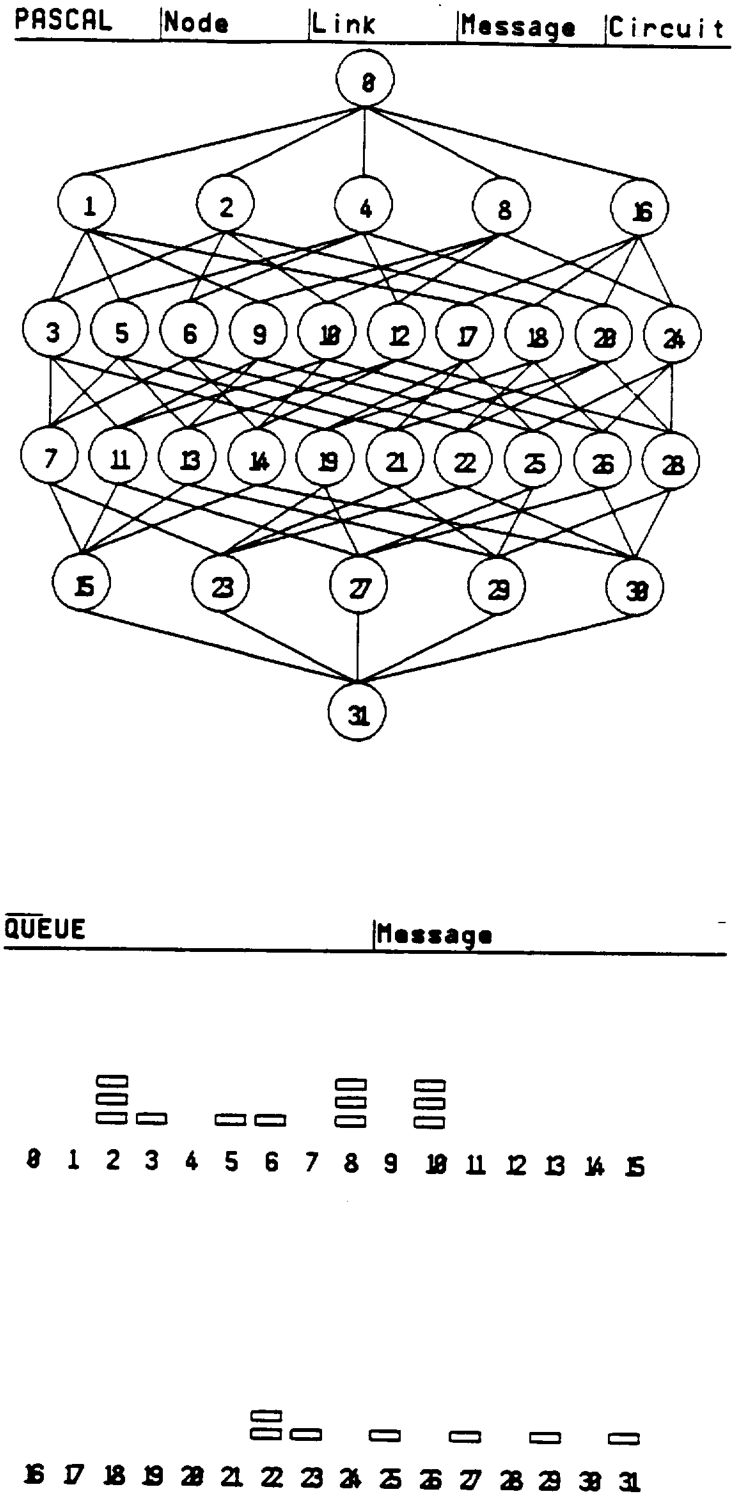


Figure 7: *PASCAL* and *QUEUE* Displays

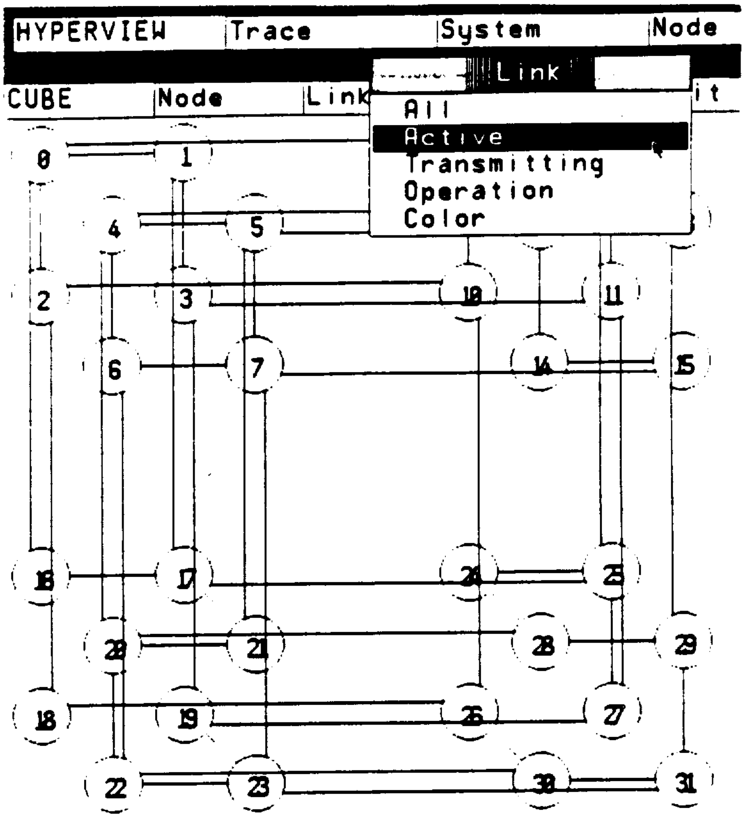


Figure 8: *Link* **Menu**

method varies greatly with input data, and these variations are intimately related to both the algorithm and its interaction with the hardware and system software.

4.1 Linear Optimization: An Example

Large, sparse, linear systems of equations arise frequently when constructing mathematical models of natural phenomena. Most often, these linear systems are fully constrained and can be solved via a variety of direct or iterative techniques. However, one important problem class requires solutions to *underconstrained* linear systems that maximise some objective function. These linear optimization problems often contain hundreds of equations with thousands of variables. Mathematically, this can be stated as:

$$\begin{aligned} \textit{Minimize}: &\quad c^T x \\ \textit{Subject to}: &\quad Ax = b \\ &\quad b \geq 0 \\ &\quad x \geq 0 \end{aligned}$$

Here, c^T is an n vector of variable coefficients that defines the objective function (i.e., the function being minimized). For a maximization problem, the negative

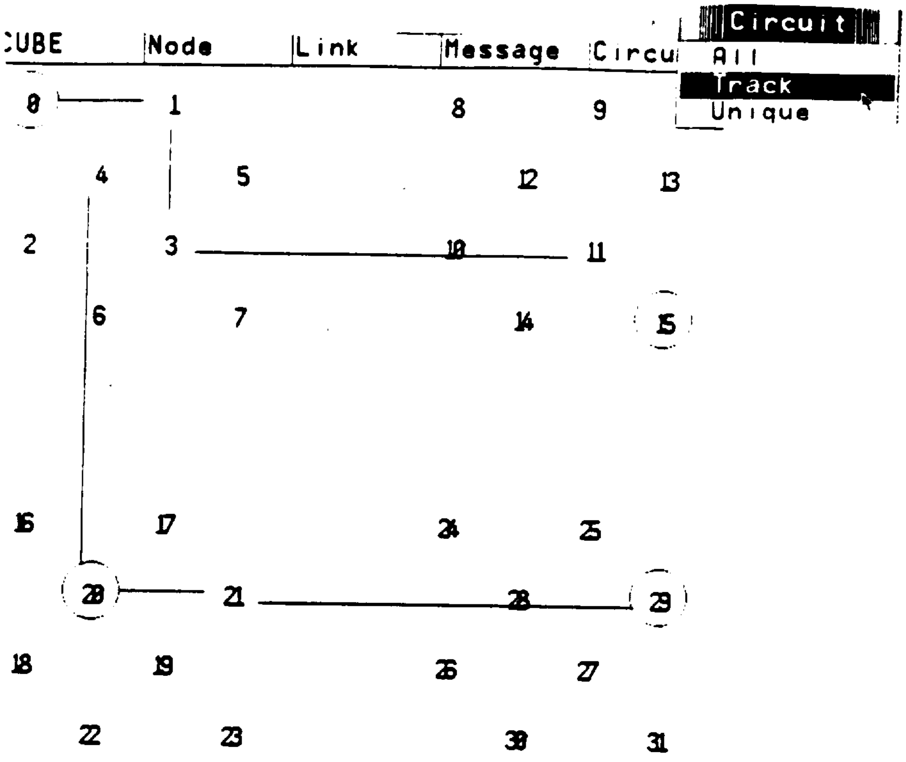


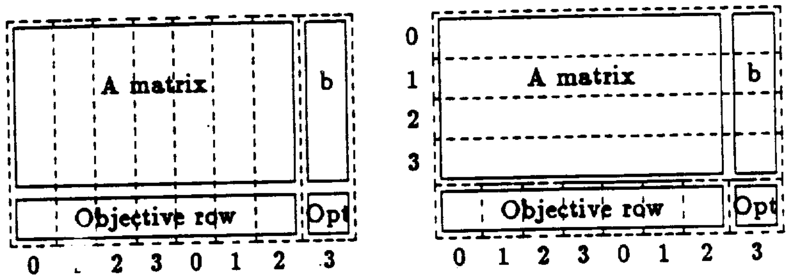
Figure 9: Message and Circuit Tracking

of the objective function can be minimized. The objective function can thus be viewed as a cost function, where the goal is to minimize total cost. The $m \times n$ linear system $Ax = b$ defines the linear constraints on the objective function x. Each of the m rows of the matrix A defines a constraint on the n variables of the objective function.

The optimization problem arises because the linear system $Ax = b$ is underconstrained (i.e., m is smaller than n, and the matrix A contains many more columns (variables) than rows (constraints)).[6] Consequently, there are many possible x vectors that satisfy the system $Ax = b$. A fundamental theorem of linear programming states that an optimal solution, if it exists, occurs when $n - m$ elements of x are zero (i.e., when there are precisely m non-zero elements of x). This corresponds to the solution of an $m \times m$ linear system, the *basis*, obtained by selecting m of the n columns of the matrix A.

Clearly, exhaustive solution of the $\binom{n}{m}$ possible linear systems is not feasible. The simplex method is a search algorithm that decreases the value of the objective function at each iteration by selecting a non-zero element of x, a so-called *basic* variable, and replacing the corresponding column of A with another column. The simplex method provides a systematic way of moving from one basic feasible solution (i.e., one satisfying the constraints) to another. This systematic movement, called *pivoting*, must

[6] See Figure 11 for an example.



(a) Column partitioning (b) Row partitioning

Figure 10: Data Placement for Simplex Row and Column Partitions

- identify a new basis column that decreases the objective (cost) function value,
- identify the column to remove from the basis that maximizes the decrease in the objective function value while still satisfying the constraints, and
- replace the old basis column with the new one.

These transformations are realized by standard techniques from numerical linear algebra (i.e., Gauss-Jordan elimination).

4.2 Parallel Simplex Variants

In message passing architectures, interprocessor communication is much more expensive than local memory access. Hence, many algorithm implementation details are constrained given the mapping of data to processors. The simplex algorithm shares similar characteristics with solution of linear systems, matrix multiplication, and other common matrix operations. Previous work on distributed matrix algorithms has advocated row or column partitioning of matrices [1] [8] [15]. We have considered similar schemes for distributing the matrix of constraints across the nodes of a hypercube [19].

In the column partitioned method, shown in Figure 10, complete columns are divided equally among the processors. To identify the column to enter the basis, each hypercube node must first find the local minimum of the objective values for those columns in its local memory, then cooperate with other nodes to identify and distribute the identity of the column containing the minimum objective value. Conversely, the *single node* containing the pivot column must identify the column to leave the basis. Thus, partitioning the matrix by columns creates both parallel and sequential computation phases.

In the row partitioned strategy, complete rows of the matrix are divided equally among the processors. As Table 2 shows, this approach also creates

Partition	*Entering Basis Column*	*Departing Basis Column*	*Gauss-Jordan Elimination*
Column	Parallel computation Global minimization	Sequential computation	Column global send Parallel computation
Row	Parallel computation Global minimization	Parallel computation Global minimization	Row global send Parallel computation

Table 2: Hypercube Simplex Variations

both parallel and sequential computation phases.[7] Despite the similarities suggested by Table 2, the performance of simplex algorithms based on row and column data partitions can be strikingly different. Why? Distributed linear systems solvers process $n \times n$ matrices. The constraint matrices processed by the simplex method contain many fewer rows than columns. Moreover, the ratio of the number of rows to columns can vary dramatically. This variance, coupled with the differences in matrix sparsity, is manifest in the relative costs of communication, sequential computation, and parallel computation. Hence, neither row nor column partitioning is uniformly superior. To understand the dynamics of algorithm interaction with matrix structure, application visualization tools are necessary.

4.3 Simplex Performance Visualization

Earlier study [19] suggested that, despite variations in matrix structure, row partitioned simplex implementations often yielded better performance. However, counterexamples exist; Figure 11 shows the non-zero matrix structure of one such problem. Although the 7:1 ratio of columns to rows suggests the reason that column partitioning is preferable, the details are best grasped via visualization.

Figures 12 and 13 show four views of the number of messages sent between tasks of the row partitioned simplex algorithm on a 16 node Intel iPSC/2 hypercube. Recall that in a D-dimensional hypercube, a node with address n is directly connected only to those other nodes with addresses whose binary expansions differ from n in exactly one bit. Although messages must cross multiple communication links to reach some nodes, the maximum distance between any two nodes is D. When exploring performance at the hardware and system software levels, understanding node connectivity is crucial. However, at the application level, messages are exchanged by tasks, not hypercube nodes. Hence,

[7] In reality, there are many subvariations of both row and column partitioning, and each has differing performance; see [19] for complete details.

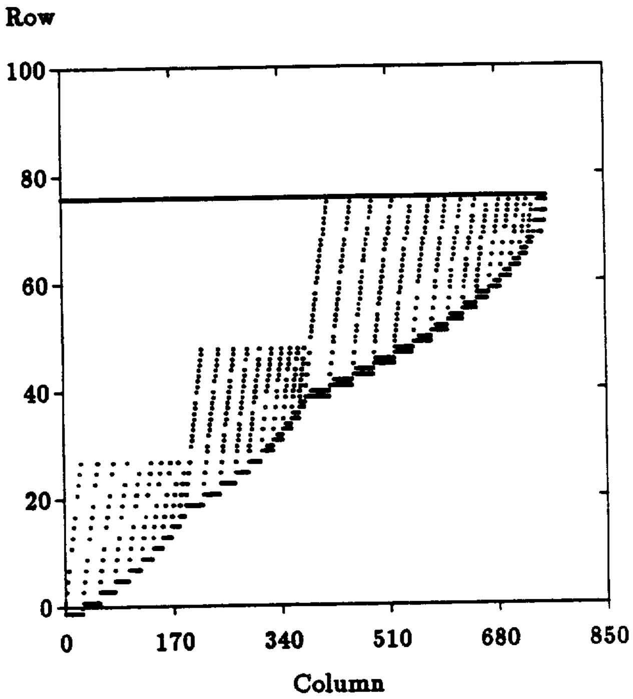


Figure 11: Simplex Benchmark SCSD1

Figure 12 [8] and subsequent figures show the *logical* interaction of tasks, not the physical transfer of data. We emphasize that complete understanding requires performance visualisation at all levels, hardware, system software, algorithm, and application. By separating the levels, the performance contributions of each level are manifest.

In Figure 12 the peaks represent the logarithmic combining necessary to identify global minima. In Figure 13 the logarithmic combining appears as lightly shaded regions in the density view and as clustered contour lines in the contour view. Because task zero is the root of the combining tree, during each simplex iteration it must broadcast the identity of the task containing the global minimum. The identified task then broadcasts the needed row to all other tasks. If the workload were perfectly distributed, each task would broadcast an equal number of times. Excluding messages due to the logarithmic combining, all other variations in communication traffic are attributable to this load

[8] In the 3-dimensional displays, counts greater than thirty were clipped, hence the uniformity.

imbalance. The multiplicity of views reflects our belief that an integrated performance visualization system should permit the performance analyst to select those views that correspond to his or her personal preference and needs.

Figure 14 shows the *volume* of data exchanged between tasks. Comparing Figure 14 with Figure 12 shows that tasks exchanging many messages do *not* exchange a large volume of data. Why? The many messages necessary to realize the combining tree are small; the row broadcasts require fewer, larger messages. The performance ramifications of this bimodal distribution of message sizes can only be understood by examining hardware and system software performance displays. These displays show that message passing systems like the Intel iPSC/2 have large message preparation times relative to communication link bandwidth, penalizing small messages. Hence, message count is the important performance metric, not message volume.

Finally, Figures 15 and 16 show the message count and volume for the column partitioned simplex algorithm. As before, a combining tree is used to identify global minima. However, as Table 2 shows, this global minimization is used only when finding an entering basis column. Because each task contains columns, a sequential computation is used to identify the departing basis column. This reduces the number of small message transmissions at the expense of reduced parallelism. More importantly, however, broadcasting matrix columns is *much* less expensive than the row broadcasts of the row partitioned algorithm. The scales for figures 14 and 16 differ significantly; this is the reason the column partitioned variant is superior for the matrix of Figure 11.

5 Current Research

The hardware and application performance visualizations just described are *ad hoc* and are not integrated. First, HyperView was designed primarily to display hardware performance. As such, it is not easily extensible to display of application performance, nor should it be — display techniques for system and application performance differ. Second, the simplex application visualizations required manual instrumentation of the simplex code and extensive preprocessing before they could be displayed using *Mathematica*, a symbolic manipulation system and mathematician's assistant not intended for this use. HyperView and the simplex application visualization are *facsimiles* or rapid prototypes of what is desired — an integrated performance specification, instrumentation, and visualization system for message passing systems.

Figure 17 illustrates the ideal. This hypothetical system, called Tapestry would weave together elements of the hardware, system software, and application levels. The hardware and system levels, shown at the top of the figure would, like HyperView and *Seecube*, dynamically display internode communication traffic[9] using multiple colors. Dynamic displays would include current

[9] Although the figure shows a four-dimensional hypercube, other views, such as the Pascal

messages, cumulative traffic (either counts or volume), and link utilisation. Via a mouse, the performance analyst also could choose from an extended menu of performance displays for each node and link:

- external input/output (i.e., file accesses),
- processor utilisation,
- context switches,
- system calls,
- memory utilisation,
- memory reference patterns (i.e., reference localities),
- virtual memory paging activity, and
- message counts and volume by destination.

Each of these could be displayed in a variety of formats (e.g., perspective, histogram, strip chart, contour, or density).

The application performance level, illustrated at the bottom of Figure 17 would display the *logical* graph of the intertask communication pattern, not the physical graph of the underlying interconnection network. By dragging graph nodes and edges with a mouse, the topological orientation of the graph could be modified to reflect the performance analyst's preferences. The application performance level, like the hardware level, would include dynamic displays of message traffic on the parallel program graph and via perspective, density, and contour plots. In addition, pull-down menus for tasks would include:

- message counts and volume by destination,
- delays for message transmission or receipt,
- dynamic procedure call graphs, and
- execution profiles.

Finally, the visualisation system would permit correlation of system and application performance.

The astute reader will have realised that near real-time processing and display of such detailed performance data (e.g., memory reference patterns) implies prodigious, indeed unrealistic, computing, storage, and display requirements. Below, we discuss those features we believe are necessary to achieve the goal of an integrated performance visualisation system.

triangle, Gray code, or FFT would be supported also.

5.1 Instrumentation and Visualization

The major limitation of HyperView, *Seecube*, and the simplex application visualization is the absence of near real-time behavior. *A posteriori* examination of performance data means that *all* data of potential interest must be captured *a priori*. Despite the consequent increase in storage requirements, this is sometimes desirable — it permits performance data browsing across the entire interval of execution, and it permits data capture at a level of detail incompatible with near real-time processing. However, *a posteriori* examination also precludes dynamic system or application reconfiguration based on observed performance. Real-time display of even a portion of the captured data would permit the performance analyst to selectively enable and disable performance instrumentation based on observed behavior, reducing the storage requirements.

Despite the manifest advantages of interactive performance instrumentation and display, on most message passing systems, including the Intel iPSC/2 hypercube, *a posteriori* data display is unavoidable because there is insufficient communication bandwidth to transmit performance data to an external host without distorting the performance being measured. Moreover, the limited memory at each node constrains the volume of performance data that can be buffered for subsequent transmission. Clearly, hardware support for performance data recording is crucial, and we assume its existence. However, detailed discussion of hardware designs for performance instrumentation is beyond the scope of this paper. See [2] for a discussion of the hardware requirements for performance instrumentation.

A visualization system must be evolutionary, adapting to the changing demands of hardware, system software, applications, and users. Thus, the implementation must be extensible, permitting addition of new display formats and performance metrics, and portable, permitting use with a variety of systems. These twin goals, extensibility and portability, suggest a modular, object-oriented design that separates interface from implementation. Using the X client-server paradigm [18] would provide portability and insure future extensibility based on an emerging standard for window systems. However, X alone provides neither the necessary abstractions (e.g., hierarchical performance displays) nor the rapid prototyping support; object-oriented window libraries such as InterViews [14] are needed.

5.2 Current Status

Based on the lessons learned with HyperView, we are implementing an initial version of Tapestry for the Intel iPSC/2 using X and InterViews. Initially, software instrumentation of NX/2, the iPSC/2 operating system, will provide data on system performance; a hardware monitor will be added later. Application performance data are captured by instrumenting application and system libraries, by modifying a compiler to automatically instrument application code, and by manually inserting instrumentation directives in application code.

Acknowledgments

Craig Stunkel designed the simplex code and conducted the experiments that motivated the application performance displays. Steven Wolfram's *Mathematica* [20] was used to generate Figures 12 — 16. We are grateful for use of prototype versions of this software; the ability to quickly generate many data views has been immensely useful. Finally, *Seecube* [3] provided the inspiration for HyperView; many of the ideas and displays were borrowed from this pioneering work.

References

[1] Aykanat, C., and Ozguner, F. Large Grain Parallel Conjugate Gradient Algorithms on a Hypercube Multiprocessor. In *Proceedings of the 1987 International Conference on Parallel Processing* (St. Charles, IL, Aug. 1987), pp. 641–644.

[2] Carpenter, R. J. Report of the Distributed Memory Discussion Group. In *These Proceedings*.

[3] Couch, A. L. *Graphical Representations of Program Performance on Hypercube Message-Passing Multiprocessors*. PhD thesis, Tufts University, Department of Computer Science, Apr. 1988.

[4] Denning, P. J. Working Sets Past and Present. *IEEE Transaction on Software Engineering SE-6*, 1 (Jan. 1980), 64–84.

[5] Denning, P. J., and Buzen, J. P. The Operational Analysis of Queueing Network Models. *ACM Computing Surveys 10*, 3 (Sep. 1978), 225–261.

[6] Foulds, L. R. *Optimization Techniques: An Introduction*. Springer-Verlag, New York, NY, 1981.

[7] Frenkel, K. A. The Art and Science of Visualizing Data. *Communications of the ACM 21*, 2 (Feb. 1988), 110–121.

[8] Geist, G. A., and Heath, M. T. Matrix Factorization on a Hypercube Multiprocessor. In *Proceedings of the First Conference on Hypercube Computers and Concurrent Applications* (Knoxville, TN, Aug. 1985), pp. 161–180.

[9] GRUNWALD, D. C., AND REED, D. A. Networks for Parallel Processors: Measurements and Prognostications. In *Proceedings of the Third Conference on Hypercube Computers and Concurrent Applications* (Pasadena, CA, Jan. 1988).

[10] GUARNA, V., GANNON, D., GAUR, Y., AND JABLONOWSKI, D. An Environment for Programming Scientific Applications. In *Proceedings of Supercomputing '88* (Orlando, Florida, Nov. 1988).

[11] GUARNA, V., AND GAUR, Y. A Portable User Interface for a Scientific Programming Environment. In *Proceedings of the Siggraph Symposium on User Interface Software* (Banff, Alberta, Canada, Oct. 1988).

[12] JABLONOWSKI, D., AND GUARNA, V. *A Dynamic Graph Tool and Its Use in an Integrated Programming Environment.* Tech. Rep. CSRD Report No. 746, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, June 1988.

[13] KUCK, D. J., DAVIDSON, E. S., LAWRIE, D. H., AND SAMEH, A. H. Parallel Supercomputing Today and the Cedar Approach. *Science 231* (Feb. 1986).

[14] LINTON, M. A., AND CALDER, P. R. The Design and Implementation of InterViews. In *Proceedings of the USENIX C++ Workshop* (Santa Fe, NM, Nov. 1987), pp. 256–267.

[15] MOLER, C. Matrix Computation on Distributed Memory Multiprocessors. In *Proceedings of the First Conference on Hypercube Computers and Concurrent Applications* (Knoxville, TN, Aug. 1985), pp. 181–195.

[16] RATTNER, J. Concurrent Processing: A New Direction in Scientific Computing. In *Conference Proceedings of the 1985 National Computer Conference* (1985), AFIPS Press, pp. 157–166.

[17] REED, D. A., AND FUJIMOTO, R. M. *Multicomputer Networks: Message-Based Parallel Processing.* The MIT Press, 1987.

[18] SCHEIFLER, R. W., AND GETTYS, J. The X Window System. *ACM Transactions on Graphics 5*, 2 (Apr. 1986), 79–109.

[19] STUNKEL, C. B., AND REED, D. A. Hypercube Implementation of the Simplex Algorithm. In *Proceedings of the Third Conference on Hypercube Computers and Concurrent Applications* (Pasadena, CA, Jan. 1988).

[20] WOLFRAM, S. *Mathematica: A System for Doing Mathematics by Computer.* Addison-Wesley, July 1988.

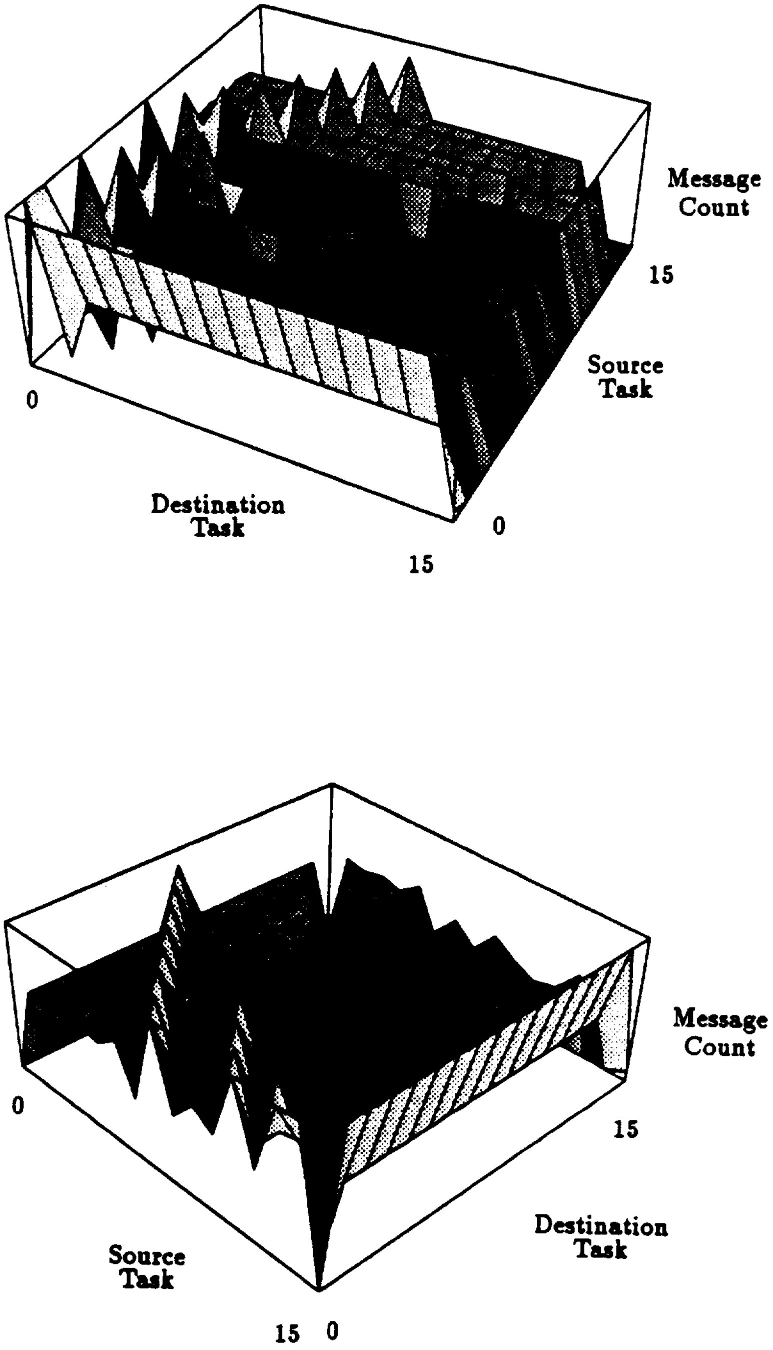


Figure 12: Perspective Views of Message Counts (Row Partitioning)

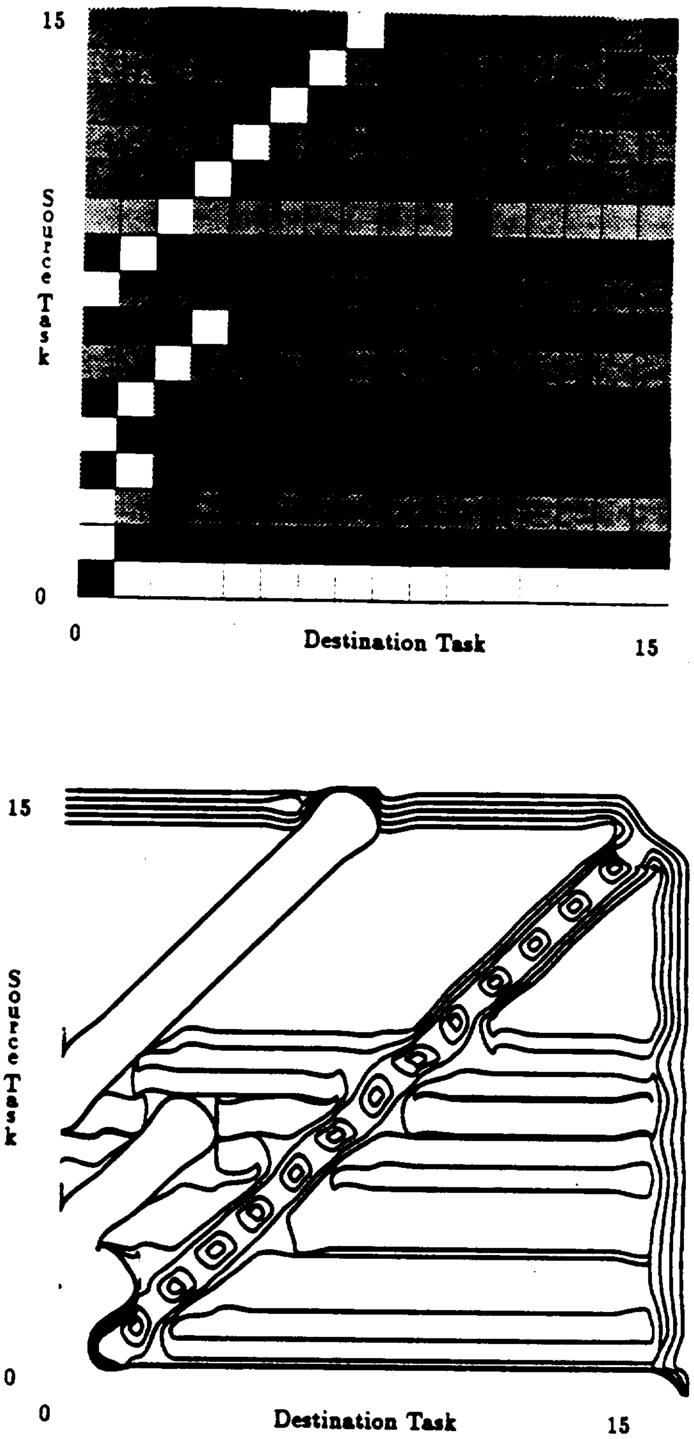


Figure 13: Density and Contour Views of Message Counts (Row Partitioning)

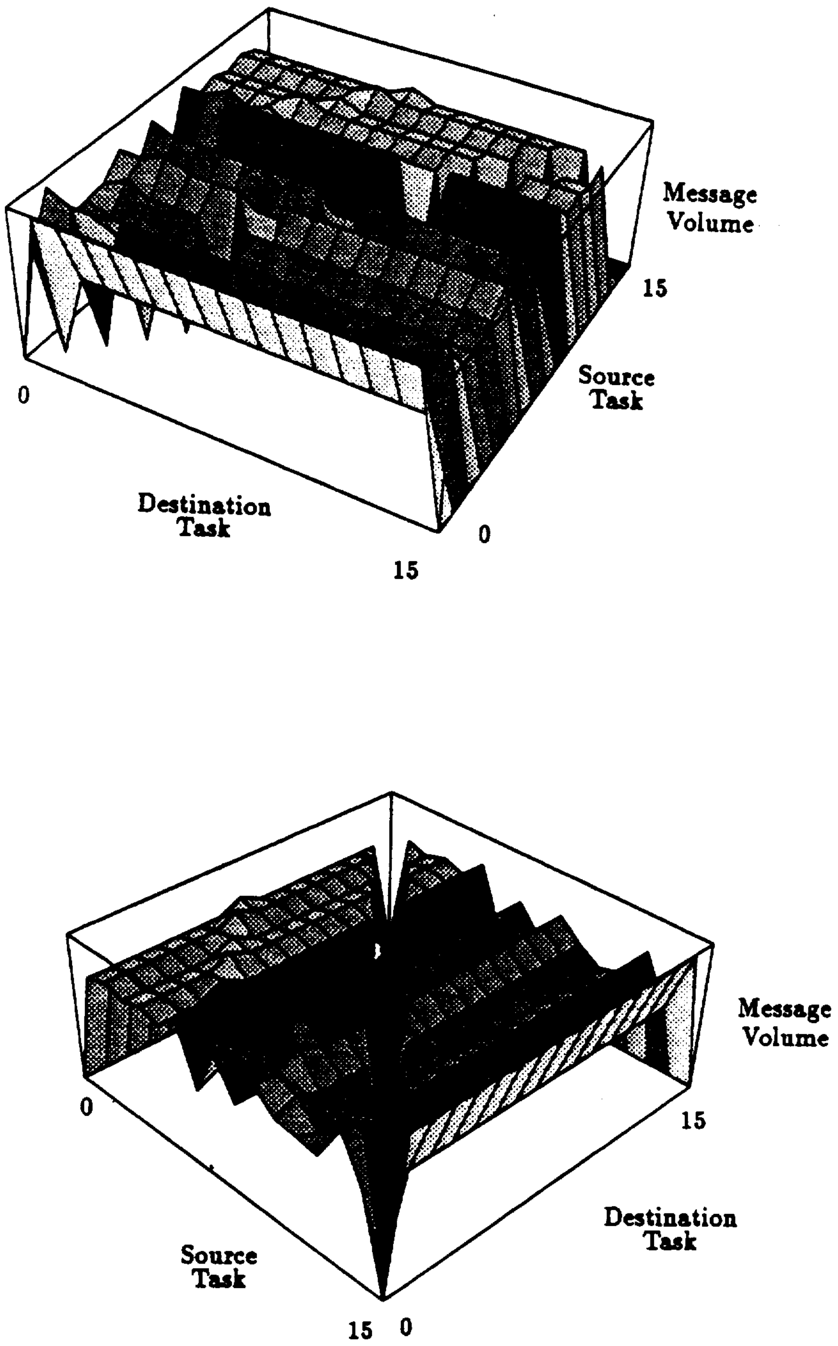


Figure 14: Perspective Views of Message Volume (Row Partitioning)

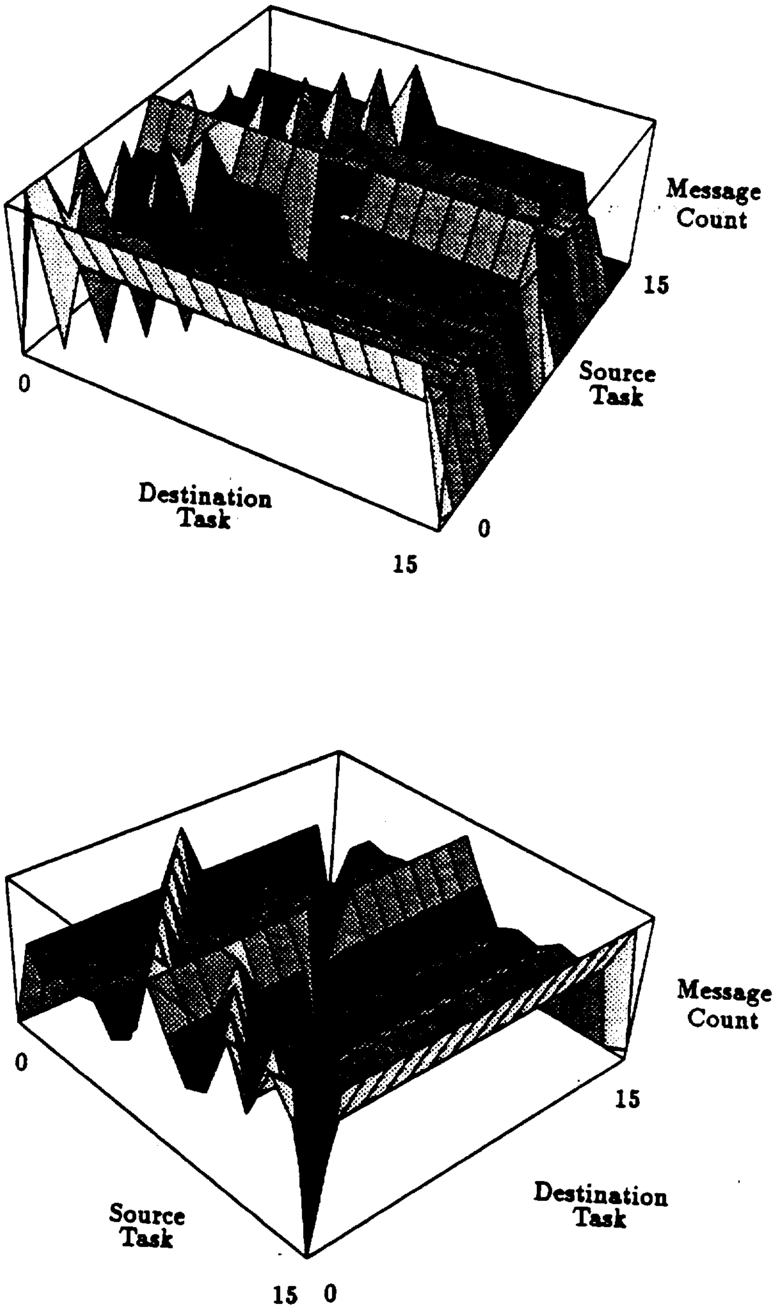
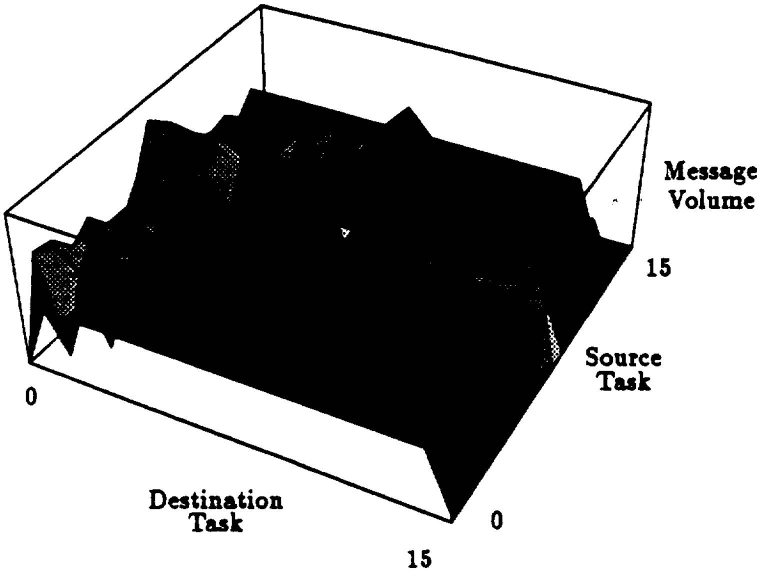


Figure 15: Perspective Views of Message Counts (Column Partitioning)



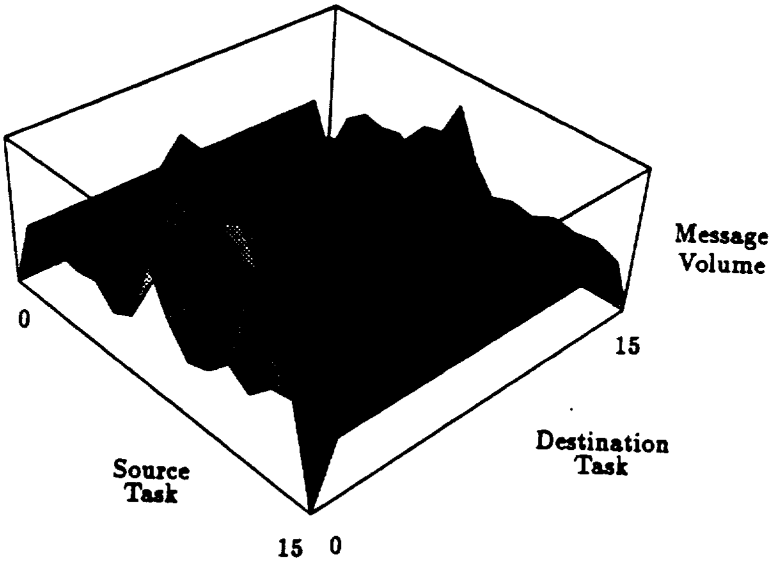


Figure 16: Perspective Views of Message Volume (Column Partitioning)

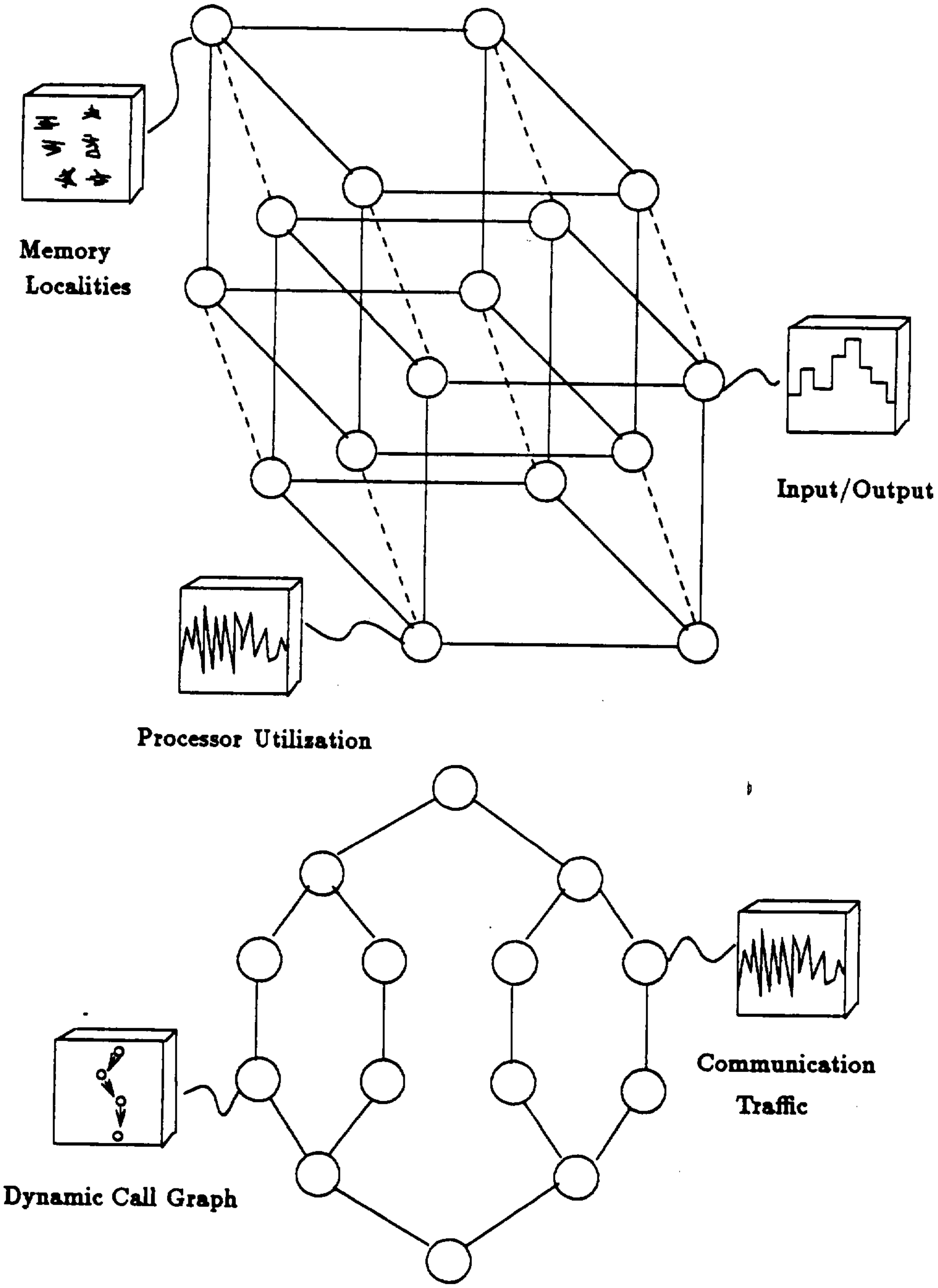


Figure 17: Tapestry Performance Visualization System

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-88-1465	2.	3. Recipient's Accession No.

4. Title and Subtitle

Visualizing Parallel Computer System Performance

5. Report Date
September 1988

6.

7. Author(s)
Allen D. Malony and Daniel A. Reed

8. Performing Organization Rept. No. R-88-1465

9. Performing Organization Name and Address

Department of Computer Science
1304 W. Springfield Ave.
240 Digital Computer Lab
Urbana, IL 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.
see page 1

12. Sponsoring Organization Name and Address

National Science Foundation
Washington, DC 20550

Department of Energy
Washington, DC 20550

Air Force Office of Scientific Research
Washington, DC 20550

NASA Langley Research Center
Hampton, VA 23665

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts

Parallel computer systems are among the most complex of man's creations, making satisfactory performance characterization difficult. Despite this complexity, there are strong, indeed, almost irresistible, incentives to quantify parallel system performance using a single metric. The fallacy lies in succumbing to such temptations. A complete performance characterization requires not only an analysis of the system's constituent levels, it also requires both static and dynamic characterizations. Static or average behavior analysis may mask transients that dramatically alter system performance.

Although the human visual system is remarkedly adept at interpreting and identifying anomalies in false color data, the importance of dynamic, visual scientific data presentation has only recently been recognized. Large, complex parallel systems pose equally vexing performance interpretation problems. Data from hardware and software performance monitors must be presented in ways that emphasize important events while eliding irrelevant details. Design approaches and tools for performance visualization are the subject of this paper.

17. Key Words

visualization, performance evaluation, parallel processing

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement unlimited	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 32
	20. Security Class (This Page UNCLASSIFIED	22. Price