2

CONF-890372--12

Parallel Graphics Algorithms on a 1024-Processor Hypercube*

SAND--89-0550C

DE89 015492

Robert E. Benner
Sandia National Laboratories, Albuquerque, NM 87185

To appear in the Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications, Monterey, CA, March 6-8, 1989.

Abstract. We have developed four parallel graphics algorithms for visualization of complex problems in PDE simulations, radar simulation, and other large applications on a 1024-node ensemble with a 16-node graphics device. We discuss the impact of system parameters on algorithm development and performance. Algorithmic issues include multistage routing of graphics data through the ensemble, non-hypercube mappings from the ensemble to the graphics system, synchronization between ensemble and graphics nodes, and synchronization between graphics nodes. These issues apply to both the present and anticipated future systems which combine highly parallel ensembles and parallel I/O devices. "Best" solutions are described for routing, mapping and synchronization on the current hardware. Implications are discussed for future hardware and software for massively parallel computers.

1. Introduction

We are engaged in research in parallel methods, algorithms, application programs, and performance models for massively parallel computer systems. In the area of application-driven research, several highly parallel applications [2] have sustained computation rates in the range of 70 to 130 MFLOPS and parallel speedups of order 1000 on a 1024-processor NCUBE/ten hypercube ensemble. Through heterogeneous use of the hypercube, a parallel radar simulation [3] runs an order of magnitude faster on the ensemble than on the CRAY X-MP or CRAY Y-MP.

Highly parallel applications require graphics and other parallel I/O systems and software capable of supporting large computations in the sciences and engineering. Hence, we are concerned with parallel graphics algorithms for real-time visualization of large, multidimensional simulations. Parallel graphics hardware is commercially available, for example, on both the NCUBE/ten and the Thinking Machines' CM-2, as well as graphics supercomputers such as the Stellar GS-1000 and Ardent Titan. This paper focuses on graphics for our 1024-node hypercube, but we anticipate that much of the discussion will apply to future parallel I/O devices, particularly those associated with MIMD computers.

To date, performance monitors [4, 6] for parallel computations have been the focus of parallel graphics research on the NCUBE RT Graphics System. This graphics device has 16 nodes which receive data from the 1024-node ensemble. Graphics nodes have their own (non-hypercube) interconnect and have only 128 K byte of local memory each, of which 40 K byte acts as a message buffer.

This paper addresses parallel graphics hardware and software limitations and ways to minimize their impact, both on the NCUBE and on future systems of highly parallel ensembles and parallel I/O devices. Section 2 summarizes hardware parameters of our hypercube, with an emphasis on system constraints upon graphics performance. Section 3

* This work supported by the Applied Mathematical Sciences Program, U.S. Department of Energy, Office of Energy Research. It was done at Sandia National Laboratories, operated for the U.S. Department of Energy under contract number DE-AC04-76DP00789.

So

outlines techniques for dealing with these constraints. Algorithmic issues include multistage routing of graphics data through the ensemble, non-hypercube mappings between ensemble and graphics device, synchronization between ensemble and graphics nodes, and synchronization between graphics nodes. "Best" solutions are described for routing, mapping and synchronizing on current hardware.

In particular, four parallel graphics algorithms have been developed for visualization on highly parallel ensembles. Section 4 presents generic algorithms for host, ensemble node, and graphics node processors, as well as specific graphics algorithms, a benchmark, and results. Section 5 summarizes advances in our understanding of parallel graphics and discusses implications for future hardware and software for massively parallel computers with high-performance, parallel I/O devices.

2. Parallel Graphics Research Facilities

We are pursuing two research directions in parallel graphics, based on different interfaces to the hypercube. Two graphics devices are included in our current hardware configuration, which is shown in Figure 1: an NCUBE RT Graphics System and a Stellar GS-1000 Graphics Supercomputer. The RT Graphics System is a suitable vehicle for real-time display of images from 2D simulations or performance data, but does not currently have a flexible, high-performance graphics software environment for real-time 3D graphics or object-oriented graphics. An alternative approach to object-oriented or 3D graphics is to interface the 1024-node hypercube to a graphics supercomputer, such as the four-processor, shared memory, Stellar GS-1000. The interface is presently under development and will not be discussed further in this paper.

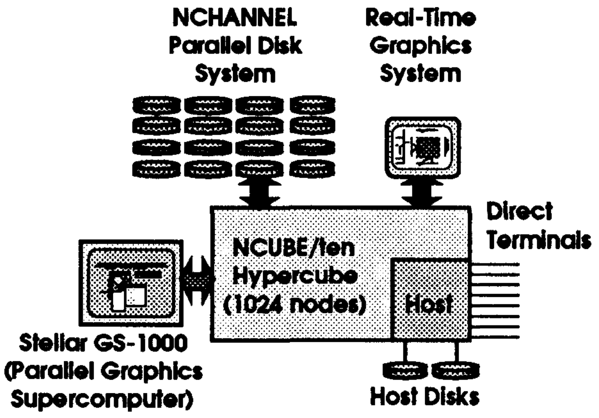
NCHANNEL Parallel Disk System
Real-Time Graphics System
Direct Terminals
NCUBE/ten Hypercube (1024 nodes)
Host
Stellar GS-1000 (Parallel Graphics Supercomputer)
Host Disks

Figure 1. Sandia Hypercube Environment

All memory is distributed in the NCUBE/ten, both within the hypercube itself and within the RT Graphics System. This latter system has 16 nodes which receive data from the 1024-node ensemble. Information is shared between processors by explicit communications across channels; 8 of the 11 channels on each graphics node connect to the ensemble. (These 128 channels have a composite bandwidth of more than 64 M byte/sec.) Graphics nodes use the remaining 3 channels for their interconnect. These nodes have only 128 K byte of local memory each, of which 40 K byte acts as a message buffer. With system calls from Fortran, sending a message takes about 0.35 msec to start and then 2 μsec per byte. However, time to move data across a channel can be partly overlapped [2] with computations or other communications. A real-time display rate of 30 frame/sec is quoted for the graphics system [5], based on communication bandwidths from the ensemble to the graphics nodes and, in turn, from the graphics nodes to the display.

3. Synchronization and Routing Issues

There are three hardware issues associated with the RT Graphics System which must be addressed in order to obtain high performance: (1) the interconnection network of the 16 graphics nodes, which dictates how efficiently they can communicate and synchronize with each other; (2) the mapping of graphics node memories to the display, which dictates the amount of data movement necessary to display images correctly; and, (3) the interconnection network between graphics nodes and ensemble nodes, which dictates how efficiently (relative to 30 frames/sec) data can be transferred to the graphics system.

The first-generation of vendor software for the RT Graphics System does not support the quoted 30 frame/sec display rate. The library of graphics routines called by ensemble nodes (1) performs data collection and transfer for each graphics node in turn, rather than in parallel; (2) limits the user to one message per graphics node per call, or about a half-frame per call; (3) uses linear synchronization of graphics nodes, *i.e.*, graphics node 0 reads messages from the other 15 nodes, and then writes back to all 15 (*cf.* § 3.1); (4) reorganizes data on the graphics nodes, rather than the ensemble nodes, for correct mapping to the display (*cf.* § 3.2); and (5) doesn't make use of direct channels from ensemble nodes to graphics nodes (*cf.* § 3.3). The resulting software is robust and can handle graphics data from an arbitrary number of processors. However, it requires about 1.5 seconds per half-frame image (0.3 frame/sec) from 1024 nodes.

3.1 Graphics Interconnect

Each graphics node, like their counterparts in the 1024-processor ensemble, has 11 bidirectional DMA channels. Eight of the channels connect to ensemble nodes, so that only three are available for the graphics nodes to communicate with each other and with the rest of the hardware (*e.g.*, an 80186 processor) on the graphics board. Figure 2(a) outlines the graphics node interconnection network, which uses three channels on each node. Two additional channels per node would be required to support a hypercube interconnect of the graphics node: four channels for the hypercube and an additional channel on graphics node 0 for communication with the 80186 processor.

Fast synchronization and communication between the graphics nodes require novel, non-hypercube algorithms. For example, we cannot synchronize the graphics nodes in four bidirectional communication steps, as on a 16-node hypercube, but we can synchronize to one graphics node in five steps by using any of several trees that are embedded in the graphics node network. Full synchronization requires ten steps—a fanin and fanout through the tree—as opposed to the eight steps (counting bidirectional reads and writes separately) required by a hypercube global exchange algorithm. Figure 2(b) shows the tree used in our graphics node synchronization, fanin, and fanout software.

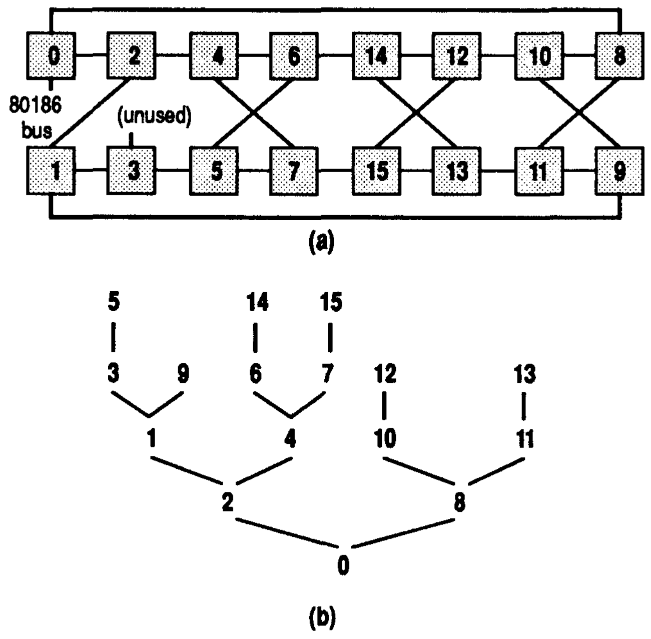

Figure 2. RT Graphics System: (a) Node Interconnection Network (*cf.* [5]); (b) a Five-Level Tree

3.2 Memory-to-Display Mapping

Figure 3 outlines the mapping of the display to the graphics node memories in the RT Graphics System. Groups of 32 consecutive pixel columns are assigned to the graphics nodes as shown. As a result, each graphics node is responsible for 32 pairs of widely-separated pixel columns. Furthermore, pixel rows are mapped consecutively into memory. This assignment and mapping scheme forces collection and reordering of pixel data on a grand scale for grid-based and other general simulations. In fact, it is hard to envision an application for which this mapping is optimal. We mitigate the effect of these hardware constraints upon performance by using assembly language kernels to move and reorder pixels and by using communication channels for pixel collection and transmission to the graphics system, as discussed below.

3.3 Graphics-Ensemble Interconnect

Figure 4 outlines the interconnection of nodes on one of the first eight boards in the ensemble to an RT Graphics System board in backplane slot 2 [5]. Our graphics software identifies these channels and makes explicit use of them in moving data by means of two-step routing from ensembles of at least 64 nodes to the graphics system. In the two step routing messages are first sent to an ensemble node in a set S,

$$S = \{4, 6, 12, 14, 20, 22, 28, 30, 37, 39, 45, 47, 53, 55, 61, 63, \ldots\},$$

which has a channel to the graphics node responsible for the data in the message. The pattern of the entries of S is repeated on each of the first eight boards of 64 processors in the ensemble. On fewer than 64 ensemble nodes, our graphics routines use the default routing from the ensemble to the graphics system.

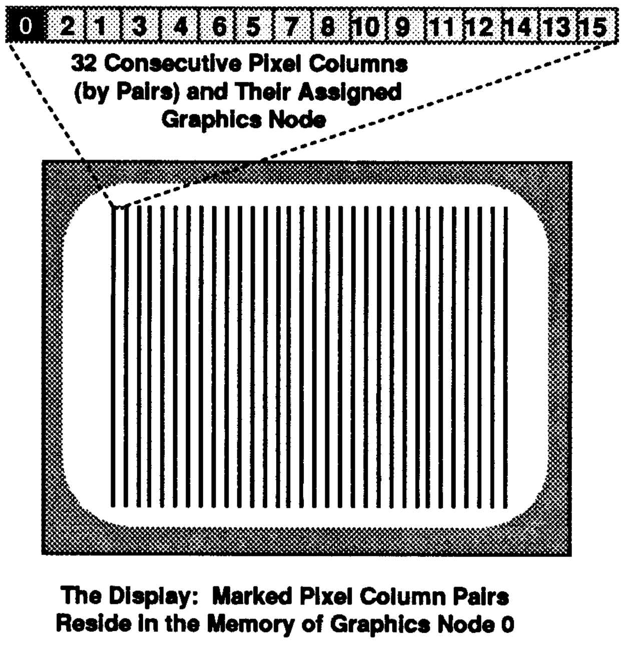
0 2 1 3 4 6 5 7 8 10 9 11 12 14 13 15
32 Consecutive Pixel Columns (by Pairs) and Their Assigned Graphics Node
The Display: Marked Pixel Column Pairs Reside in the Memory of Graphics Node 0

Figure 3. Mapping of Pixel Columns on the Display to Graphics Node Memories

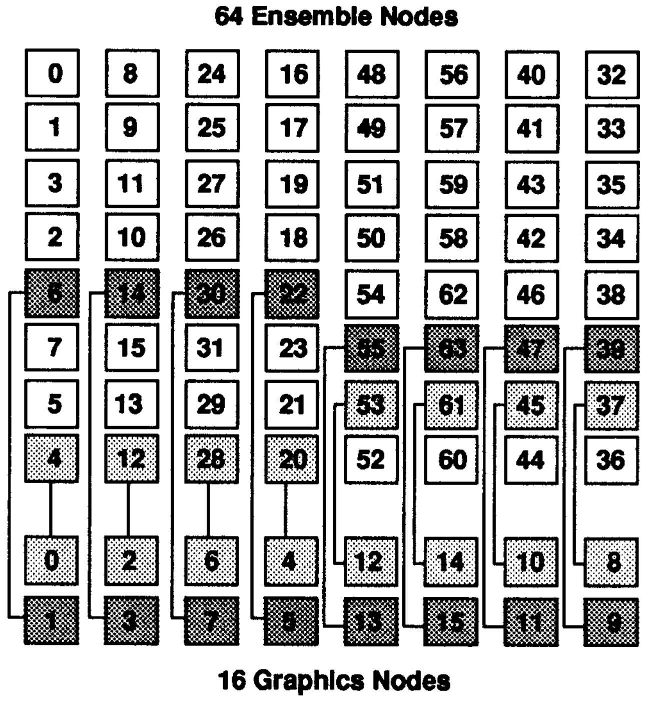

Figure 4. Channels Between the Graphics Board and a Board of Ensemble Nodes.

4. Graphical Methods and Performance

Parallel graphics algorithms are discussed below in terms of a generic parallel implementation, four specific parallel graphics algorithms, a test problem, and results.

4.1 Generic Parallel Implementation

The parallel implementation of an application program with parallel graphics is summarized by the following host processor and ensemble node algorithms, *H* and *E*, respectively (*cf.* [2]), as well as a generic graphics node algorithm, *G*. Some references are also made to specific graphics algorithms (Column, Column-Section, Column-Collection, and Global Collection) introduced in § 4.2.

ALGORITHM *H*
Host Program for Applications with Parallel Graphics

*H*1. [Get job parameters.] Prompt the user for input values. For example, read the subcube dimension, grid size in the *x* and *y* directions, model parameters, names of input files, etc. Read the number of pixels in the horizontal (*x*) and vertical (*y*) directions if graphical display of solutions is specified. In addition, read *k* and *l*, the display intervals for inner and outer solutions, respectively. Read the number of sections per pixel column for the Column-Section algorithm. (The number of sections is given by the number of processors assigned to the *y* direction for the Column-Collection or Global Collection algorithm).
*H*2. [Open and load cube.] Open a subcube of the hypercube, and send the node program (Algorithm *E*) to all ensemble nodes using a logarithmic fanout (*cf.* [2]).
*H*3. [Send input.] Send the input values to ensemble node 0 (step *E*2).
*H*4. [Start output.] Create the output header, including input values from step *H*1.
*H*5. [Collect output.] Collect output data from ensemble node 0 (step *E*6 or *E*7) and print them in the output file.
*H*6. [Monitor solver.] Check the *message type* of step *H*5 for solver completion, continuation, or failure. Repeat step *H*5 if *message type* denotes continuation.
*H*7. [Post-mortem.] Receive and print timing statistics from ensemble node 0 (step *E*8).
*H*8. [Done.] Close subcube. Stop the host program. ▮

ALGORITHM *E*
Ensemble Node Program for Applications with Graphics

*E*1. [Start.] Record the time and execute a system call to get this node's process number (hereafter referred to as the node number) and the allocated cube dimension.
*E*2. [Get job parameters.] Receive job parameters from the host on node 0 (step *H*3). (This data is then propagated using a logarithmic fanout).
*E*3. [Create spatial topology.] Use data from *E*1 and *E*2 to compute node numbers of nearest neighbors in an appropriately dimensioned (usually 2D or 3D) subset of the hypercube interconnect, a binary-reflected gray code order (*e.g.*, [5]).
*E*4. [Initialize graphics.] Open the graphics system and send the graphics operating system (GRAPHOS) and graphics node program (Algorithm *G*) from node 0 to the 16 graphics nodes using linear load routines. Send input values from node 0 to graphics node 0 (hexadecimal node address 8007 [5]) and receive a synchronization message in return (step *G*3). Synchronize nodes on a global exchange.
*E*5. [Start outer iteration.] Start outer timer. Begin outer iteration: *e.g.*, time stepping for a transient problem, Newton iteration for a nonlinear iteration. Set outer iteration counter *j* to 1.
*E*6. [Inner solver.] Start inner timer. Set inner iteration counter *i* (if needed) to 1. Begin inner solver: *e.g.*, preconditioned conjugate gradient iteration or multifrontal solver for linear systems of equations. Iterate so long as *i* is less than or equal to an upper bound (a job parameter). Compute and send pixel data to the graphics system (step *G*7) every *k*-th inner iteration. Stop the inner timer, send a message to the host that the inner solver failed (step *H*5), and go to step *E*8 if *i* exceeds its upper bound, or the direct solver fails, etc.
*E*7. [Update the outer solution.] Send a message from node 0 to the host if the outer

iteration is complete. Compute and send pixel data to the graphics system (step *G*7) every *l*-th outer iteration. Send a message from node 0 to the host (step *H*5) if *j* exceeds its upper bound (a job parameter). Otherwise, increment *j* and go to step *E*6.

*E*8. [Complete timings.] Gather complete (steps *E*1-*E*7) and partial (steps *E*5-*E*7) timing statistics by a global exchange. Receive graphics node timing statistics (step *G*10) on node 0. Send ensemble and graphics node timing statistics from node 0 to the host (step *H*7). Stop the ensemble node program. ▮

ALGORITHM *G*
Graphics Node Program

---

*G*1. [Start.] Record the time and execute a system call to get this node's number.

*G*2. [Set graphics interconnects.] Calculate the parent and children of this node in a five-step tree algorithm used for data fanout and graphics processor synchronization.

*G*3. [Get job parameters.] Receive job parameters (number and size of frames) on node 0 from ensemble node 0 and acknowledge receipt (step *E*4). Propagate job parameters to all nodes using the tree algorithm (§ 3.1).

*G*4. [Set array-graphics interconnects.] Use data from step *G*3 to compute the expected number of and frame buffer destinations of messages from the ensemble, such that images are centered on the display.

*G*5. [Start outer iteration.] Start outer timer and set outer iteration counter *j* to 0.

*G*6. [Start frame.] Start inner timer and set inner iteration counter *i* to 0.

*G*7. [Acquire data.] Read a message from the 1024-processor ensemble (step *E*6 or *E*7). Read the message directly into the display array if the Global Collection algorithm is in use on the ensemble nodes. Otherwise, read the message into a buffer and, based on several bytes of header information in the buffer, perform an assembly language move with stride 64 into the display array. Increment *i* by one. Repeat step *G*7 if *i* is less than the expected number of messages.

*G*8. [Update display.] Synchronize nodes using the tree algorithm. Issue a DMA write command from node 0 to the 80186 processor in the graphics system. Send a completion message for the frame from node 0 to ensemble node 0.

*G*9. [Complete frame.] Increment *j* by 1. Go to step *G*6 if *j* is less than the expected number of frames.

*G*10. [Complete timings.] Gather complete (steps *G*1-*G*9) and partial (steps *G*5-*G*9, steps *G*6-*G*8) timing statistics by the tree algorithm. Send timing statistics from node 0 to ensemble node 0 (step *E*8). Stop the graphics node program. ▮

4.2 Four Parallel Graphics Algorithms

We now outline four specific graphics algorithms. The simplest is the Column algorithm, which is applicable to simulations, such as radar simulation, which can conveniently produce complete columns of pixels. The others are the Column-Section, Column Collection, and Global Collection algorithms, which are applicable to 2D grid calculations and, in some cases, more arbitrary decompositions. Therefore, the descriptions of the last two algorithms frequently refer to operations on the 2D processor grid (*e.g.*, Figure 4) onto which a 2D application has been mapped. The sections handled by the Column-Section algorithm can be either the portions of pixel columns that belong to each node in the 2D processor grid, or portions of pixel columns that are generated in an arbitrary order.

The Column algorithm first sends a pixel column from the ensemble node which calculated it to an ensemble node in the set *S* (§ 3.3) which has a channel to the responsible graphics node. The message is read by the recipient ensemble node and rewritten to the graphics node. Each graphics node receives as many as 64 messages per frame. At most two nodes (from the set of nodes 0 to 127) are allowed to write to each graphics node; this limitation prevents overflow of graphics node message buffers. Graphics nodes synchronize with each other using the tree algorithm (§ 3.1). Graphics node 0 also synchronizes with ensemble node 0 at the end of each frame. The resulting software is robust and can handle graphics data from an arbitrary number of ensemble nodes.

The Column-Section algorithm first sends each section of a pixel column from the ensemble node which calculated it to the appropriate node in the set *S*. This algorithm proceeds as the Column algorithm, except that each graphics node receives a number of messages that is up to 64 times the number of sections in each pixel column. This algorithm seems ideal for 2D grid calculations and is readily extended to arbitrary geometries by slightly increasing the amount of header information in each message. However, the algorithm is presently not robust when the number of sections per pixel column is large (*e.g.*, greater than 24), and, in fact, even fails consistently in one case (§ 4.4) when each pixel column is divided into only two sections.

The Column-Collection algorithm first collects sections of a pixel column by means of a logarithmic exchange among columns of processors in the 2D processor grid (*e.g.*, nodes 0 to 7 in Figure 4). Complete pixel columns then reside on one node in the column of processors. That node applies the Column algorithm to the pixel column. This algorithm is well-suited to 2D grid calculations, is robust (like the Column algorithm), but does not readily extend to arbitrary geometries.

The Global Collection algorithm first exchanges sections of pixel columns among rows of processors in the 2D processor grid (*e.g.*, nodes 0, 8, 16, ..., 56 in Figure 4). At each step of the exchange each processor sends data belonging to half of the graphics nodes and receives data belonging to the remaining half. Upon completion of the row exchange, each node has data belonging to one or two graphics nodes. The data is then reordered so that its order corresponds to the contiguous order in graphics node memory.

Data is then collected by means of a logarithmic exchange down half-columns of processors in the 2D processor grid (*e.g.*, nodes 0 to 3 in Figure 4). This results in two messages per graphics node per frame, which are then sent in turn from the node in each half-column which either has a channel to or is at most distance two from the recipient graphics node. For example, in the first column of processors in Figure 4, nodes 0 and 2 first write to graphics nodes 0 and 1, respectively, followed by nodes 4 and 6. Graphics nodes synchronize with each other and with the ensemble nodes to ensure that their two large messages do not collide.

The Global Collection algorithm is robust, but is limited to 2D grid calculations on regular geometries. It requires more than 130 K byte of ensemble node memory. In its present version it is limited to simulations on 64 to 512 ensemble nodes, *i.e.*, the smallest and largest subcube sizes for which complete sets of channels to the graphics system are available. A general extension to 1024 nodes is nontrivial, because some nodes will become idle during the initial row exchange process whenever 32 or more nodes are assigned to the *x* direction in the 2D processor grid.

4.3 A Benchmark

The algorithms presented above have been used in a number of applications: the Column algorithm in radar simulation [3], and the others in finite difference and finite element computations. For example, the wave mechanics, fluid dynamics, and structural analysis applications presented in [2] supplied graphical or disk output at the end of a simulation; the Column-Collection and Global-Collection algorithms now provide continuous output from these highly parallel simulations.

In this paper, a simple test problem is used to compare all four graphics algorithms. The benchmark produces a hyperbolic test pattern on the screen. The computational effort per pixel is uniform: 4 integer adds, 3 integer multiplies, and 1 integer divide. The elapsed time required to compute a frame of 1024 by 768 pixels scales perfectly with the number of processors: 27.021 seconds on one processor and 0.026 seconds on 1024 processors. These elapsed times correspond to computation rates of 0.233 M and 238 M integer operations per second, respectively.

4.4 Results

We present timings for the four graphics algorithms of § 4.2 in terms of either the time required to collect (if necessary), transmit, and display a full 1024 by 768 pixel frame, or the M byte per second rate for collection, transmission and display regardless of the image size. Although these measures are more informative than a vendor's quoted display rate, they are not meant to be (and should not be taken as) all-encompassing performance measures [1] in the fashion of MFLOPS, parallel speedup, polygons/sec, etc. The overall display rate for a real application is determined by user-specified display intervals (parameters *k* and *l* in § 4.1), the time to compute a frame (*e.g.*, as reported in § 4.3 for the test problem), and the sustained display rate of the graphics hardware and software.

The range of run times in Table 1 for the Column-Collection and Global-Collection algorithm represent two cases: (1) twice as many processors assigned to the *x*-direction as to the *y*-direction (the faster case); and (2) twice as many processors assigned to the *y*-direction as to the *x*-direction. Typical display rates are 2 to 3 frame/sec for the Global Collection algorithm and 0.5 to 2.5 frame/sec for the Column Collection algorithm. The results show that if one can afford to use a quarter of the ensemble memory (~130 K byte per node) as a buffer for graphics data, then the Global Collection algorithm is the method of choice for 2D grid calculations on a large ensemble.

Table 1
Graphics System Run Time for Various Graphics Algorithms: Image of 1024 by 768 Pixels. Two Sections were Used in the Case of the Column-Section Algorithm.

	Graphics System Run Time (sec)			
Number of Processors	Column	Column Section	Column Collection	Global Collection
1	0.96	1.47	1.05	.
2	0.49	0.74	0.54 - 2.39	.
4	0.26	0.38	1.18	.
8	0.14	0.20	0.57 - 1.91	.
16	0.087	0.12	0.86	.
32	0.17	(0.47)	0.52 - 1.20	.
64	0.18	0.25	0.79	0.38
128	0.21	0.26	0.45 - 0.81	0.29 - 0.45
256	0.20	0.25	0.53	0.36
512	0.21	0.23	0.38 - 0.66	0.50
1024	0.21	0.27	0.41	.

The highest display rates presented in Table 1 are associated with the Column algorithm. Figure 5 summarizes the dependence of sustained display rate upon ensemble size for the Column algorithm. The display rate of the Column algorithm increases nearly linearly through 16 nodes, beyond which the inability of graphics nodes to process bursts of incoming data dominates. The highest rate is 9.04 M byte/sec, or 11.5 frame/sec, when 16 ensemble nodes are used. Display rates of about 4 M byte/sec, or 5 frame/sec, are typical when 32 or more ensemble nodes are used. Only minor variations in display rate are observed between using 16 or 32 channels to the graphics system. This observation suggests that the speed at which graphics nodes read and move data limits the display rate, which explains message buffer overflow when more than 32 channels are used.

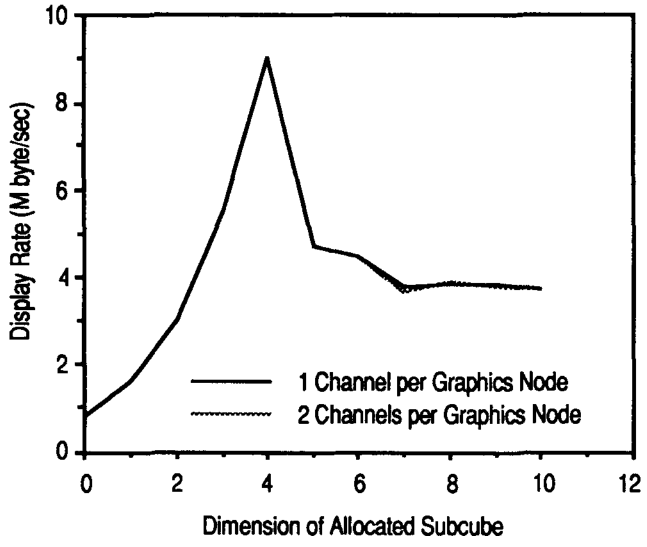

Figure 5. Display Rate as a Function of Communication Channels for a 1024 by 768 Image (Column Algorithm)

A sizable performance penalty is paid just for splitting columns in two and applying the Column-Section algorithm (Table 1). In addition, the Column-Section algorithm fails consistently on 32 ensemble nodes: some ensemble nodes are swamped by message traffic unless the slower, two-stage routing scheme is used. The run time for the Column-Section algorithm is asymptotically linear in the number of sections per pixel column, as shown in Figure 6. This indicates that message start-up time dominates, as expected, when many small graphics messages are transmitted.

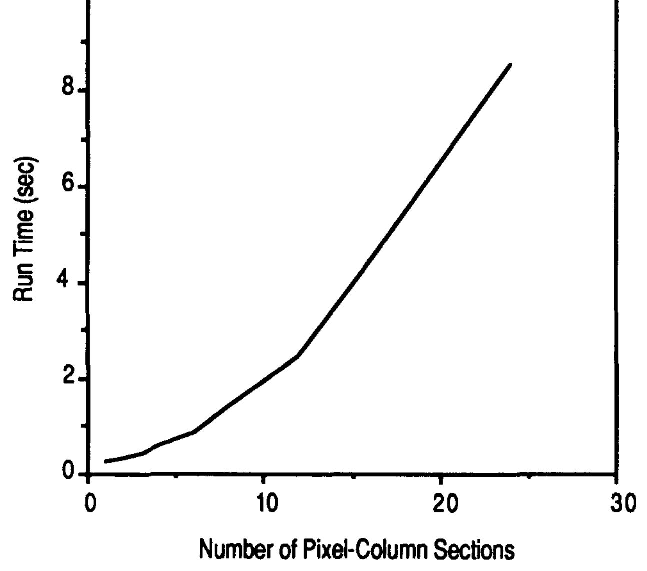

Figure 6. Run Times for the Column-Section Algorithm (1024 ensemble nodes, 1024 by 768 Image)

Next, the effect of small changes in the image size is explored in Figure 7 with the Column algorithm. The display rate is relatively insensitive, as expected, to the number of pixels in a column. (The variation in display rates for repeated simulations, about 0.05 M byte/sec, is evident in the pair of results for a 768 by 768 image.) However, display rate varies by almost 15% in the number of pixel columns (the highest sustained display rate is 9.32 M byte/sec or 11.9 frame/sec). Note that images are centered on the screen (§ 4.1, step *G*4), so that a small change in the number of columns drastically changes the mapping of ensemble node data to graphics nodes. This phenomenon suggests that a better algorithm would approximately center the image so as to keep the left boundary of the image at a multiple of 32 columns. This scheme would preserve the mapping of the first two pixel columns to graphics nodes 0, etc.

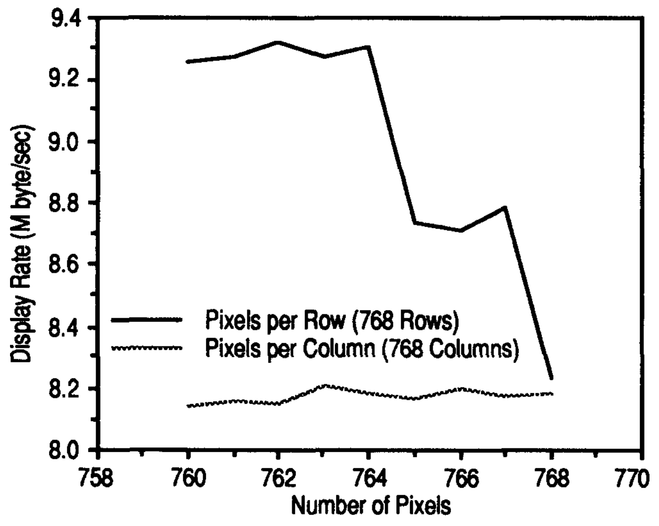

Figure 7. Display Rate from 16 Array Nodes as a Function of Image Size (Column Algorithm)

Figure 8 shows the dependence of display rate upon a broad range of image sizes for the Column algorithm. Display rate tends to increase as image size increases in increments of 32 rows and columns, because message startup is amortized by the longer messages. However, the trend is somewhat erratic, because of the dynamics of message traffic. This effect is seen with both default message routing, as on 16 ensemble nodes, and explicit two-stage routing, as on 64 or more nodes.

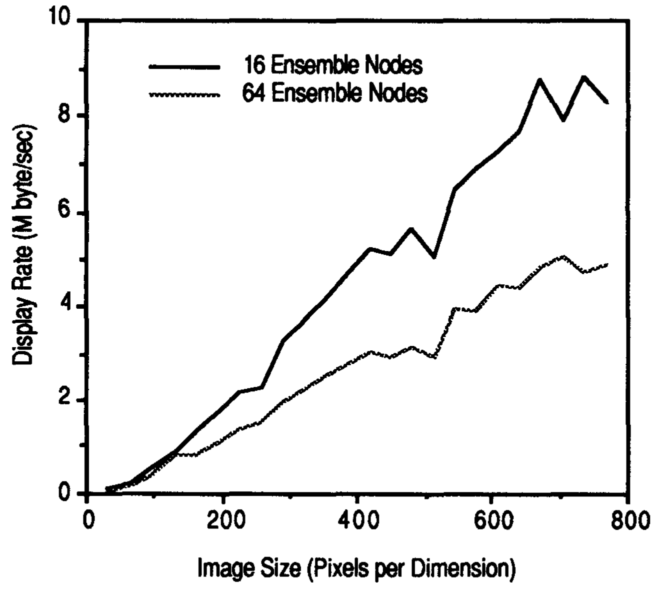

Figure 8. Display Rate as a Function of Image Size (Column Algorithm)

4.5 Graphics Performance Parameters

In terms of performance evaluation of parallel graphics, we have attempted to identify and quantify meaningful parameters in addition to display time or display rate. Table 2 summarizes dimensionless parameters which govern the performance of the RT Graphics System and our algorithms. The ideal value of each parameter is unity.

Table 2. Performance Evaluation of Parallel Graphics: Dimensionless Parameters

Hardware Parameters:	
(graphics nodes)/(ensemble node)	= 0.016
(channels to graphics)/(ensemble node)	= 0.125
(graphics sync steps)/(hypercube sync steps)	= 0.8
(contiguous pixels in memory)/(total pixels)	= 0.00004
System Software Parameter:	
(GRAPHOS buffer size)/(frame buffer size)	= 0.8
User Software Parameter:	
(channels to graphics used)/(total channels)	= 0.25

Although all of the parameters deviate significantly from the ideal, the organization of pixels in memory and the GRAPHOS buffer size have proven to be more critical than the others in terms of their impact on performance. For example, neither the number of graphics nodes in the system nor the number of channels to the graphics system are crucial, so long as sufficient space is available to buffer message traffic. The number of steps required to synchronize the graphics system is more of a issue in terms of software development (for synchronization and data fanin and fanout) than performance. Finally, the user-software restriction on the number of channels used is not an independent parameter, but rather a consequence of the other parameters.

5. Summary

Four parallel graphics algorithms have been developed for visualization of complex problems in PDE simulations, radar simulation, and other large applications. Parallel graphics hardware and software limitations have been addressed. Algorithmic techniques for dealing with system constraints include multistage routing of graphics data through a large hypercube ensemble, explicit use of non-hypercube mappings between ensemble and graphics device, tree algorithms for fast synchronization and data fanin and fanout between graphics nodes, and synchronization between ensemble and graphics nodes.

Display rates of up to 11.9 frame/sec have been achieved on an NCUBE/ten hypercube and RT Graphics System. Display rates of 2 to 5 frame/sec are typical across a wide range of graphics algorithms, applications, image sizes, and ensemble sizes. Key parameters which govern the performance of the RT Graphics System have been identified. These parameters are associated with some operating system and hardware issues to be resolved in future systems. Operating system issues include having graphics message buffers large enough (> 49 K byte) to hold all of the incoming data associated with a frame, and unbuffered message passing. Hardware issues which should be addressed in future systems include simplified memory-to-display mappings and larger graphics node memories.

More issues in performance evaluation need to be explored, as well as 3D and object-oriented visualization, and use of the present set of algorithms to transfer data to an NCUBE-Stellar interface and other I/O interfaces. We have assembled a library of our evolving software for graphics, disk I/O, interprocessor communications, etc., for use in parallel applications. The library routines use the novel techniques introduced into the graphics algorithms to ensure robustness.

Many of these graphics techniques are useful in other I/O and communications tasks. For example, the NCUBE NCHANNEL parallel disk system consists of 16 disks served by 16 processor nodes on an I/O board that resembles the RT Graphics System in some respects. Application program issues, such as message buffer limitations and message routing considerations are the same for both systems, except that messages and file operations on the NCHANNEL nodes are handled by the DISKOS operating system rather than a user program. Preliminary tests indicate that some of the algorithms presented above will also safely transfer images to the parallel disk system at a sustained transfer rate of a few M byte/sec.

Acknowledgements

I thank Gary Montry for several helpful discussions concerning this research. I also thank Rod Morison of Caltech, Bert Stiles of the University of South Carolina, and Matthew Hall and Chaz Sliger of NCUBE for additional stimulating discussions. Finally, I thank John Gustafson for numerous helpful suggestions concerning the paper.

References

[1] COMMITTEE ON SUPERCOMPUTER PERFORMANCE AND EVALUATION, *et al.*, "An Agenda for Improved Evaluation of Supercomputer Performance," National Academy Press, Washington, D.C. (1986).

[2] GUSTAFSON, J. L., MONTRY, G. R., AND BENNER, R. E., "Development of Parallel Methods for a 1024-Processor Hypercube," *SIAM J. Sci. Stat. Comput.*, 9 (1988), pp. 609-638.

[3] GUSTAFSON, J. L., BENNER, R. E., SEARS, M. P., AND SULLIVAN, T. D., "A Radar Simulation Program for a 1024-Processor Hypercube," *Proc. Supercomputing '89*, (1989), submitted.

[4] MORISON, R., "Interactive Performance Display and Debugging on the NCUBE Real-Time Graphics System," *Proc. 3rd Conf. Hypercube Concurrent Comput. Appl.*, 1 (1988), pp. 760-765.

[5] NCUBE Users Manual, Version P2.1, NCUBE Corp., Beaverton, Oregon (1987).

[6] TOLLE, D., "A Graphics Facility Useful for Performance Monitoring on the NCUBE," *Proc. 3rd Conf. Hypercube Concurrent Comput. Appl.*, 1 (1988), pp. 766-771.