

# A Radar Simulation Program for a 1024-Processor Hypercube\*

John L. Gustafson, Robert E. Benner, Mark P. Sears, and Thomas D. Sullivan  
*Sandia National Laboratories, Albuquerque, NM 87185, U.S.A.*

**Abstract.** We have developed a fast parallel version of an existing synthetic aperture radar (SAR) simulation program, **SRIM**. On a 1024-processor NCUBE hypercube, it runs an order of magnitude faster than on a CRAY X-MP or CRAY Y-MP processor. This speed advantage is coupled with an order of magnitude advantage in machine acquisition cost. **SRIM** is a somewhat large (30,000 lines of Fortran 77) program designed for uniprocessors; its restructuring for a hypercube provides new lessons in the task of altering older serial programs to run well on modern parallel architectures. We describe the techniques used for parallelization, and the performance obtained. Several novel approaches to problems of task distribution, data distribution, and direct output were required. These techniques increase performance and appear to have general applicability for massive parallelism. We describe the hierarchy necessary to dynamically manage (*i.e.*, load balance) a large ensemble. The ensemble is used in a *heterogeneous* manner, with different programs on different parts of the hypercube. The heterogeneous approach takes advantage of the independent instruction streams possible on MIMD machines.

**Keywords.** Algorithms, hypercubes, load balancing, parallel processing, program conversion, radar simulation, ray tracing, SAR.

## 1. INTRODUCTION

We are developing parallel methods, algorithms, and applications for massively parallel computers. This includes the development of production-quality software, such as the radar simulation program **SRIM**, for execution on massively parallel systems. Here, *massive parallelism* refers to general-purpose Multiple-Instruction, Multiple-Data (MIMD) systems with more than 1000 autonomous floating-point processors (such as the NCUBE/ten used in this study) in contrast to Single-Instruction, Multiple-Data (SIMD) systems of one-bit processors, such as the Goodyear MPP or Connection Machine.

Previous work at Sandia has shown that regular PDE-type problems can achieve near-perfect speedup on 1024 processors, even when all I/O is taken into account. The work described in [9] made use of static load balancing, regular problem domains, and replication of the executable program on every node. This is the canonical use [8] of a hypercube multicomputer [6, 10, 11, 17], and we observed speeds 1.6 to 4 times that of a conventional vector supercomputer on our 1024-processor ensemble.

We now extend our efforts to problems with dynamic load balancing requirements, global data sets, and third-party application programs too large and complex to replicate on every processor. The performance advantage of the ensemble over conventional supercomputers appears to *increase* with the size and irregularity of the program; this observation is in agreement with other studies [7]. On a radar simulation program, **SRIM**, we are currently reaching speeds 7.9 times that of the CRAY X-MP (8.5 nsec clock cycle, using the SSD, single processor) and 6.8 times that of the CRAY Y-MP. This advantage results primarily from the non-vector nature of the application and the relative high speed of the NCUBE on scalar memory references and branching. Since either CRAY is 12–20 times more expensive than the NCUBE/ten, we can estimate that our parallel version is at least 15 to 25 times more cost-effective than a CRAY version using all four or eight of its processors.

---

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

The **SRIM** application is one of importance to Sandia’s mission. It permits a user to preview the appearance of an object detected with synthetic aperture radar (SAR), by varying properties of the view (azimuth, elevation, ray firing resolution) and of the radar (wavelength, transfer function, polarization, clutter levels, system noise levels, output sampling). **SRIM** is based on ray-tracing and SAR modeling principles, some of which are similar to those used in optical scene generation. Hundreds of hours are consumed on supercomputers each year by the **SRIM** application.

Successful restructuring of **SRIM** required that parallel application development draw upon new and ongoing research in four major areas:

- heterogeneous programming
- program decomposition
- dynamic, asynchronous load balancing
- parallel graphics and other input/output

These four areas are the focus of this paper. The resulting performance of parallel **SRIM** represents major performance and cost advantages of the massively parallel approach over conventional vector supercomputers on a real application.

In § 2, we examine background issues. These include the traditional approach to ray tracing, characteristics of the third-party simulation package, **SRIM** [1], salient features of our 1024-processor ensemble, and how those features fit the task of radar simulation. We then explain in § 3 the strategy for decomposing the program—both spatially and temporally—with particular attention to distributed memory and control. The performance is then given in § 4, and compared to that of representative minicomputers and vector supercomputers.

## 2. BACKGROUND DISCUSSION

### 2.1 TRADITIONAL RAY TRACING AND PARALLELISM

The ray-tracing technique has its origin at the Mathematical Applications Group, Inc. (MAGI) in the mid-1960s [12, 13]. Ray tracing was first regarded as a realistic but intractable method of rendering images, since the computations could take hours to days per scene using the computers of that era. As computers became faster, use of the ray tracing technique increased.

In optical ray tracing, a light source and a set of objects are stored as geometrical descriptions. Rays from the viewer are generated on a grid or randomly. The rays are tested for intersection against the entire set of objects as shown schematically in Figure 1. A “bounce” is recorded for each ray that experiences at least one intersection. That is, the first intersection is found by sorting and treated as a generator of another ray. The set of records associated with these ray bounces is known as the *ray history*. The process repeats until the ray leaves the scene or the number of bounces exceeds a predetermined threshold. The effect (line-of-sight returns) of the original ray on the viewer can then be established.

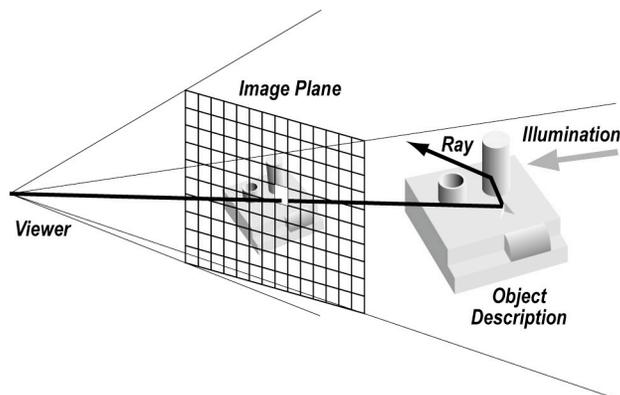


Figure 1. Traditional Ray Tracing (Optical)

The ray firing events are operationally independent, so the problem appears to decompose so completely that no interprocessor communication is necessary. It is a common fallacy that putting such an application on a parallel processor is a trivial undertaking. Although the rays are independent, the object descriptions are not. Also, the work needed for each ray is uneven and the resulting work imbalance cannot be predicted without actually performing the computation.

Therefore, some form of dynamic, asynchronous load balancing is needed to use multiple processors efficiently. A common approach, inadequate for our 1024-processor ensemble, is the *master-slave model*, with the host delegating tasks from a queue to processors that report in as idle. Figure 2 illustrates this scheme. On a hypercube, a logarithmic-height spanning tree provides the required star-shaped topology. This is handled by the hypercube operating system and need not be visible to the programmer.

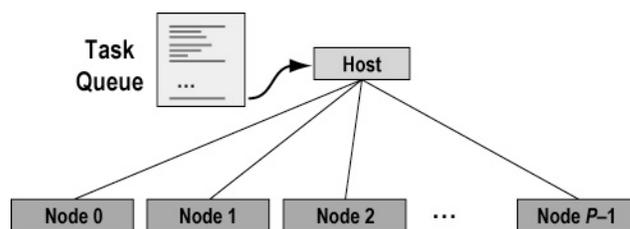


Figure 2. Master-Slave Load Balancer on a Hypercube

Simple master-slave schemes do not scale with the number of slave processors. Suppose, as in our hypercube, that the host uses at least two milliseconds to receive even the shortest message from a node. If all 1024 nodes send just a two-byte message indicating their work status to the host, the host must spend a total of two seconds just receiving all the messages. If the average node task takes less than 2 sec, the host will be overwhelmed by its management responsibilities and the nodes will often be in an idle state waiting for the host for new assignments. A much better method is described in § 3.3 below.

An entirely different approach is to divide *space* rather than the set of fired rays, which potentially removes the need for every processor to be able to access the entire object database. This approach creates severe load imbalance near the light sources and high interprocessor communication cost; see [15].

## 2.2 PROBLEM SCALING

The algorithmic cost of a ray-tracing algorithm depends primarily on:

- The number of rays fired
- The maximum allowed number of bounces
- The number of objects in the scene

The increase in computational effort and the amount of ray history data is approximately linear in the number of rays fired. However, memory requirements are unaffected by the number of rays, because ray history data resides in either an SSD scratch file on the CRAY or interprocessor messages on the hypercube.

The maximum allowed number of bounces  $b$  can affect computational effort, since every bounce requires testing for intersection with every object in the scene. In some optical ray tracing models, rays can split as the result of reflections/refractions, causing work to increase geometrically with maximum bounces. Here, we increase work only arithmetically. As with the number of rays fired, varying  $b$  has little effect on storage requirements. Therefore, it is possible, in principle, to compare elapsed times on one processor and a thousand processors for a fixed-size problem [9], since scaling the problem does not result in additional storage. These scaling properties contrast with those of the applications presented in [9], in which increasing ensemble size leads naturally to an increase in the number of degrees of freedom in the problem.

## 2.3 THE SRIM RADAR SIMULATOR

The **SRIM** program has about 30,000 lines (about 150 subroutines) of Fortran 77 source text. The two most time-consuming phases are **GIFT** (18,000 lines), and **RADSIM** (12,000 lines). **GIFT** reads in files from disk describing the scene geometry, and viewpoint information from the user via a console, and then computes “ray histories,” the geometrical paths traced by each ray fired from the emanation plane. It has its roots in the original **MAGI** program [4], and uses combinatorial solid geometry to represent objects as Boolean operations on various geometric solids. In contrast to many other ray tracing programs that have been demonstrated on parallel machines, **GIFT** supports many primitive geometrical element types:

<i>Box</i>	<i>4 to 8 Vertex Polyhedron</i>
<i>Sphere</i>	<i>Elliptic Hyperboloid</i>
<i>General Ellipsoid</i>	<i>Hyperbolic Cylinder</i>
<i>Truncated General Cone</i>	<i>Right Circular Cylinder</i>
<i>Truncated Elliptical Cone</i>	<i>Rectangular Parallelepiped</i>
<i>Half-Space</i>	<i>Right Elliptic Cylinder</i>
<i>Truncated Right Cone</i>	<i>Arbitrary Polyhedron</i>
<i>Circular Torus</i>	<i>Parabolic Cylinder</i>
<i>Elliptical Paraboloid</i>	<i>Right-Angle Wedge</i>
<i>Arbitrary Wedge</i>	<i>Elliptical Torus</i>

This large number of object types is a major reason for the large size of the serial **GIFT** executable program.

To reduce the need to compare rays with every object in the database, **GIFT** makes use of nested bounding boxes that greatly reduce the computational complexity of the search for intersections. For the price of additional objects in the database, the cost of the comparison is reducible from linear to logarithmic complexity in the number of objects.

A subset of **GIFT** converts a text file describing the geometries to a binary file that is then faster to process. However, it is common to convert a text file to binary once for every several hundred simulated images, much as one might compile a program once for every several hundred production runs. Therefore, we have not included the conversion routine in either the parallelization effort (§ 3) or performance measurement (§ 4).

The ray histories generated by serial **GIFT** are also stored on disk (SSD on the CRAY version). **RADSIM**, a separately loaded program, then uses this ray history file to simulate the effect on a viewing plane of radar following those paths (see Figure 3.) By separating the simulation into these two parts, the user can experiment with different radar properties without recomputing the paths taken by the rays.

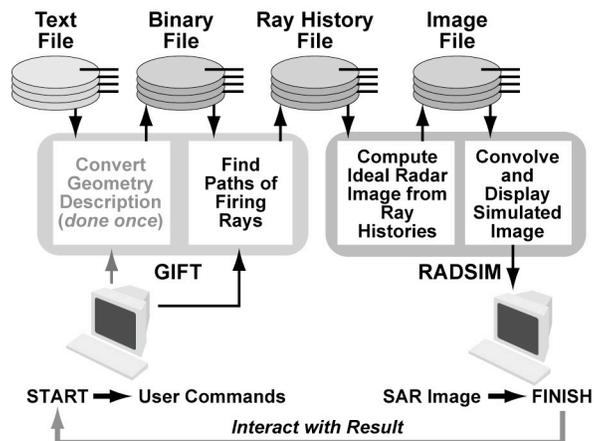


Figure 3. Serial SRIM Flowchart

Unlike optical ray tracing, *every* object intersection visible from the viewing plane contributes to the image, and the contributions add as complex numbers instead of simply as real

intensities. That is, interference between waves is a significant effect. Also unlike optical ray tracing, the ray-traced image must be convolved with the original radar signal (usually by FFT methods). Another difference from optical ray tracing is that dozens of bounces might be needed to obtain an accurate image. While optical ray tracing algorithms might reduce computation by setting a maximum of three to ten bounces, synthetic aperture radar often undergoes over 30 bounces without significant attenuation.

Perhaps the most important qualitative difference between SAR and optical imagery is that the  $x$ -axis does not represent viewing azimuth in SAR images. Instead, it represents the length of the path from the radar source to the target. Thus, the SAR image may bear little resemblance to the optical image. This is one reason users of SAR (as well as systems for automatic target recognition) must be trained to recognize objects; they cannot rely on simple correspondence with the optical appearance of targets. As a result, large databases of SAR images are needed for every object of interest, which poses a daunting computational task.

The **RADSIM** approach is described in [1]. Other methods for predicting radar signatures are described in [5]. The so-called *dipole approximation method*, or *method of moments*, takes into account the fact that conductors struck by electromagnetic radiation become radiation sources themselves, creating a large set of fully-coupled equations. *Radiosity methods* [3] for predicting optical images also lead to fully coupled equations. Although the dipole and radiosity methods are in principle more accurate than ray tracing, the cost of solving the resulting dense matrix restricts the method of moments to relatively coarse descriptions of scenes and objects.

### 2.3 THE 1024-PROCESSOR ENSEMBLE AND SRIM ISSUES

The NCUBE/ten is the largest Multiple-Instruction Multiple-Data (MIMD) machine currently available commercially. The use of processors specifically designed as hypercube nodes allows ensembles of up to 1024 processors to fit in a single cabinet. Each node consists of the processor chip and six 1-Mbit memory chips (512 Kbytes plus error correction code) [14]. Each processor chip contains a complete processor, 11 bidirectional communications channels, and an error-correcting memory interface. The processor architecture resembles that of a VAX-11/780 with floating-point accelerator, and can independently run a complete problem of practical size. Both 32-bit and 64-bit IEEE floating-point arithmetic are integral to the chip and to the instruction set; currently, **SRIM** primarily requires 32-bit arithmetic.

Since there is no vector hardware in the NCUBE, there is no penalty for irregular, scalar computations. This is appropriate for the **SRIM** application, since vector operations play only a minor role in its present set of algorithms. Also, much of the time in

**SRIM** is spent doing scalar memory references and testing for the intersection of lines with quadratic surfaces, which involves scalar branch, square root, and divide operations. Since each NCUBE processor takes only 0.27 microsecond for a scalar memory reference, 1 microsecond for a branch, and 5 microseconds for a square root or divide, the composite throughput of the ensemble on those operations is 50 to 550 times that of a single CRAY X-MP processor (8.5 nanosecond clock).

All memory is distributed in our present hypercube environment. Information is shared between processors by explicit communications across channels (as opposed to the shared memory approach of storing data in a common memory). This is both the greatest strength and the greatest weakness of ensemble computers: The multiple independent paths to memory provide very high bandwidth at low cost, but global data structures must either be duplicated on every node or explicitly distributed and managed.

Specifically, there is no way a single geometry description can be placed where it can be transparently accessed by every processor. Each processor has 512 Kbytes of memory, of which about 465 Kbytes is available to the user. However, 465 Kbytes is not sufficient for both the **GIFT** executable program (110 Kbytes) and its database, for the most complicated models of interest to our simulation efforts. It is also insufficient to hold both the **RADSIM** executable program (51 Kbytes) and a high-resolution output image. (The memory restrictions will be eased considerably on the next generation of hypercube hardware.) Our current parallel implementation supports a database of about 1100 primitive solids and radar images of 200 by 200 pixels (each pixel created by an 8-byte complex number). Two areas of current research are distribution of the object database and the radar image, which would remove the above limitations at the cost of more interprocessor communication.

The hypercube provides adequate external bandwidth to avoid I/O bottlenecks on the **SRIM** application (up to 9 Mbytes/sec per I/O device, including software cost [2]). It is also worthy of note that for the 1024-processor ensemble, the lower 512 processors communicate to the higher 512 processors with a composite bandwidth of over 500 Mbytes/sec. This means that we can divide the program into two parts, each running on half the ensemble, and have fast internal communication from one part to the other.

The host may dispense various programs to different subsets of the ensemble, decomposing the job into parts much like the data is decomposed into subdomains. Duplication of the entire program on every node, like duplication of data structures on every node, is an impractical luxury when the program and data consume many megabytes of storage. This use of *heterogeneous programming* reduces program memory requirements and eliminates scratch file I/O (§ 3.1). With respect to program memory, heterogeneous programming is to parallel ensembles

what *overlay* capability is to serial processors; heterogeneous programming is spatial, whereas overlays are temporal.

One more aspect of the system that is exploited is the scalability of distributed memory computers, which makes possible personal, four-processor versions of the hypercube hosted by a PC-AT compatible. We strive to maintain scalability because the performance of personal systems on **SRIM** is sufficient for low-resolution studies and geometry file setup and validation. Also, since both host and node are binary compatible with their counterparts in the 1024-processor system, much of the purely mechanical effort of program conversion was done on personal systems.

### 3. PARALLELIZATION STRATEGY

To date we have used four general techniques to make **SRIM** efficient on a parallel ensemble: heterogeneous ensemble use (reducing disk I/O), program decomposition (reducing program storage), hierarchical task assignment (balancing workloads), and parallel graphics (reducing output time).

#### 3.1 HETEROGENEOUS ENSEMBLE USE

As remarked above, traced rays are independent in optical ray tracing. In the **SRIM** radar simulator, the task of tracing any one ray is broken further into the tasks of computing the ray history (**GIFT**) and computing the effect of that ray on the image (**RADSIM**). In optical ray tracing, the latter task is simple, but in radar simulation it can be as compute-intensive as the former.

The serial version completes all ray histories before it begins to compute the effect of the rays on the image. We extend the parallelism within these tasks by pipelining them (Figure 4); results from nodes running **GIFT** are fed directly to the nodes simultaneously running **RADSIM**, with buffering to reduce the number of messages and hence communication startup time. We divide the parallel ensemble in half to approximately balance the load between program parts; depending on the maximum number of bounces and the number of objects, processing times for **GIFT** are within a factor of two of processing times for **RADSIM**. Why is this heterogeneous approach better than running **GIFT** on all nodes, then **RADSIM** on all nodes, keeping the cube homogeneous in function?

First, it eliminates generation of the ray history file, thereby reducing time for external I/O. Although the ray history file in principle allows experiments with different radar types, practical experience shows that it is always treated as a scratch file. The human interaction, or the preparation of databases of views of an object, invariably involves modifications to the *viewing angle* or *object geometry* instead of just the radar properties. Hence, only unused functionality is lost, and we avoid both the time and the storage needs associated with large (~200 Mbyte) temporary disk files. (Applying this technique to the CRAY would have only a small effect, since scratch file I/O on the SSD takes only 3% of the total elapsed time.)

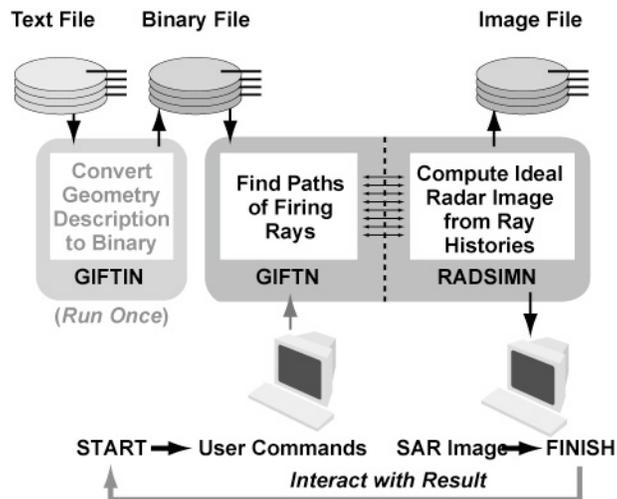


Figure 4. Parallel SRIM Flowchart

Secondly, it doubles the *grain size* per processor. By using half the ensemble in each phase, each node has twice as many tasks to do. This approximately halves the percent of time spent in parallel overhead (*e.g.* graphics, disk I/O) in each program part. Furthermore, by having both program parts in the computer simultaneously, we eliminate the serial cost of reloading each program part.

#### 3.2. PROGRAM DECOMPOSITION

As shown in Figure 3 above, the original serial version of **SRIM** exists in two load modules that each divide into two parts. Since every additional byte of executable program is one less byte for geometric descriptions, we divide **SRIM** into phases to minimize the size of the program parts. Some of these phases are concurrent (pipelined), while others remain sequential.

First, we place I/O management functions, including all user interaction, on the host. Most of these functions, except error reporting, occur in the topmost levels of the call trees of both **GIFT** and **RADSIM**. They include all queries to the user regarding what geometry file to use, the viewing angle, whether to trace the ground plane, etc., as well as opening, reading, writing, and closing of disk files. Much of the program (83500 bytes of executable program) was eventually moved to the host, leaving the computationally intensive routines on the much faster hypercube nodes, and freeing up their memory space for more detailed geometry description. Both **GIFT** and **RADSIM** host drivers were combined into a single program. This was the most time-consuming part of the conversion of **SRIM** to run in parallel. To illustrate the host-node program decomposition, Figure 5 shows the original call tree for **RADSIM**. (The call tree for **GIFT** is too large to show here.)

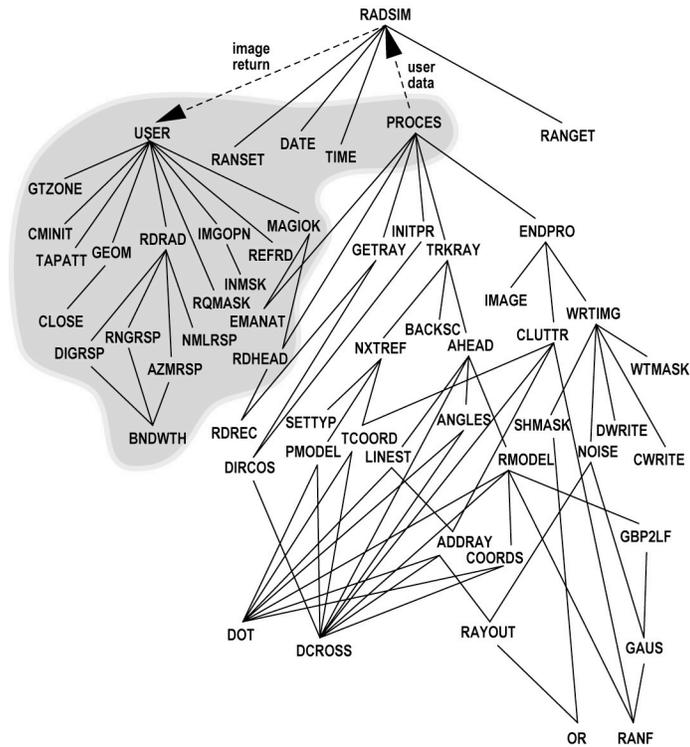


Figure 5. Call Tree for Radar Image Generation

The parts of the program run on the host are shown in the shaded region in Figure 5. The others run on each **RADSIM** hypercube node. The dashed lines show messages between hypercube nodes and the host program. One such path is the communication of the computed radar image from **RADSIM** to subroutine **USER** on the host. The other is the **PROCES** subroutine, which has been divided into host and node routines that communicate program input by messages. Routines **RDHEAD** and **EMANAT** in the host portion are also needed by the node; these were replicated in the node program to eliminate the calling tree connection. The **GIFT** program contained many subroutines that had to be replicated on both host and node in a rather arduous process of separating host and node functionality.

Error reporting can happen at any level in the program, a common practice in Fortran. To remove the need for computationally-intensive subroutines to communicate with the user, errors are reported by sending an error number with a special message type to the host, which waits for such messages after execution of the **GIFT** and **RADSIM** programs begins.

Next, the conversion of text to binary geometry files was broken into a separate program from **GIFT**. We use only the host to create the binary geometry file. The conversion program, **CVT**, uses many subroutines of **GIFT**, so this decomposition only reduces the size of the executable **GIFT** program from 131 Kbytes to 110 Kbytes.

Similarly, the convolution routines in **RADSIM** are treated separately. Much of the user interaction involves unconvolved images and, unlike the ray history file, the unconvolved image

files are both relatively small and repeatedly used thereafter. This reduces the size of the **RADSIM** executable program, allowing more space for the final image data in each node running **RADSIM**.

One option in **SRIM** is to produce an optical image instead of a radar image. Since this option is mainly for previewing the object, ray tracing is restricted to one bounce for greater speed. Although optical and the radar processing are similar and use many of the same routines, only the executable program associated with one option needs to be loaded into processors. The user sets the desired option via host queries before the hypercube is loaded, so breaking the program into separate versions for optical and radar processing further reduces the size of the node executable program.

Finally, we wish to display the radar images quickly on a graphics monitor. This means writing a program for the I/O processors on the NCUBE RT Graphics System [14]. These I/O processors are identical to the hypercube processors except they have only 128 Kbytes of local memory. Although library routines for moving images to the graphics monitor are available, we find that we get a 10- to 30-fold increase in performance [2] by crafting our own using specific information about the parallel graphics hardware.

Figure 6 summarizes the division of the hypercube into subsets performing different functions. There are currently eight different functions in our parallel version of **SRIM**:

- Host program
- Host I/O node program (the **VORTEX** operating system; no user program)
- Manager node program (currently part of **GIFT**)
- **GIFT** node program
- Image node program (currently part of **RADSIM**)
- **RADSIM** node program
- Graphics node program
- Graphics controller program (loaded by user, supplied by the vendor)

We have not fully optimized the layout shown in Figure 6. For example, it may be advantageous to reverse the mapping of the **GIFT** Manager program and **RADSIM** Image program to the nodes, to place the **RADSIM** nodes one communication step closer [2, 15] to the graphics nodes. Also, a significant performance increase (as much as 15%) should be possible by driving disk I/O through the parallel disk system instead of the host disk, in the same manner that graphics data is now handled by the parallel graphics system. Successful use of parallel disk I/O requires that we complete software which renders the parallel disk system transparent to the casual user, so that portability is maintained between the 1024-processor system and the four-processor development systems.

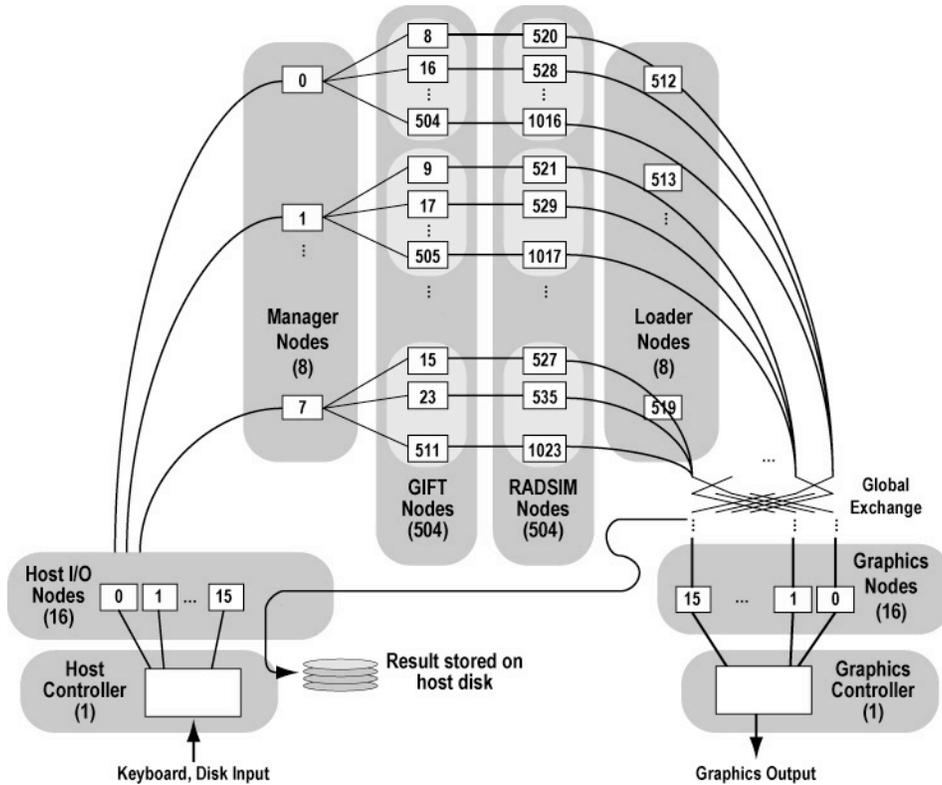


Figure 6. Multiple Functions of Hypercube Subsets on Parallel **SRIM**

We have extended system software to facilitate fast loading of the **GIFT-RADSIM** pair of cooperating node executable programs into subcubes of an allocated subcube. This fast loading software can be used for any application requiring multiple programs on the hypercube. If operating system support for this software is improved, one could load an arbitrary number of programs into an allocated hypercube in logarithmic time [9]. Separate **MANAGER** and **IMAGE** programs could be created, in addition to possible additional functions, such as a separate **CLUTTR** program (see Figure 5) to handle the clutter-fill function embedded in **RADSIM**.

The most fundamental (and unresolved) research issue in parallel load balance is posed by the allocation of processors to **GIFT** and **RADSIM**. For example, one may assign twice as many processors to **GIFT** as to **RADSIM**, to achieve high parallel efficiency on all processors. Two factors must be considered here: (1) the routing of ray history information between sets of nodes so as to maintain load balance, and (2) the dynamics of the **GIFT-RADSIM** load balance; that is, a truly dynamic balance would require one to switch executable programs on some processors during the course of a simulation.

### 3.3 HIERARCHICAL TASK ASSIGNMENT

Conventional master-slave dynamic load balancing has a bottleneck at the host for large ensembles (§ 2). To relieve the host of the burden of managing the workloads of the hypercube nodes, we instead use a subset of the hypercube itself as managers (Figure 7). The host statically assigns a task queue to each manager. Each manager then delegates tasks from its queue to workers that report in as idle, as in the master-slave model of § 2.1. If the host-manager task queue allocation were dynamic, this would be a *two-tier master-slave* system. One can envision the need for even more management layers as the number of processors increases beyond 1024.

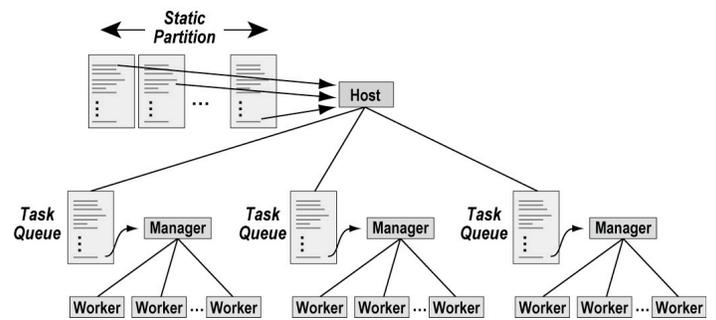


Figure 7. Manager-**GIFT** Load Balancer

Table 1 shows **SRIM** performance in elapsed time as a function of the number of manager nodes. Two geometry models ("slicy," shown in Figure 1, and an M60 tank) and two program

options (**rad** (radar) and **ovr** (optical view in slant plane)) were used. Managers assigned 4-by-4 blocks of rays in all cases. Firing ray patterns ranged from 187 by 275 to 740 by 1100.

**Table 1**

Performance for Various Manager Node Allocations on 1024 Processors. Entries are Elapsed Time in Seconds. Work units are 4 by 4 ray groups for dynamic assignment, ray columns for static (no manager) assignment. **Bolded** entries show the optimum range for each case.

Manager Nodes	Worker-Manager Ratio	slicy		m60	
		ovr 474 by 639	rad 945 by 1275	ovr 187 by 275	rad 740 by 1100
0	—	27	82	42	172
1	511:1	99	301	35	181
2	255:1	41	77	<b>32</b>	134
4	127:1	27	<b>72</b>	<b>32</b>	<b>129</b>
8	63:1	<b>23</b>	<b>72</b>	<b>31</b>	131
16	31:1	<b>23</b>	<b>72</b>	<b>32</b>	131
32	15:1	<b>24</b>	<b>75</b>	<b>31</b>	132
64	7:1	25	77	<b>32</b>	139
128	3:1	25	89	33	160
256	1:1	26	120	34	242

To determine the best ratio of worker nodes to manager nodes, we experimented with 1 to 256 managers. Table 1 shows, across a range of data sets and various viewing options, manager nodes saturate in processing work requests for ratios greater than 127:1. Managers are relatively idle for ratios less than 15:1, suggesting that some managers would be put to better use as workers. The need for managers increases with model size (the M60 is larger than "slicy") and image size. Four to 16 manager nodes suffice in the parallel version of **SRIM** for the representative model and image sizes presented here.

### 3.4 GRAPHICS OUTPUT

The **RADSIM** nodes each contain an array capable of storing the entire final image, initially set to zero (blank). As ray histories are processed, rays contribute to the image in a sparse manner. By replicating the entire output screen on every **RADSIM** node, we eliminate interprocessor communications until all rays have been processed. Once all **RADSIM** nodes have been sent messages indicating there are no more rays to process, they participate in a *global exchange* process [9, 16] to coalesce the independent sparse images into the final summed image. This takes  $O(\lg P)$  steps, where  $P$  is the number of **RADSIM** processors. After the global exchange, each **RADSIM** node has a subset of the computed radar signature, represented as one complex pair of 32-bit real numbers per pixel. The distributed image is then scaled to pixels for transmission in parallel to an I/O channel of the hypercube.

We have found that high-speed graphics on our current massively-parallel system required major effort [2]. The graphics board organizes the 16-bit memory paths of its 16 I/O processors into a single 256-bit path into the frame buffer. The 256 bits represent a row of 32 pixels of eight bits each. The display is tiled with these 32-pixel rows, which gives each I/O processor responsibility for a set of columns on the display. Each column created by **RADSIM** is routed to a **GIFT** node that is a nearest neighbor of the I/O processor responsible for that column. The **GIFT** node sends the data to the I/O processor, with software handshaking to prevent overloading the I/O processor. The net effect of this complicated scheme is the reliable gathering and display of a complete simulated radar image in less than one second. We have built a library of these graphics algorithms to relieve other hypercube users from dealing with the system complexity.

### 4. PERFORMANCE

The following performance data are based on a single problem: the M60 tank composed of 1109 primitive solids, with 0.82 million rays fired and a 200 by 200 final image. This problem is representative of current production calculations on Sandia's CRAY X-MP. The problem is identical to **m60.rad** in Table 1, except for finer-grained ray assignments by the dynamic load balancer (2-by-2 blocks of rays, found to be optimal for this problem).

Table 2 compares the performance of **SRIM** on various machines: a representative minicomputer, traditional supercomputers, and a massively parallel supercomputer. The performance range is almost 300 to 1. The elapsed time for a complete simulation is compared in each case, including all input and output. The NCUBE/ten time includes four seconds for real-time graphics, which is unavailable on the other systems. Some objects of interest will require several times as much elapsed time per image.

**Table 2**

Performance Summary for Various Machines.

	Elapsed Time, seconds	Elapsed Time, normalized	Mflop/s
VAX 11/780 + FPA <sup>†</sup>	35383	285	0.15
CRAY X-MP <sup>††</sup>	981	7.9	5.3
CRAY Y-MP <sup>††</sup>	843	6.8	6.2
NCUBE/ten hypercube	124	1.0	42

<sup>†</sup>VAX times are for dedicated runs with a large working set.

<sup>††</sup>CRAY times are CPU, due to unavailability of dedicated time.

Previous comparisons of applications on the NCUBE with their serial CRAY X-MP production counterparts [9] had shown the parallel ensemble to be 1.6 to 4 times the speed of the vector supercomputer. Here we see the effect of irregularity in the computation (memory references, branches; see § 2.4). It is

difficult to count operations for this application, but we can infer them using a CRAY X-MP hardware monitor and estimate Mflop/s.

This example shows the massively parallel hypercube to be superior in absolute performance to the traditional serial approach for the **SRIM** application. It is more difficult to assess relative cost-efficiency because of the wide error margins in computer prices, but we can assert that the CRAY computers are 15 to 25 times more expensive than the NCUBE on this application. Using acquisition prices only, we estimate that the NCUBE/ten costs \$1.5 million, a CRAY X-MP/416 \$18 million (including SSD, 8.5 nanosecond clock cycle), and CRAY Y-MP \$30 million. If we generously assume that all of the processors in either CRAY can be used independently with 100% efficiency, then the cost performance of the NCUBE versus the CRAY X-MP is

$$(\$18 \text{ M})/(\$1.5 \text{ M}) \times (7.9 \text{ speedup vs. a CRAY CPU}) / (4 \text{ CPUs}) \\ = \mathbf{23.7 \text{ times}}$$

and the cost performance of the NCUBE versus the CRAY Y-MP is

$$(\$30 \text{ M})/(\$1.5 \text{ M}) \times (6.8 \text{ speedup vs. a CRAY CPU}) / (8 \text{ CPUs}) \\ = \mathbf{17 \text{ times}}$$

Such advantages have been predicted for parallel computers for years [17]. To our knowledge, this is the first published result demonstrating such a large cost-performance advantage on a supercomputer application.

We now analyze the parallel performance. Traditional analysis of parallel speedup involves varying the number of processors. This type of analysis is simplistic in the case of **SRIM** because we have at least six different sets of processors to increase or decrease in number. We have already presented one breakdown (Table 1) that shows the result of varying the number of Manager nodes, but it is difficult and inappropriate to apply such ingrained concepts as "serial component" and "communication overhead" to this system of cooperating sets of processors.

One approach to a performance-versus-processors evaluation is to vary the number of **GIFT-RADSIM** pairs, keeping all the other sets of processors and the set of rays fixed in size. Table 3 shows times for 64 to 512 pairs, for both unbalanced workloads (no managers) and balancing by a 63:1 ratio of managers (taken from the set running **GIFT**) to **GIFT-RADSIM** pairs. We note that the "best serial algorithm" to which we compare performance is itself somewhat parallel, since it requires the host, a host I/O node, a **GIFT** node, a **RADSIM** node, and the graphics system (composed of some 20 processors)! For this "skeleton crew" of processors, we have measured an elapsed time of 42980 seconds. If we assume **GIFT** is the limiting part of the computation, then Table 3 implies that the full 1024-processor ensemble is being used with about 68% efficiency. We note that the 68% efficiency achieved here on a

fixed-sized problem exceeds the efficiencies of 49% to 62% presented in [9] for three application programs on 1024 processors. Finally, dynamic load balancing is not needed for fewer than 64 **GIFT-RADSIM** pairs in this case.

**Table 3**  
**SRIM** Elapsed Time, T, and Efficiency, E, versus **GIFT-RADSIM** pairs

<b>GIFT-RADSIM</b> Processor Pairs	<i>No Manager</i>		<i>Managers</i>		
	T (sec)	E (%)	Number of Managers	T (sec)	E (%)
64	751	89	1	750	90
128	414	81	2	393	85
256	244	69	4	220	76
512	172	49	8	124	68

Lastly, we break down the time by task in Table 4. A total of 19 seconds is needed to initialize the system and the simulation, and 12 seconds is needed for output operations.

**Table 4**  
Timing Breakdown by Task (Time in Seconds)

	Host Active	Nodes Active	Elapsed Time
<i>Initialization Phase</i> (15% of elapsed time)			
Read initial input, open cube	2	—	2
Load <b>GIFT</b> program	4	—	4
Load <b>RADSIM</b> program	1	1	1
Load geometry file	8	8	8
Read, load other <b>GIFT</b> Data	< 1	< 1	< 1
Read, load other <b>RADSIM</b> data	< 1	< 1	< 1
Initialize graphics	—	4	4
<i>Computation Phase</i> (75% of elapsed time)			
Compute/use ray histories	—	92	92
Empty ray history pipeline ( <b>RADSIM</b> )	—	< 1	< 1
Global exchange/sum image	—	1	1
<i>Output Phase</i> (10% of elapsed time)			
Display image	—	< 1	< 1
Save image on disk	12	12	12
Close cube	< 1	—	< 1
<i>Totals:</i>	27	118	124

The results of Table 4 pinpoint two areas for further improvement in the parallel **SRIM** application. First, we expect to reduce the 12 seconds of disk output and the 8 seconds for loading the geometry file to at most 3 seconds total through use of the parallel disk system. Second, we have measured one second (out of 92) of residual load imbalance within the **GIFT**

processors, which could be addressed by a dynamic allocation of task queues to managers by the host. Therefore, the total elapsed time could be reduced to 106 seconds using the current hardware. This would be 9.3 times faster than the CRAY X-MP, 8 times faster than the CRAY Y-MP, and would represent a parallel efficiency of 79% relative to the tow-processor version. Further improvements are possible when the **GIFT-RADSIM** load imbalance is measured and addressed. To further define the performance and cost-performance advantages using the latest hardware for both vector and massively-parallel supercomputers, we plan to compare the CRAY Y-MP results with results obtained on the new NCUBE 2 as soon as the latter is available.

## 5. SUMMARY

We believe the parallelization of SRIM is the first such effort to use heterogeneous programming, program decomposition, dynamic load balancing, and parallel graphics in concert. The work has pointed to further research opportunities in areas such as parallel disk I/O and decomposition of object databases and images. Furthermore, the parallel performance of **SRIM** represents a major advantage of the massively parallel approach over conventional vector supercomputers on a real application: 6.8 times a CRAY Y-MP processor and 7.9 times a CRAY X-MP processor in speed, coupled with a factor of 12 to 20 times the vector supercomputer in machine acquisition cost.

## ACKNOWLEDGMENTS

We thank Gary Montry for suggestions regarding parallel random number generation (cf. [8]) and heterogeneous program loading, Jim Tomkins for CRAY timings, Guylaine Pollock and Jim Tomkins for operation count estimates, and Gil Weigand for optimizing the load balancing parameters to increase performance. In addition, we thank Gerald Grafe and Ed Barsis for critical reviews of the paper.

---

## REFERENCES

- [1] C. L. ARNOLD JR., "SRIM User's Manual," *Environmental Research Institute of Michigan*, Ann Arbor, Michigan (Sept. 1987).
- [2] R. E. BENNER, "Parallel Graphics Algorithms on a 1024-Processor Hypercube," *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, to appear (Mar. 1989)
- [3] C. M. GORAL, K. E. TORRANCE, AND D. GREENBERG, "Modeling the Interaction of Light Between Diffuse Surfaces," *Computer Graphics*, **18** (3), ACM SIGGRAPH (1984), pp. 213–222.
- [4] P. C. DYKSTRA, "The BRL-CAD Package: An Overview," *Proceedings of the BRL-CAD Symposium '88* (Jun 1988).
- [5] P. C. DYKSTRA, "Future Directions in Predictive Radar Signatures," *Proceedings of the BRL-CAD Symposium '88* (Jun 1988).
- [6] G. C. FOX, ed., *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, ACM Press, New York (Jan 1988).
- [7] G. C. FOX, "1989—The Year of the Parallel Supercomputer," *Proceedings of the Fourth Conference on Hypercube Concurrent Processors, Vol. I*, Prentice Hall, Englewood Cliffs, New Jersey (1988).
- [8] G. C. FOX, M. A. JOHNSON, G. A. LYZENGA, S. W. OTTO, J. K. SALMON, AND D. W. WALKER, *Solving Problems on Concurrent Processors, Vol. I*, Prentice Hall, Englewood Cliffs, New Jersey (1988).
- [9] J.L. GUSTAFSON, G. R. MONTRY, AND R. E. BENNER, "Development of Parallel Methods for a 1024-Processor Hypercube," *SIAM Journal of Scientific and Statistical Computing*, **9** (1988), pp. 609–638.
- [10] M. T. HEATH, ed., *Hypercube Multiprocessors 1986*, SIAM, Philadelphia, (1986).
- [11] M. T. HEATH, ed., *Hypercube Multiprocessors 1987*, SIAM, Philadelphia, (1987).
- [12] MAGI, "A Geometric Description Technique Suitable for Computer Analysis of Both Nuclear and Conventional Vulnerability of Armored Military Vehicles," *MAGI Report 6701, AD847576* (August 1967).
- [13] M. J. MUUSS, "RT and REMRT—Shared Memory Parallel and Network Distributed Ray-Tracing Program," *USENIX: Proceeding of the Fourth Computer Graphics Workshop*, (Oct 1987).
- [14] NCUBE, "NCUBE Users Manual", Version 2.1, NCUBE Corporation, Beaverton, Oregon (1987).
- [15] T. PRIOL AND K. BOUATOUCH, "Experimenting with a Parallel Ray-Tracing Algorithm on a Hypercube Machine," *Rapports de Recherche N° 405*, INRIA-RENNES (May 1988).
- [16] Y. SAAD AND M. H. SCHULTZ, "Data Communication in Hypercubes," *Report YALEU/DCS/RR-428*, Yale University, (1985).
- [17] C. L. SEITZ, "The Cosmic Cube," *Communications of the ACM*, **28** (1985) pp. 22–23.