Multiprocessor Strategies for Ray-Tracing

Bob Boothe

Master's Project Report
Under Direction of
Professor Carlo H. Séquin

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
September 1989

ABSTRACT

Ray-tracing is often suggested as a problem which is well suited for execution on multiprocessors. It is characterized by having abundant parallelism, a very small sequential part, and aggravatingly long run-times. For simple well behaved scenes, linear speedup is easily achievable. However for realistic scenes which are typically both complex and non-uniformly distributed, parallel ray-tracing is a challenging problem. This thesis evaluates and compares implementations of a sophisticated ray-tracer on both shared memory and distributed memory machines.

Acknowledgements

I would like to thank my research advisor, Dr. Carlo H. Séquin, for suggesting that I compare real machines with real scenes, rather than studying parallel ray-tracing abstractly. I would also like to thank him for his support and patience as this research turned out to be more complex than anticipated. Wook Koh helped me get started using the parallel machines. Finally, I wish to thank Ziv Gigus, Henry Moreton, Seth Teller, and others for their abundant solicited and unsolicited advice.

This work was supported in part by Tektronix, Inc.

Table of Contents

1. Introduction

Ray-tracing is often suggested as a task which is well suited for execution on multiprocessors. Several characteristics make parallel ray-tracing attractive. First, a typical image consists of one million pixels which can be computed independently. Thus there is abundant high-level parallelism. Second, the sequential preprocessing portion of the program that reads the scene file and creates the data structures takes an insignificant fraction of the total total run time (as little as $\frac{1}{1000}$). Thus Amdahl's Law [Amd67] doesn't preclude achievement of very high speedups. Third, run-times can be several hours or even days. This is unacceptably long for most uses, so quicker and less realistic rendering techniques are used instead. An efficient parallel ray-tracer will allow ray-tracing to be more broadly used.

When trying to manually obtain parallelism from a program, higher-level large-grain parallelism is generally easiest to schedule. For computer animation, where a large number of frames are rendered, separate executions of the rendering program can be distributed around a network [Pet87]. When there is enough computing power on the network to provide the desired throughput, and when the long latency for the computation of a single image can be tolerated, frame parallelism is clearly the best choice. However if shorter latency for computing a single image is needed, parallelism at a lower level must be used, and a more tightly coupled multiprocessor will be needed. If a machine must be purchased for the primary purpose of rendering images, a multiprocessor will be the most cost effective choice since a multiprocessor will have the highest ratio of processors to memory. This paper evaluates implementations of a ray-tracer on both a shared memory multiprocessor (Sequent Balance) and a message passing multiprocessor (Intel iPSC Hypercube).

1.1. Background

Ray-tracing creates highly realistic images by tracing light rays in reverse from the eye point to the light sources (Figure 1.1). Realistic images are produced because ray-tracing is able to render shadows, reflections, refractions, and attenuation. For each pixel in an image a ray is traced from the eye point through the position of the pixel in the image plane to its first intersection with an object in the scene database. From this intersection multiple rays may be recursively sent out to test for shadows and to follow the

reflected and refracted light paths. Eventually all of the rays will have been 'absorbed' or have escaped the scene and their lighting contributions to the pixel can be summed. (On rare occasions rays can keep bouncing around the scene for a long time. At some recursion depth these rays are simply discarded with negligible effect on the final image.)

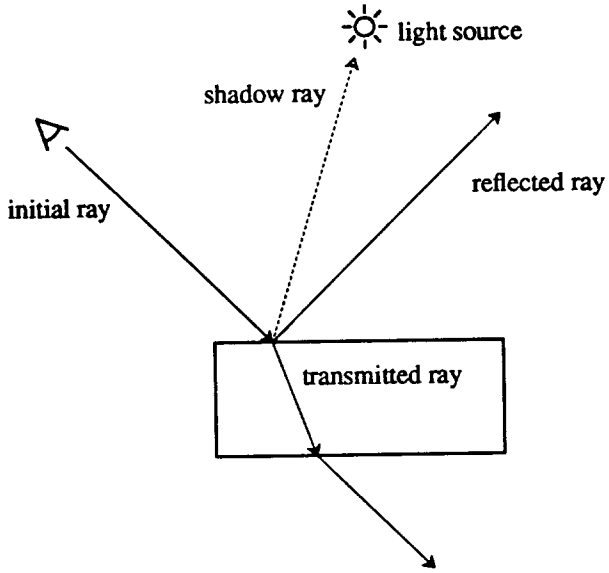

Figure 1.1. Types of Rays

Today's complex scenes typically have on the order of 10,000 primitives (generally polygons or spheres). As rendering machines become more powerful, scene complexity will undoubtably increase further. Finding the intersection of a ray with the closest object in the scene consumes the majority of the rendering time and therefore must be done efficiently. This is done through either subdividing or clustering of the object space so that only a few possible intersections need to be tested for each ray. My research is based on an efficient ray-tracing renderer written by Don Marsh [Mar87]. This renderer uses uniform spatial subdivision.

Uniform spatial subdivision is a technique where the region of space containing objects is mapped into a three-dimensional grid of cells. A complex scene will use on the order of one million cells. Each cell contains a list of the objects that intersect it. A ray can be traced through the scene by tracing its path through the cells of the subdivision. Typically only a few object intersection checks will be required to find the first intersection. Figure 1.2 shows a two-dimensional analog of how a simple scene could be divided

into an 8 by 12 cell subdivision. The ray shown would be traced through each of the shaded cells until it first hits the side of the house. In this case only a single object intersection test will be required since all of the intermediate cells are empty and the final cell contains only a single face.

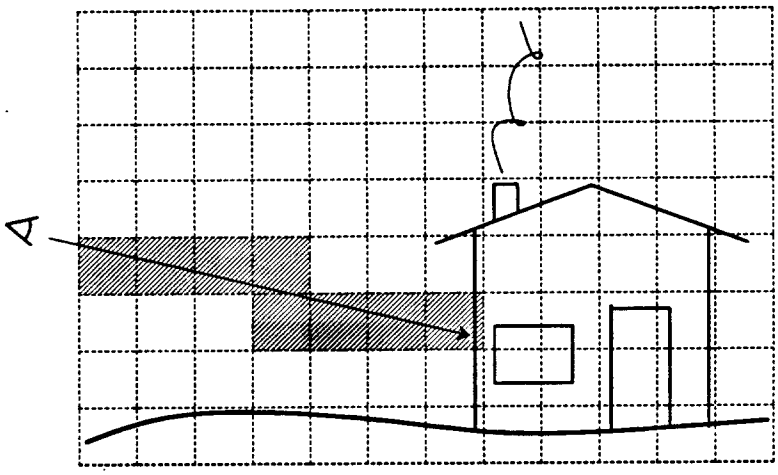

Figure 1.2. Cellular Subdivision

A 1000 by 1000 pixel image requires one million rays to be traced, all of which can be traced independently. Extra rays will be traced to perform antialiasing where edges occur in the image. These extra antialiasing rays are not independent of the original rays since antialiasing rays are only sent out when neighboring rays produce pixels of different colors. These dependencies complicate the parallelizing task, but parallelism is still abundant.

The independence of rays provides the natural point for parallelization. The huge parallelism and ray independence has led many people, such as parallel computer manufactures [Seq86], to suggest that ray-tracing is an ideal application for parallel processing. One difficulty is that different rays can take significantly different computation times. On a shared memory machine, scheduling of rays for load balancing purposes is the primary concern. Distributed memory machines have the additional problem of dynamically adjusting the data distribution to maintain a balanced load.

1.2. Previous Work

1.2.1. Ullner [Ull83]

Ullner suggested building a parallel ray-tracing machine with a two-dimensional grid of processors. Each processor is assigned a square section of the image and a prismatic slice of the database. The slices are aligned so that rays cast from the eye point can be traced within the originating processor. Shadow, reflection, and refraction rays which leave the region of the database contained within a processor can be sent to a neighboring processor. There are several potential problems.

A mechanism is needed to handle scenes where the section of the database assigned to a processor is too large for its local memory. Ullner suggested that processors should hold whatever they can and that when rays attempt to access data not present, the rays should be deferred until later when the data can be brought in. There are two important ideas here. First, the processors can manage their local memory as a cache of the the scene database, and second, rays can be traced out of order. These ideas will re-occur in my research.

The data assignment is optimized so that primary rays (the rays originating from the eye) can be traced entirely within a single processor. But primary rays can account for as little as 2% of the total rays. The other rays can be assumed to be oriented randomly, and therefore most of them will have to be sent as messages to neighboring processors. Often they won't hit anything and will be sent all the way to the edge of the database. Many processor boundaries will be crossed in the process. Message traffic will be very high and may be a significant design challenge for the machine. A three-dimensional grid of processors and corresponding cells in space might be better since it will minimize sub-volume surface area and will therefore minimize the total message traffic when rays are oriented randomly.

Load balancing will be poor since some sections of an image will be much easier to compute than others. For example, sometimes a corner of the image will be entirely sky. No objects will occur in it at all. Likewise, some other section may contain a complicated glass object that takes a substantial portion of the entire sequential run time. Some dynamic load balancing mechanism must be added for efficient utilization of the machine.

1.2.2. Dippe & Swensen [DiS84]

Dippe & Swensen [DiS84] wrote an early spatial subdivision paper. They suggests initially starting with a coarse uniform 3-D subdivision of space and dynamically moving the dividing walls between cells based on the amount of activity in each cell. The cells of the spatial subdivision can be assigned to separate processors arranged in a 3-D grid. The dynamic adjustments to the subdivision serve to balance the load among all the processors.

The dynamic cell modifications are rather complex and may be a significant overhead if load adjustments are frequent. Eight processors have to be involved and synchronized when a corner of a cell is moved. I suggest instead that a similar but simpler load balancing mechanism is to give each processor a collection of cells from a uniform spatial subdivision. To adjust the load distribution, a processor simply transfers a cell to one of its neighbors. Current subdivision research [Mar87] has found that fine subdivisions are needed anyways in order to minimize the number of ray-object intersection tests.

The authors assume very fast communication speed between processors but do not present results about the number of messages required. They mention that sufficient local memory is crucial to their architecture.

1.2.3. Cleary, Wyvill, Birtwistle, & Vatti [CWB86]

Cleary, Wyvill, Birtwistle, & Vatti [CWB86] also studied the subdivision of space onto a grid of processors where rays are sent as messages between processors. They compared 2-D meshes to 3-D meshes and conclude that 2-D meshes are better, although speedup would be less than linear, especially when ten or more processors are used. Unfortunately their results are based on a static subdivision of a simple uniformly distributed scene. Whelan [Whe85] studied the data distribution of real scenes and concluded that studies based upon such simplified assumptions will invariable lead to incorrect results.

1.2.4. Nishimura, Ohno, Kawata, Shirakawa, & Omura [NOK83]

The LINKS-1 system [NOK83] was a 64 processor machine built to perform parallel rendering using both parallel and pipelined chains of processors. It did only ray-casting (primary rays from the eye point) and did not use any spatial subdivsion technique to reduce the number of intersection tests. The scene

complexity was limited because the entire scene was stored on each processor. The machine achieved 50% efficiency when using all 64 processors. The architecture was designed for processing independent tasks and can not provide the general communications necessary for a scene larger than the local memory of a processor.

1.2.5. AT&T Pixel Machine [Pot88]

The AT&T Pixel Machine [Pot88] was designed for executing parallel graphics programs. There are 64 independent processors each with 1/2Mb data storage. Their parallel ray-tracer achieves linear speedup for scenes in which the entire database fits on each processor.

1.2.6. Priol & Bouatouch [PrB88]

Priol & Bouatouch [PrB88] implemented a parallel ray-tracer on an Intel iPSC Hypercube. They wisely decided to avoid the many complexities involved in dynamic load balancing, and instead did an initial static scheduling based upon sub-sampling the image with a small number of rays. The image space and data were partitioned so that each processor would trace an equal number of rays. Their speedup on 32 processors was 11. This might have been improved if they had used a more accurate load metric than simply counting the number of rays. The computational load is a function of both the ray flux through a processor and the object density.

2. Shared Memory Multiprocessors

The referenced previous works all look at distributed memory machines rather than shared memory machines. Ray-tracing, however, is very well suited to shared memory multiprocessors. A large read only data base is shared among the processors, and little communication or synchronization is needed if implemented correctly. The only drawback of shared memory machines is that their sizes have been limited to on the order of ten to thirty processors. Some ray-traced images are so expensive that much larger machines may be desired.

This section discusses research done on a 12 processor Sequent Balance 21000 [Seq86]. This is a single-bus shared-memory multiprocessor, or multis [Bel85], based on $\frac{1}{2}$-MIP NS32032 processors with special hardware for fast locking and symmetric I/O [BKT87]. Figure 2.1 shows a typical shared memory machine.

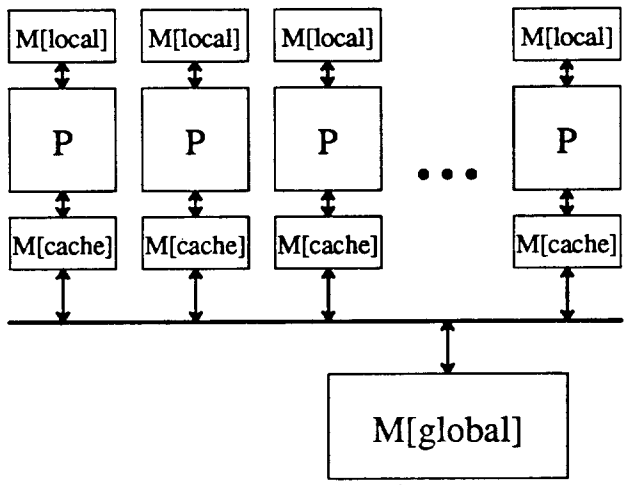

Figure 2.1. Shared Memory Architecture

Before I discuss the ray-tracing program, I will analyze a fractal program which is much simpler to parallelize and will serve as a basis for speedup comparison.

2.1. Fractals

Simple iterative procedures can create intricate fractal images as in Figure 2.2. The computation that creates these images uses an iterative calculation for each pixel that can take up to a few hundred floating

point operations. A typical image contains a million pixels, all of which can be computed independently. There isn't any large shared database as in ray-tracing. The only impediment to perfect parallel speedup is the non-uniformity of pixel calculation complexities. Some pixels can be finished in one iteration while others may take a hundred. If the number of processors used is orders of magnitude smaller than the number of pixels (which is the case for all machines except for very fine grained parallel machines) the non-uniformity can be dealt with in two ways. One way is to choose the set of pixels assigned to a processor so that the pixels are scattered across the image rather than clustered together [Sat85]. A second way is to dynamically schedule small regions of the image. Scattering a processor's pixels works because with high probability the distribution of pixel complexities will match the distribution for the entire image, and thus all the processors will receive the same load. Dynamic scheduling works because processors that get easy jobs can be given enough extra jobs that the total work is balanced.

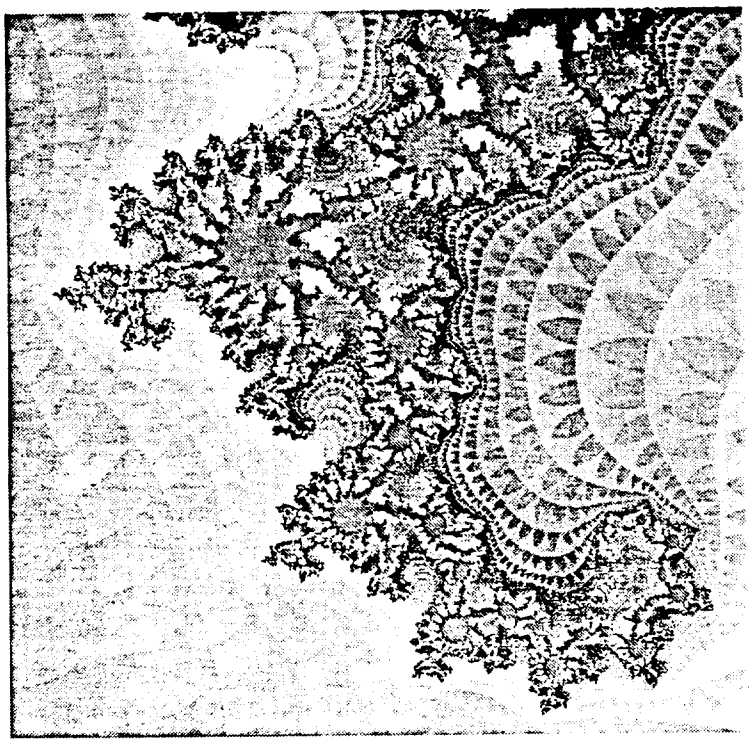

Figure 2.2. Typical Fractal Image of Mandelbrot Set

One reason for preferring dynamic scheduling is that a processor can be assigned a group of neighboring pixels. The coherence of neighboring pixels may be exploited explicitly if possible, or benefits may

result from data reference locality effects of the memory system. This argument doesn't matter for fractals because fractal programs don't take advantage of coherence and don't reference any data structures. For ray-tracing, coherence and locality are important.

On a machine with an interleaved-distributed frame buffer, static scheduling is the best choice, since it balances the load and doesn't require any communication of results. On the Sequent, where the resulting image is written to a file in scanline order, one process does all of the output to the file and only computes pixels of the image when there aren't any pixels ready for output. Dynamic scheduling automatically gives the process doing the output an appropriately diminished computational load. Dynamic scheduling of regions starting from the top of the image makes ordering the output easy and allows output to be done concurrently with image computations. For these reasons I have chosen to use dynamic scheduling.

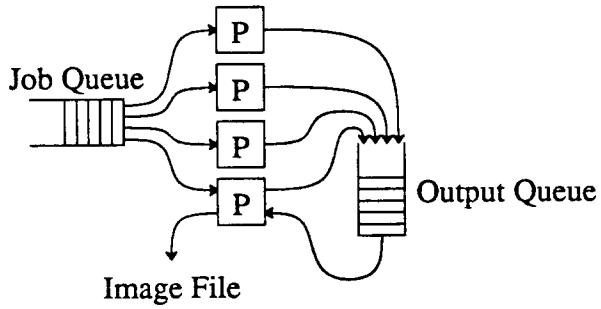

Figure 2.3. Simple Processing Model

2.1.1. Program Model

Figure 2.3 shows the organization of a four processor system where the processors take jobs from a global job queue and put results into a global output queue. The last processor is required to do all of the transfers from the output queue to the image file. The output queue is given higher priority so that it is kept as small as possible. It will typically receive a job from each processor before being emptied by the output processor.

The most natural units of work to schedule are either individual pixels or scanlines. If scheduling cost (synchronized access to a global work queue) is expensive, pixels will be too fine a work unit. On the other hand, a large unit such as a scanline won't incur much overhead from scheduling because there are so few scanlines, but it will incur a *termination loss* which is the idle time spent by processors that finish their

last work unit before other processors finish.

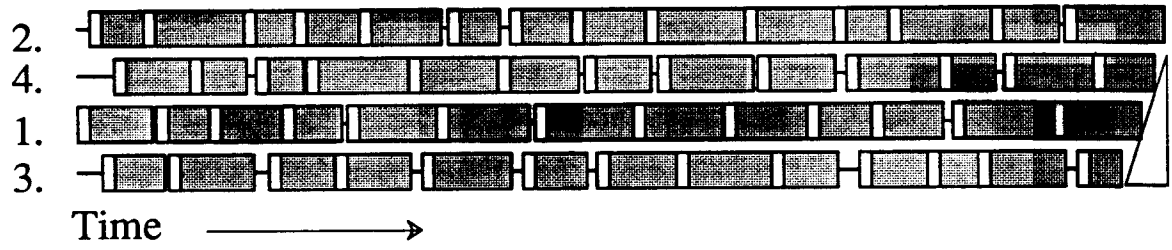

Figure 2.4

Figure 2.4 show a hypothetical dynamic scheduling system with four processors. Each of the horizontal strips show the activity of a processor: the shaded boxes represent time spent doing real work, the white boxes represent time spent scheduling a new piece of work, and the lines represent time wasted waiting for exclusive access to the scheduler. The termination loss (roughly indicated by the triangle) is the sum of the idle times spent by processors that finish early.

We can model this queuing system to find the size for the scheduling unit that gives an optimal run time. Let:

C be the sequential computation time

P the number or processors

N the number of jobs which the computation is divided into

S the time to schedule a job

T the average computation time per job ($T=\frac{C}{N}$)

The total work(W) of the system is (computation time + scheduling time + synchronization time + termination loss). When formulated in this way the execution time of the parallel program is simply W/P.

The computation time is C.

The scheduling time is $S*N$.

The synchronization time can be approximated by the waiting time in a M/M/1 queuing system. This can be seen by treating the critical section as the server and the processors as the customers in the queuing

model. The mean service time is S and thus the service rate $\mu = 1/S$. The mean interarrival time is $\frac{T+S}{P}$ and thus the arrival rate $\lambda = \frac{P}{T+s}$. Strictly speaking the model does not apply because arrivals and departures are not Markovian, but for a lightly loaded system the results from this simple model are reasonable.

The average waiting time in a M/M/1 queue is $\frac{1}{\mu-\lambda} - \frac{1}{\lambda}$ [Wol89]. This gives an average wait in the queue of $\frac{S^2*P}{S+T-S*P}$. The total over all N jobs is $\frac{N*S^2*P}{S+T-S*P}$.

The termination loss is $T*(P-1)/2$. This comes from noting that if the processes are sorted in order of termination, as they were in figure 2.4, the expected termination loss will be the area of a triangle with width equal to T, and height equal to $(P-1)$.

The execution time is thus:

$$\frac{C+S*N+\frac{N*S^2*P}{S+T-S*P}+T*(P-1)/2}{P}$$

For the Sequent on a test fractal of size 512x512 pixels: C = 1000 secs, P = 12, S = 45μsec. Figure 2.5 shows the predicted effect on execution time as a function of the number of jobs. The optimal point is when the task is broken into 10,000 jobs of 25 pixels each. The optimal size is so small because scheduling is very quick and the overhead is primarily a function of the termination loss which is minimized with small jobs.

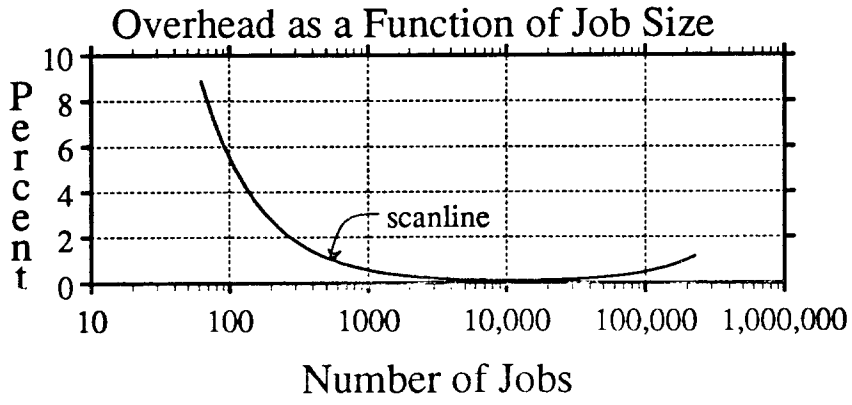

Figure 2.5

I chose to use scanlines which are much larger than optimal as the job unit for several reasons: First, scanlines are a natural unit and their manipulations are slightly easier than other units. Second, the overhead at this size, although not minimized, only amounts to one percent of the run-time. And finally, my measurement of the scheduling time is probably underestimated because some of the code outside the critical section should be counted as scheduling time and not computing time.

If more processors are used, the job size must be decreased because the termination loss becomes more significant. This simple model indicates that 1,000 processors could be used with 95% efficiency. There are other limits which appear far before this. Bus bandwidth and I/O bandwidth typically limit the size of shared memory machine to less than fifty processors [Bel85].

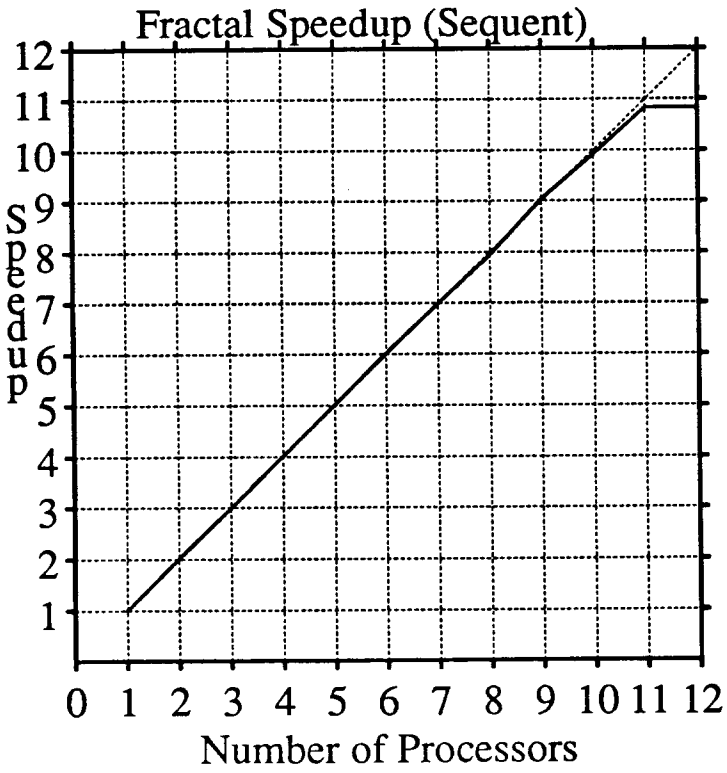

Figure 2.6

2.1.2. Experimental Results

Figure 2.6 shows the speedup curve for the fractal program running on the 12 processor sequent. The dotted line at 45 degrees show the ideal speedup. The solid line shows the speedup actually achieved.

The sequent is a multiuser machine, so part of it's resources are consumed by the operating system to service other processes. The last processor doesn't contribute to performance since it is reserved for other users.

In summary: simple dynamic scheduling is sufficient to obtain linear speedup on shared memory machines. Fractals are a computationly bound task with abundant high-level parallelism that is easily exploited.

2.2. Ray-Tracing

The fractal program model and analysis were introduced because they are very similar to the model of the ray-tracing program.

2.2.1. Basic Program Model

The sequential ray-tracing program works in two phases. First it reads in the scene, does some preprocessing transformations, and creates the spatial subdivision by linking polygons to all the cells they touch. Second, the image is rendered by tracing rays through the subdivision. The rendering phase typically takes 1000 times longer to execute than the startup phase.

Because the startup phase is such a small portion of the total run time, it does not make sense to parallelize it. My parallel ray-tracer does the startup phase with a single processor and creates the data structures in shared memory. After the startup processing is completed, a worker process is forked for each processor. These workers dynamically schedule regions of the image to render. The completed pieces of the image are left in shared memory for a single processor to correctly order the pieces of the image and to write the image file.

Dynamic scheduling is used, just as with fractals, so that the output can be done concurrently and so that the processor with the burden of doing the output can receive a reduced ray-tracing load. As mentioned earlier, dynamic scheduling allows coherency to be utilized by scheduling nearby pixels onto the same processor. Some ray-tracers [JoB86] use simple coherence mechanisms such as saving intersection points on patches so that the next (hopefully nearby) ray can be intersected faster. Others [HaG86] remember past shadowing polygons and test them first for nearby shadow rays. Speer [SDB85] attempted

to find pathways through empty space through which successive rays could be quickly traced, but their research found the coherence insufficient for this task. A final potential benefit of coherence is that data reference locality might help the memory caching system perform better. Pixels are actually rather large computational tasks (from 150,000 to 5,000,000 cycles), and therefore the cache effects (particularly with the 8K cache on the Sequent) will be small. In summary, coherence is not very important for this ray-tracer, but it may be important for others. The dynamic scheduling of groups of neighboring pixels allows this coherence to be utilized.

The scheduled group of pixels need not form a scanline. Square blocks of pixels might logically have better locality. An expirement mentioned in the next chapter shows that the shape doesn't matter. Since scanlines are easiest to manipulate, they will be used.

2.2.2. Model for Simple Antialiasing

So far I have assumed that pixels are independent. This in fact is rarely the case. At one sample per pixel, images show severe aliasing problems (jagged edges). Rather than take one sample of the scene per pixel, ray-tracers typically take several samples per pixel and average them to 'antialias' the image. Figure 2.7 shows a common antialiasing trick. Each square region encloses a portion of the image that a pixel should represent. By sampling at the corners of the region, four samples (shown by dots) can be averaged to form each pixel (shown by a circle). Each sample can be used by four different pixels so that the average number of samples per pixel is still only one. This technique improves the image quality without increasing the number of rays to be traced.

Unfortunately, antialiasing introduces dependencies between neighboring pixels. The two pixels (circles) shown in the figure share two samples along their common edge. Either pixel can be computed first, but if they are both computed in parallel, the samples will be calculated twice. In general, duplicate calculations will occur along the edges of the allocated regions. For scanlines this would double the total amount of work to be done. Square regions would be better, but even for reasonably large regions (20 by 20 pixels) the extra work is significant (10%). An alternative approach would be to share the samples at the edges. This requires synchronization and communication between processes, and could lead to processes occasionally waiting for their neighbors to compute the shared samples.

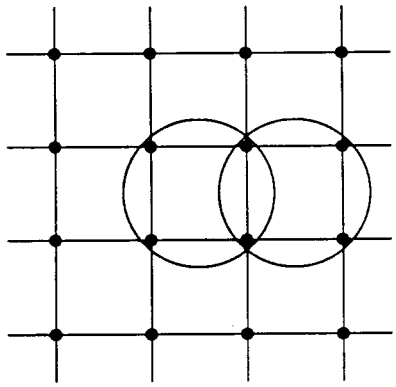

Figure 2.7. Sample Sharing for Simple Antialiasing

A much easier method exists when one realizes that samples are the independent entity, not pixels. Lines of samples can be scheduled just as lines of pixels were scheduled before. The output process can do all the averaging of neighboring samples to form pixels.

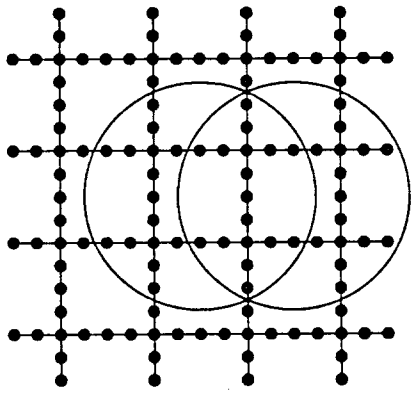

Figure 2.8. Sample Sharing for Complex Antialiasing

2.2.3. Model for Complex Antialiasing

More complex antialiasing techniques are in fact generally used since the previous technique still shows many aliasing problems. If required by variations in the picture, additional shared samples can be taken along the edges between pixels as shown in figure 2.8. Each pixel (circle) may be computed by combining as many as 32 samples. Instead of simple averaging, the samples are filtered so that samples closer to the center of the pixel are weighted more strongly.

All of the 32 samples are not always computed, but rather are adaptively computed as needed. The

samples at the grid points are computed first. If neighbors are sufficiently close to the same color, the intermediate samples (super samples) are simply interpolated. Otherwise rays are traced for the super samples.

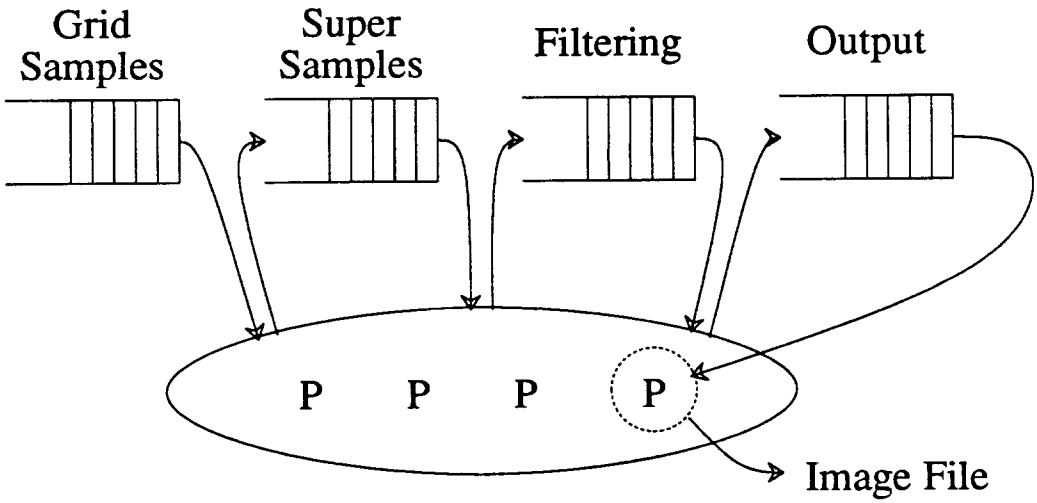

Figure 2.9. Multiple Queue Processing Model

New dependencies are introduced by this antialiasing technique since the computation of the super samples is dependent upon the results of the grid samples. The best way to deal with these dependencies is to break the original job queue into several queues as shown in figure 2.9. Jobs from the first queue are lines of grid samples. Once neighboring lines of grid samples have been computed, jobs from the second queue compute the super samples. After neighboring supersamples have been computed, the filtering can be done. Finally, a single processor can do all of the output as before. This reordering of the computation allows all jobs to be computed independently. The dependencies are managed entirely by the movement through the four queue system.

If locality were being exploited, one might think that doing the supersamples independently from the grid samples would sacrifice some locality. The supersamples, however, are samples taken near edges where locality is poor anyways. The real cost of the multiple queues is the cost of transferring the intermediate results through global memory buffers.

The allocation of jobs from the queues needs to be managed correctly. If all jobs are allocated from one queue before the next queue is started, excessive buffer space for storing the intermediate results will be used, and the output processor will be swamped as the filtering queue is emptied. On the other hand, if

the later queues are given highest priority, the final job from the first queue will move sequentially through each stage while the other processors wait. This sequential ordering (required because of the dependencies) will increase the termination loss beyond what it could be. My compromise approach it to leave a small number of jobs in the later queues until the earlier queues are exhausted. If the number of buffered jobs is chosen appropriately, at the point where the final job is taken from the first queue, there will be enough jobs in the second queue to keep the other processors busy until the final job from the first queue is completed. Likewise at the completion of the second queue there should be enough buffered jobs in the third queue.

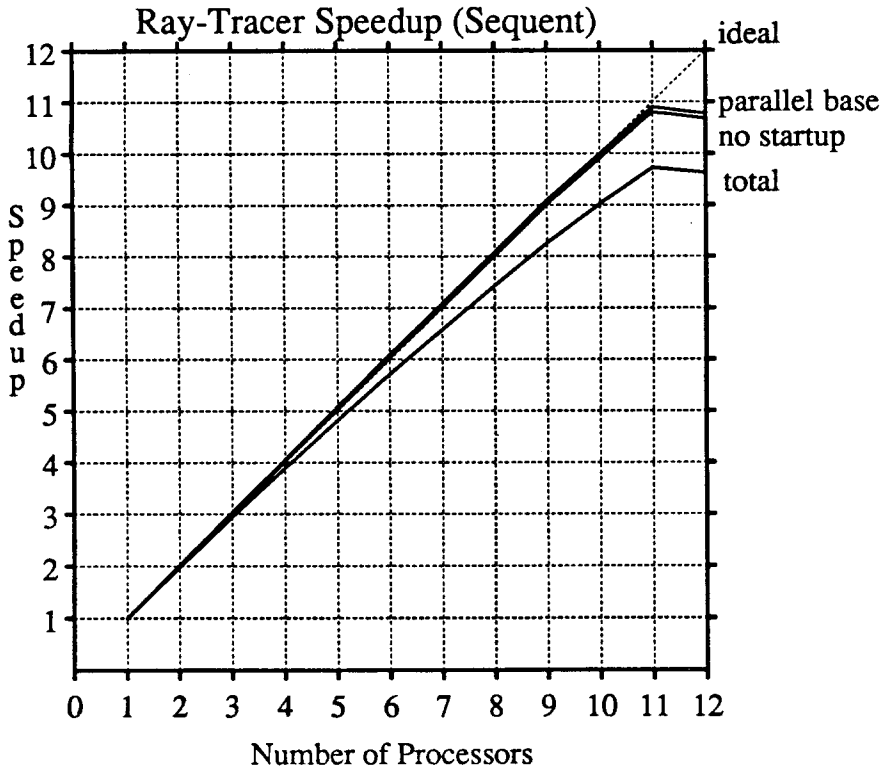

Figure 2.10

Figure 2.10 shows the speedup obtained on the Sequent. The curve labeled 'total' computes speedup by dividing the execution time of the original sequential ray-tracer by the execution time of the parallel ray-tracer. This is a pessimistic view of the speedup that would be obtained from real use since my test scenes were only 512 by 512 pixels where as 1024 by 1024 or larger is typically used. Since the startup cost of placing the scene into the space partitions is constant with respect to the image size, its effect is

overemphasized by a factor of at least four.

The curve labeled 'no startup' subtracts the non-parallelized startup times from the previous totals. Although this is an optimistic view, startup costs should not be considered to be the bane of large speedups that Amdahl's Law [Amd67] makes them appear to be. Gustafson [Gus88] points out that as faster (parallel) machines become available, startup costs often remain constant while job size increases. Startup costs lose significance.

The curve labeled 'parallel base' computes speedup in relation to the parallel version of the program using only one processor. The difference between this curve and the previous curve shows the costs of reorganizing the code for parallelization. These costs are overhead for scheduling and buffering.

For all of these curves, using a twelfth processor does not help because the operating system reserves the last processor for use by others. Performance actually goes down because of the context switching needed to run 12 processes on 11 processors.

The shared memory model works very well because the entire scene data structure can be placed in shared memory and is accessed primarily for reading. One exception is that each face has a tag that is written whenever a processor looks at the face. These tags must be separated out so that local copies can be made for each processor. A second exception is the global statistics that are kept about various operations such as: number of rays, number of intersections, and maximum ray depth. All of the statistics are reduction variables which can be computed locally for each processor and combined at program termination.

The critical sections of the code involve removing ready jobs from queues and linking completed jobs into queues. The decision of which job to allocate next is also done inside a critical section. To provide mutually exclusive access to the queues, each queue could have been protected with a separate lock. Instead, a single mutual exclusion lock was used for everything. The single lock is adequate since very little time is spent in the critical sections.

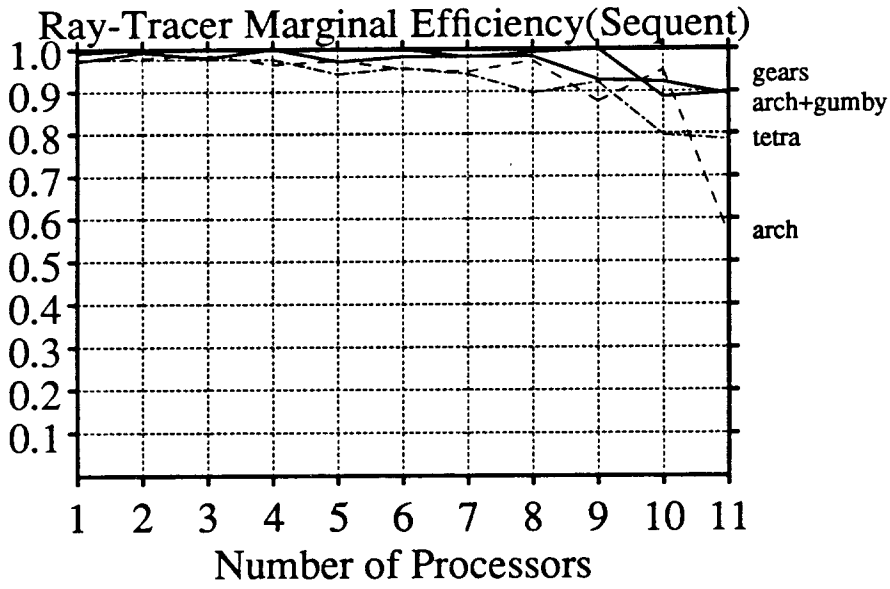

Figure 2.11

2.2.4. Analysis

The question typically asked about a multiprocessor application is how well does it scale. While one is interested in total speedup, a better way to look at the result is the marginal efficiency of each processor. Marginal efficiency is the increase in computational power resulting from each additional processor. The integral of the marginal efficiency curve gives the speedup curve. Figure 2.11 shows the marginal efficiency curves for the test images shown in the appendix. The graph shows only the 'no startup' results where start up costs are ignored and comparison is made relative to the original sequential program.

The performance on all of the scenes is very good. For the more complex scenes (gears and arch+gumby) the marginal processor efficiency is greater than 90% through the eleventh processor. The results for the simpler scenes are somewhat noisy but similar. Marginal efficiency starts to decrease after about the eighth or ninth processor. This may be an indication that bus congestion is starting.

There are four factors that may limit scalability: bus traffic, lock contention, scheduling unit size, and output bandwidth.

Bus traffic can be reduced by using larger caches. (The Sequent uses only 8K byte caches.) A later section will show that the large read only database has sufficient locality to benefit from large caches.

	Job Time (ms)	Scheduling Time (ms)
Grid Samples	5220	0.21
Super Samples	1120	0.21
Filtering	211	0.26
Output	70	0.18

Table 2.1

Table 2.1 shows the average job times and scheduling times for the arch scene. Scheduling times are insignificant compared to computation times. Hundreds of processors can be supported before lock contention becomes a problem.

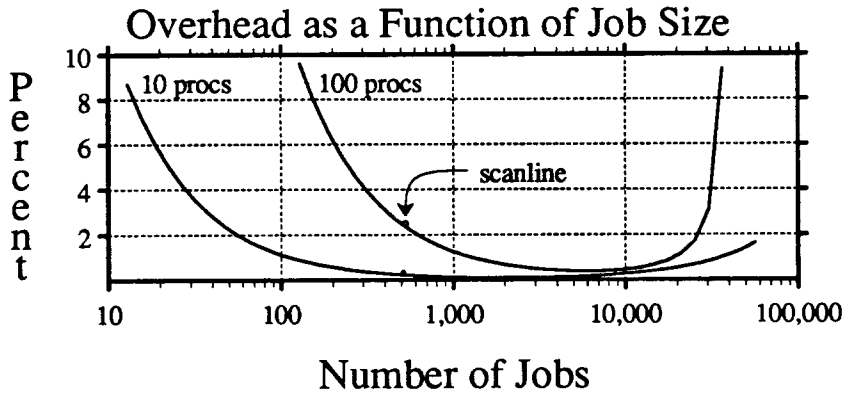

Figure 2.12

If more processors are used, the scheduling unit size will have to be reduced to a fraction of a scanline. Figure 2.12 shows the job size vs overhead trade off curves for 10 and 100 processors. This uses the queuing system model from the previous section and averages job times and scheduling times. The overhead curve for 10 processors is flat over a large enough range that using scanlines (512 jobs) incurs little overhead. With 100 processors, a job size closer to the optimal value of 50 pixels should be used. The four stage queue is not really as simple as a single stage queue, but the performance model should still accurately estimate the lock contention. The termination loss will in fact be less than predicted since it is based on the draining of the last queue, which fortunately has the smallest jobs. Small jobs will also help reduce the buffer space needed for the queues. Several megabytes are currently used for intermediate result buffers because the job's size is so large.

Output processing, which is done on a single processor, is the severest scalability constraint. It accounts for about 1% of total processing and will therefore limit the number of usable processors to 100. This limit could be extended by doing the run-length encoding portion of output processing as part of the filtering jobs instead. If a single output file were not required, I/O could be done by several processors in parallel.

2.2.5. Conclusion

This section has shown that for shared memory multiprocessors, ray-tracing can be parallelized at the level of image samples. Multiple job queues were used to handle dependencies caused by antialiasing. The optimal job size is a fraction of a scanline, although this becomes more important for large numbers of processors. On the 12 processor Sequent, efficiency was nearly 100% and speedup was linear. If larger machines can handle the bus traffic, this parallelization method should easily support 100 processors.

3. Algorithms for Distributed Memory

Machines with distributed memory, such as the N-cube, are not constrained by the bandwidth of a single shared bus, and can therefore be built with a much larger number of processors. N-cubes have been built with 1024 processors. Linear speedup on machines of this size has been obtained for some applications [Gus88]. Using distributed memory machines for ray-tracing, however, is complicated by the large irregularly accessed database.

When the scene database is small enough to fit entirely within a processor's memory, the best algorithm is to simply replicate the database, and let each processor render a different section of the image. The only communications will be for the shipping of results to the host, dynamic scheduling of image regions, and sample sharing required by the antialiasing mechanism. The algorithm becomes essentially the same as the shared memory machine.

For complex scenes where the database is too large to fit within a single node this simple algorithm breaks down. The large database problem can be addressed in two general ways. Data can be moved around the machine and cached where it is needed, or oppositely, computations can be migrated to where the data is stored. These two approaches to handling the large database problem for ray-tracing are analyzed below.

The analysis in this section could be used for any message passing computer, but the specific values for parameters, such as message passing time, apply to the Intel iPSC Hypercube. The message passing overhead is frequently found to limit performance of applications for the hypercube. Message passing time consists of .6ms overhead plus 3ms per kilo-byte of data sent [DoM87]. This high cost of message passing is the focal point for this study of parallelization methods. To efficiently use the hypercube, one must make sure that message passing is not the dominant task. A minimum acceptable efficiency is 50%. At this point half of a processor's time is spent computing and half is spent passing messages. If message passing is done concurrently, this is the case where both the processors and the message passing hardware are fully utilized.

3.1. Distributed Database

In the work of [Ull83], [CWB86], [PrB88] and [DiS84] there are two methods to deal with the distributed database: send rays, and send data. [Ull83], [CWB86] and [PrB88] consider only sending rays, while [DiS84] relies primarily upon sending rays but also sends data for load balancing. A third approach that we didn't find in the literature is to only send data and never send rays.

3.1.1. The Data Caching Approach

In this new approach, each processor would dynamically schedule a group of rays and trace them to completion. The processor would manage its local memory as a cache of the scene database. When a ray needs to access data not in the cache, the processor sends out a request for the data. Eventually when the data is returned, the ray tracing can be continued. Since all processors perform equivalent tasks and dynamically schedule work, load balancing is automatically achieved.

This approach requires a high degree of locality among the data references. Messages to move parts of the database around will be much larger than messages for rays. If this algorithm is to perform better, it must therefore send fewer messages. An analysis of the reference locality will be given in a later section.

Essentially, this approach emulates a shared memory machine. The differences are that the memory system is distributed and the caches are managed manually. Cache misses are now very expensive since messages have to be sent, but if the hit rate is large enough, long miss service time will be acceptable. The shared memory emulation approach is very appealing because it is simpler than ray-passing algorithms and automatically solves the load-balancing problem.

3.1.2. The Ray Passing Approach

At a high enough level, the algorithms suggested by [Ull83], [CWB86], [PrB88] and [DiS84] are all roughly the same. Each processor manages a region of the database, and when rays leave a processor's region, they are sent to the processor in charge of the neighboring region. The differences lie in the allocation of data onto the processors and the allocation of image regions. [Ull83] and [CWB86] statically allocate regions according to arbitrary rules. [PrB88] uses sub-sampled trace results to create a more load-balanced static schedule. [DiS84] performs dynamic adjustments to the regions to balance the load. All

four algorithms assume that the processors contain mutually exclusive regions of the database. This means that for simpler scenes most of the memory will be unused, which in turn implies that more message traffic will occur. [CWB86] suggests that to fully utilize memory, a scene should be packed into as small a group of processors as possible and that this group should be replicated across all available processors.

3.2. Poor Locality

On an intuitive level, good locality (coherence) would seem likely since neighboring rays generally hit the same objects and are colored similarly. One study [SDB85] found that coherence was insufficient to be utilized by software. On second evaluation, reflection angles, particularly off of curved surfaces, are magnified so that after a few reflections neighboring rays can diverge significantly (figure 3.1). A later section of this paper quantifies the amount of locality actually observed.

I hypothesized that sampling in scanline order may waste much of the locality available and that other scanning orders that cluster neighboring pixels would work better. For example, small rectangular regions would provide better locality but still be easy to implement. An awkward but very local ordering is that traced by a Hilbert curve. It turns out that there is no significant improvement from using Hilbert ordering over scanline ordering.

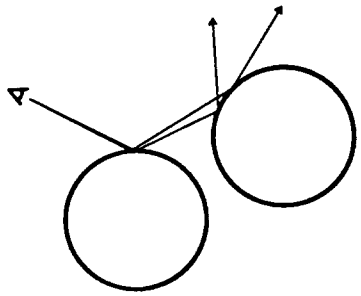

Figure 3.1. Diverging Rays

Other evidence of poor locality comes from noting that the ray tracer will cause virtual memory thrashing unless there is enough physical memory for the entire data structure. The primary cause of this may be that data is simply not organized well in memory. A 100 by 100 by 100 spatial subdivision array occupies four mega-bytes. As rays move through this array they will access locations at strides of one, one

hundred, or ten thousand words. The large strides will hit new pages on each access, and thereby touch large portions of memory.

If the four mega-byte array is distributed over the 32 processors of the Hypercube, it will occupy 128K on each processor. This is more than half of the available memory. For parallel ray-tracing, a new data structure with better locality and smaller memory usage is needed.

3.3. Reorganized Data Structure

As seen in the previous section, the uniform cellular subdivision is an inappropriate data structure for a distributed ray tracer. A better data structure must make more efficient use of memory.

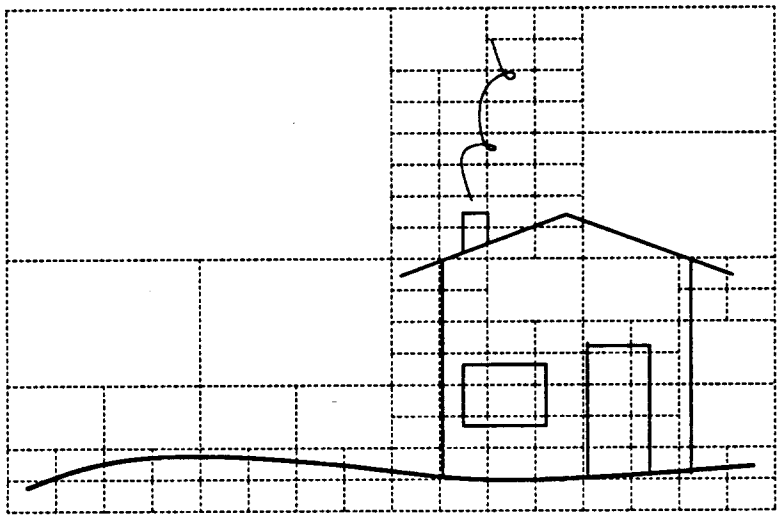

Figure 3.2. Oct-tree Subdivision

A common data structure used in ray-tracers is the oct-tree. Figure 3.2 shows a two-dimensional analogue (quad-tree). For this example, the image is recursively divided into quadrants until each quadrant contains a small number of objects. For an oct-tree the object space is recursively divided into octants. This data structure uses large cells for large empty spaces so that little storage is wasted.

Oct-trees have the disadvantage that moving to neighboring cells is more complex and therefore more expensive than for uniform spatial subdivision. Uniform spatial subdivision has better performance for scenes where the object distribution is not highly clustered. A second disadvantage is that the distribution of an oct-tree across a parallel machine appears to be awkward. Finally, replacing the data structure

would have required significant modification to the existing ray-tracer.

A compromise data structure is a uniform two-level subdivision. Figure 3.3 shows a two-dimensional analogy of this. Object space is divided first into relatively large cells, and if these contain too many objects, they are subdivided further. The size of the small cells is kept the same as it would have been for uniform spatial subdivision. This allows the same cell stepping method to be used. The advantage of the two-level structure is that large cells can be used for empty space while small cells can be used only where they are needed.

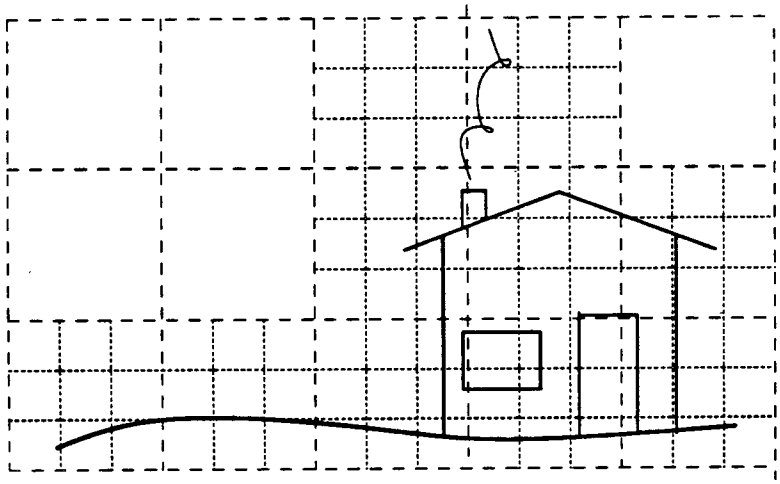

Figure 3.3. Two-Level Subdivision

Figure 3.4 shows a graph of the memory usage for different implementation choices. The vertical axis shows the total amount of memory used. The horizontal line at 1100K bytes is the memory required for the objects themselves. The excess above this is the cost of the subdivision structure. Along the horizontal axis is the size of the larger cells. The value 8 at the right end means that large cells contain 8 by 8 by 8 small cells. The value 1 at the left is for the original one level uniform subdivision. The various lines are for choices of the number of objects allowed in a large cell before it is broken up into small cells. The lines are for the values 0 through 4, and since they all appear similar, they have not been specifically labeled. This particular scene is the Gearbox (see Appendix). Other scenes have similar savings.

Cells of size 4 by 4 by 4 are best for optimal memory usage. The number of objects allowed in large cells should be 0 since this doesn't hurt the memory usage and can't cause any extra intersection tests to be

performed. The performance cost of these choices is between 4 and 8 percent, which results from the increased complexity of moving between cells.

Not only is memory usage greatly reduced, but reference locality is also improved. The cells of a 4 by 4 by 4 cluster can now be stored together rather than being spread over the global array.

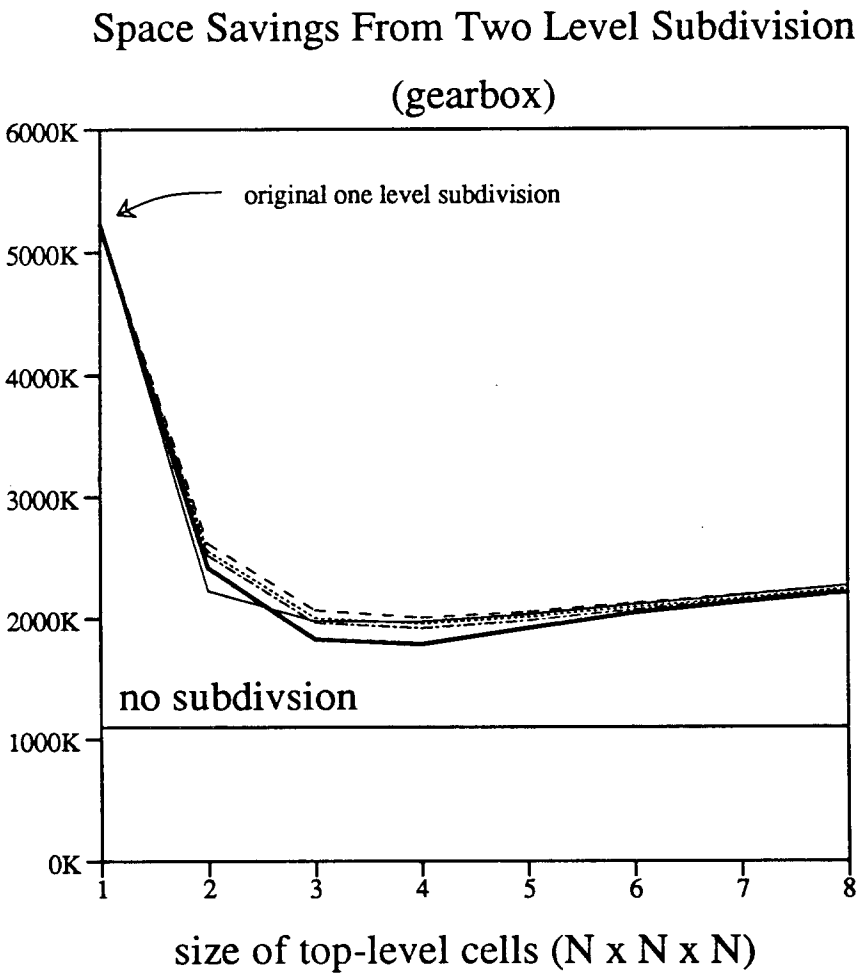

Figure 3.4

This structure is very appealing for distribution on a parallel machine. The top-level of cells can serve as a global map of the clusters of small cells. Data caching of top-level cells is easily supported. Empty cells don't require any extra storage, just a NULL pointer, so every processor can know where the empty space is. This allows all processors to navigate through empty space without sending any messages.

3.4. Locality Measurement

The locality of the data reference stream can be measured by the hit rate achieved in a cache. Figure 3.5 shows the hit rate curves for four test scenes (see the Appendix). The graph shows that the gearbox achieves a 99% hit rate when the cache size is 400K bytes. The other three scenes achieve 99% hit rates with caches of less than 70K bytes.

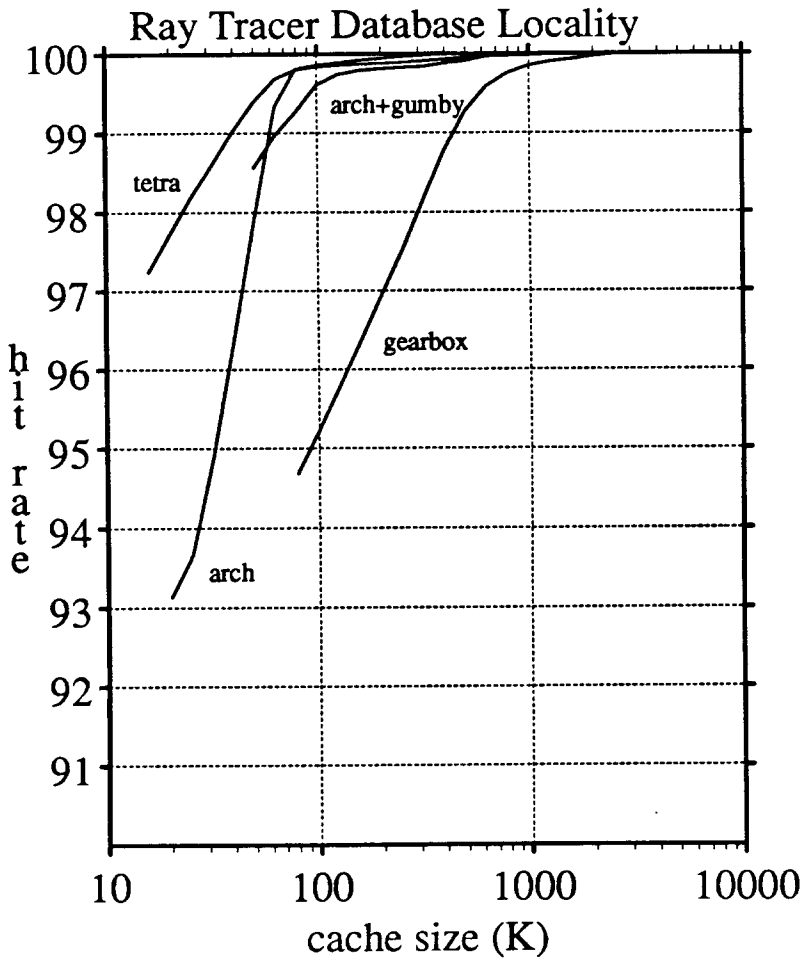

Figure 3.5

3.4.1. Cache Simulation

In a parallel implementation, single top-level cells (of which there are typically thousands) will be the units transferred between processors. The cache simulation therefore treats all of the structures (faces, vertices and colors) that are referenced from a top-level cell as a single cacheable unit. If two top-level

cells each reference the same face, then that face is replicated in both. The cache simulation records the hit rate that would be achieved if the cache could manage these variable sized units fully associatively with an LRU replacement policy. Each time that a ray enters a new small cell, an access is registered to the top-level cell to which it belongs. Accesses to empty top-level cells are ignored since empty cells are known by all processors and don't require any data accesses.

The cache simulation gives optimistic results. It ignores the memory management problems of a variable sized cache, and it ignores the costs of managing this cache. The analysis has the benefit that it is relatively easy and fast. The performance curve for a fully assosciative LRU cache over all cache sizes can be calculated in a single pass.

Scene	Number Top Cells	Percent Touched	Percent Occupied	Average Size	Largest Size	Total Size	Real Size	Average Run Length
Tetra	3584	16%	16%	5.4K	15K	3.1M	1.6M	24
Arch	7220	6%	18%	1.4K	20K	0.6M	.7M	10
Arch+Gumby	22344	10%	13%	1.0K	46K	2.2M	1.4M	13
Gearbox	13079	7%	8%	4.0K	76K	4.1M	2.1M	11

Table 3.1. Cache Simulation Statistics

3.4.2. Scene Statistics

Table 3.1 shows some other statistics from the cache simulations. **Number of Top Cells** refers to the total number of cells in the top-level of the spatial subdivision structure. **Precent Touched** is the percentage of these cells that were ever entered by a ray. These numbers are small primarily because accesses to empty cells were not recorded but also because some parts of the image are hidden and rays never reach them. **Percent Occupied** is the percentage of non-empty cells. This can be compared to the previous column to see that for the Arch scene only one third of the occupied cells are ever accessed. The other scenes are more completely accessed. **Average Size, Largest Size,** and Total Size refer to the data blocks used to hold the data of a super cell.

With realistic scenes, data is distributed very non-uniformly. Some cells will have only one face while others may have a hundred polygons to create fine details. The large size of some cells (76K for the gearbox) will be very expensive to send around the computer. This indicates a weakness of the data

structure. Oct-trees would be able to keep the cell size distribution more uniform.

Real Size is the amount of memory needed for a uni-processor. Generally the parallel algorithm would use more space because objects referenced by multiple cells must be duplicated.

Average Run Length is the average number of times which a cell is repeatedly accessed. Repeated accesses occur for several reasons: First, multiple second level cell accesses within a top-level cell are each recorded as an access. A typical ray passing through a top cell will hit from 4 to 10 small cells. Second, reflection and shadow rays will originate from the endpoint of the previous ray and thereby continue accessing from the same cell. Finally, navigation through empty space is not recorded so that it is possible for several successive rays to touch only one unique occupied cell.

Scene	Computation Time per Access	Message Overhead	Hit Rate with 200K cache	Efficiency with 200K cache	Percent of Total Scene on a Processor
Tetra	144μs	69600μs	99.88	63%	13%
Arch	240μs	21600μs	99.94	95%	29%
Arch+Gumby	236μs	16800μs	99.81	88%	14%
Gearbox	680μs	52800μs	96.95	29%	10%

Table 3.2. Data Caching Performance

3.4.3. Performance of Data Caching Algorithm

These cache simulation results along with the times required for computation and message passing can be used to estimate the performance of the parallel program. Table 3.2 shows the cache hit rates that would be achieved when 200K bytes are used for the data cache. The computation time is the average run time between data structure accesses. The message overhead is the transmission time of an average message from table 3.1. These hit rates, computation time, and message overhead are used to find the efficiency of the program. The Gearbox scene will run at only 29% efficiency. If the cache size were doubled, it would reach 50% efficiency. The rest of the scenes are adequately efficient.

These results are optimistic because they are based on the assumption that data will be easy to locate and on average will be serviced by a processor that is two nodes away. Unfortunately the trend in the data is that more complicated scenes, which are our primary interest, have worse efficiency than the simpler

scenes. On the other hand, as message passing times become shorter and memory sizes become larger (which is the trend), this algorithm will become more viable.

Scene	Number of Messages per Ray	Computation Time per Ray	Message Overhead per Ray	Efficiency
Tetra	0.5	6.4ms	1.6ms	80%
Arch	0.9	8.4ms	3.3ms	72%
Arch+Gumby	0.4	11.2ms	1.6ms	88%
Gearbox	0.3	12.4ms	1.0ms	93%

Table 3.3. Ray Passing Performance

3.4.4. Performance of Ray Passing Algorithm

Simple execution statistics can also be used to analyze the ray passing algorithm. Table 3.3 shows estimates of the number of messages that will be sent per ray. This is based on analyzing the distance a ray travels before it hits an object, and the likely size of a data cluster. Most rays will not leave their starting cluster and therefore won't require any messages. For all of the scenes, less than one message per ray is needed on average. The efficiency has been estimated based on the overhead for sending messages and the average computation time per ray. Unlike the results for the data caching algorithm, the more complicated scenes do better here.

The results are again optimistic since there were many simplifying assumptions that went into these estimates. Rays were assumed to go to adjacent nodes since physically neighboring processors are expected to contain neighboring regions of the object space. Memory size was assumed to be sufficient, and load balancing was assumed to be completely effective, cost free, and capable of maintaining reasonably round clusters of cells assigned to each processor. These are serious issues and need to be studied further.

On the optimistic side, duplication of heavily referenced parts of the database may be able to reduce the number of messages needed. The optimization that allows all processors to know where empty space is will sometimes allow a processor to send rays directly to a distant processor. Bundling messages that are sent to the same node could be used to overcome the overhead associated with sending short messages. A

good load balancing mechanism might even be able to adjust the object distribution away from round clusters to oblong clusters that follow the major channels of ray flux through the scene.

3.5. Reorganizing Access Order

The data caching algorithm might be improved by allowing processors to trace more than one ray at a time. When a ray misses the cached data, it's execution can be deferred until later. The hope is that a large collection of rays can be traced until they all leave the current data collection, and then new data can be loaded and most of the rays can be traced through it as well. In contrast, the non-deferring system would have shuffled data in and out of the cache repeatedly for subsequent rays and thus required a much higher messages traffic. This modification is equivalent to allowing a traditional data cache to look ahead in its address stream and service requests out of order. Clearly this is an advantage, but the magnitude of the advantage needs to be studied. With large caches and high hit rates, the advantage should be small.

A combination of the data caching and ray passing algorithms will probably work best. Ray passing has an advantage because rays can be sent as compact messages, but data passing is required for load balancing. Probably the only way to study the interactions of these two approaches is to actually implement them on a real machine.

There is one difficulty that I have neglected so far. Whenever an object is intersected against a ray, it is tagged so that the intersection against the same object will not be performed again in another cell. One can imagine a ray grazing along the surface of an object through many cells but never actually hitting the object. Without the tagging optimization, the intersection test would have to be performed for each cell. In practice this optimization halves the number of intersection tests performed.

The problem is that tags are global variables referenced by more than one cell. When a ray leaves a processor, the tags of the objects that were hit need to be moved also. It is impractical to send the entire tag table since it may have 10,000 or more entries.

Extra Face Tests		
	Arch	Gearbox
Remember originating face	62%	27%
16 entry hash table	5%	9%
64 entry hash table	2%	5%
Local tags	16%	

Table 3.4. Mechanisms for Reducing Extraneous Face Tests

Table 3.4 shows the performance of several mechanism for reducing the number of extraneous face tests. The simplest optimization is to remember the face from which a ray originates (for secondary rays) and avoid testing for an intersection with that face. A better method is to keep a small hash table of the most recently intersected faces. Experiments indicate that a 16 entry table will reduce the extra intersection tests to 9% or less, and that a 64 entry table cut this to 5%. This is not entirely satisfactory since a 64 entry table takes more space than the ray itself, thus more than doubling the message passing overhead. Another possibility was studied in the actual implementation. Complete tags were kept locally, but they were left behind when moving to a new processor. Extra tests are only required for objects that straddle processor domains. For the arch scene the resulting performance penalty was 16% extra tests.

3.6. Conclusions

This section has demonstrated that despite abundant parallelism, ray-tracing of complex scenes is not the ideal task for parallel computers that it is often believed to be. Performance estimates of data caching and ray passing algorithms indicate that ray passing is better for complex scenes. The analysis makes many assumptions that should be verified with a real implementation. Larger memory sizes and faster message passing will improve the efficiency of both algorithms.

4. Distributed Memory Implementation

This section discusses the implementation of the ray-tracer on a distributed memory hypercube architecture. Two good papers introducing first generation hypercubes are [Sei85] and [TPP85]. We have at Berkeley a 5 dimensional Intel iPSC hypercube with 32 nodes. Each node consists of an Intel 80286 processor, 80287 floating point co-processor, and 512K bytes of local memory. Only about 300K of this memory is available to the user because the operating system uses the rest [Hyp86]. Code is written in C. A library of message passing routines is used to communicate between nodes.

Berkeley also has available a 64 processor N-Cube, but its nodes only have 96K bytes available to the user. Similar work [Lia87] on parallel beam-tracing found the memory size of the N-Cube much too constrictive. The code size of the parallelizable kernel of the ray-tracer is more than 100K bytes.

The hypercube as shown in figure 4.1a is a beautifully symmetric interconnection topology. With N processors, there is a path of length at most $\log_2 N$ that connects any two processors. Other useful topologies such as linear arrays, 2-D grids, 3-D grids and binary trees can be embedded using a subset of the connections. However from the I/O perspective as shown in figure 4.1b, the system has a potential bottleneck since it must drain all output through a single link to the host. Fortunately ray-tracing is sufficiently compute bound that this I/O bottleneck is not a problem.

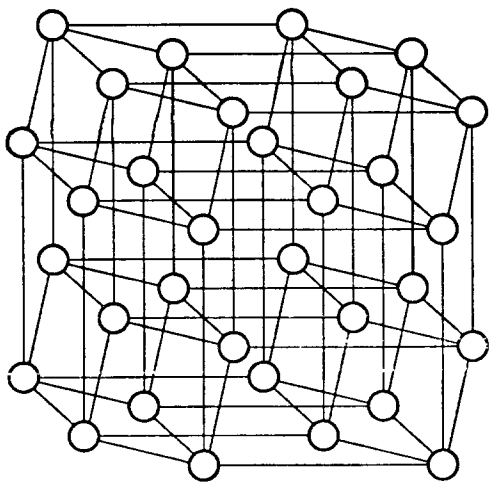

Figure 4.1a

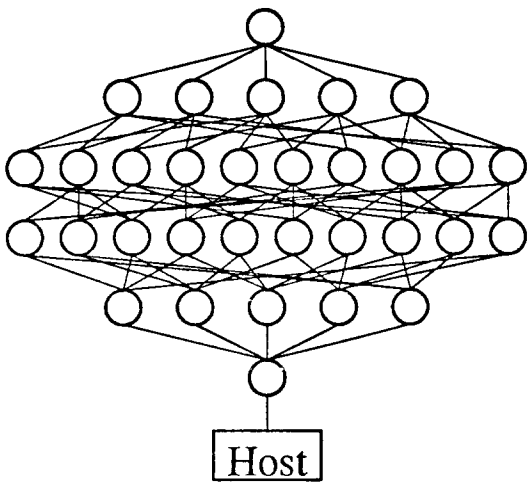

Figure 4.1b

The hypercube nodes have routing hardware that allows multi-hop messages to be passed through intermediate nodes without affecting their processors. This independence from message traffic lets all nodes provide an identical computational resource. Recall that on the Sequent ray-tracer, one processor was burdened with all of the I/O processing and therefore could not do as much computation as the other processors. The balanced performance of the nodes suggests that static scheduling can be effectively used. Priol & Bouatouch [PrB88] took this approach, but with only limited success. Probably their disappointing performance resulted from an inaccurate load metric that they used for scheduling. In retrospect, I should have chosen static scheduling, but instead chose to experiment with the more difficult problems that arise from a dynamic solution to load balancing.

4.1. Program Overview

From the simulation studies, I have shown that ray passing is the more promising algorithm to investigate. This section gives a brief overview of the structure of the Hypercube ray-tracer. The remaining sections explain the important design decisions.

Each processor is assigned responsibility for some region of space and stores the data associated with all objects that fall in its region. As a processor traces a light ray through its region, that light ray will either hit an object or pass through to a region managed by a different processor. When a ray passes through to another processor, it is sent as a message. When it hits an object, the resulting color of the ray is determined and sent back to the processor that originated the ray.

Actual operation is much more complex. A ray does not stop when it hits an object, but rather spawns a tree of rays that determine the reflection, refraction, and shadow contributions to the color of the ray. On the sequential ray-tracer, this tree is evaluated recursively. When a ray hits an object, the reflection, refraction and shadow rays are recursively traced and then combined by weighting their contributions according to the surface properties of the object. This recursive evaluation tree presents a problem for parallel execution because of the long latency when part of the tree is computed on another processor.

The latency could be dealt with by context switching, but instead, the recursive calculation can be reorganized to eliminate the need to return results back up the tree. Instead of computing each ray's contri-

bution after the ray's completion, the contribution factor is computed before the ray is started. When the color of a terminating ray is eventually determined, the color is scaled by the ray's contribution factor, and the result is added directly to the final pixel value. Now when a ray leaves a processor, the sender does not have to wait for the result since wherever the result is eventually determined, it can be sent directly to the processor collecting results. Many rays can be sent simultaneously and they each require only a small buffer in which to collect the pixel color as result messages are returned from the various branches of the ray tree. When all the result messages are eventually received, the pixel value is finished.

The advantages of this new method are that it is easier to manage the result buffers than the multiple execution threads of many ray trees and that much less space is needed for a single result buffer than for a saved process state. A disadvantage is that more result traffic may be required. Rather than recombining results along the ray tree, which would most likely involve messages to nearby nodes, results are sent all the way to the collection site, which might be many nodes away. This is not as large of a problem as it might seem since distant nodes in a 2-D or 3-D grid are often closer when the extra dimensions of the hypercube are used. Furthermore, at the cost of increased latency, much of the overhead due to returning result messages can be eliminated by packaging results destined for the same processor together. The primary overhead of small messages is dependent on the number of messages, and not their size.

A technique suggested by Ullner [Ull83] is used to determine when all contributions to a pixel have been received. The contribution fractions are returned along with the results of a ray. As the results are added to the pixel value, the contribution fractions are also summed. When the sum of the contribution fractions reaches 100%, the pixel has been completed. Integer arithmetic is needed to avoid roundoff, and integer precision turns out to be adequate since the ray tree is truncated at the point where the contribution of a ray is less than the precision of a pixel, and pixel colors have only 8 bit precision.

The difficulties that arise when implementing this algorithm are deadlock avoidance and load balancing. Deadlock occurs when too many rays are released into the system and some buffering capability is exceeded. This can be in the system's buffers for routing messages or in a process's internal buffers for storing rays received from neighbors. Load balancing is required because the computational density of the data base is very nonuniform. Some regions are hit by many rays, and some by few or none. The regions

managed by the processors must be adjusted if all of the processors are to be kept busy.

4.2. Ray Generation

The first decision to be made about ray generation is whether a single processor (possibly the host) will be responsible for initiating all of the rays, or if all of the processors will initiate a fraction of the rays. An advantage of a single processor generating all of the rays is that controlling the number of messages in the system, so as to avoid deadlock, is easier. Unfortunately, a single processor is unable to generate rays quickly enough to keep 32 processors busy.

In the static data distribution by Priol & Bouatouch the image plane was subdivided into regions with equal expected computational load. The data objects were then distributed among the processors in rectangular cones so that primary rays (rays from the eye point) would stay within the originating processor. This is a minor optimization since only a few percent of the total rays are primary rays. With dynamic load redistribution I can't predict where the data will end up, so I simply divide up the image evenly. Each processor generates rays for a rectangular block of the image. Data objects are initially distributed so that primary rays can stay on the same processor, but the reality of fitting the data onto the processors and later balancing the workloads will disturb the correspondence between image regions and data regions.

Regulating the ray generation rate to avoid deadlock is a very difficult problem. The operating system has 100 message buffers per processor for routing (which unfortunately use 1K of memory each regardless of message size.) Despite the large number of buffers, the operating system is quite susceptible to deadlock whenever the processes let messages accumulate without receiving them.

From the program's perspective the exact size of messages is known (258 bytes for a ray), and buffering can be done more efficiently inside the program than by the Operating System. The program allocates space for 50 rays (and possibly more on demand). The input queues must not be allowed to reach this limit, or deadlock can result when some processor tries to send a message to a congested processor and the congested processor tries to send a message back. In practice, once a processor's buffers fill up, the deadlock cycle is completed quickly.

One can try to avoid deadlock by many methods: the number of rays in the system can be limited, the current buffer status can be frequently propagated around the machine, panic messages can be sent when buffers are nearly full, or the rate at which new rays are generated can be regulated.

The most critical test for the ray generation strategy is immediately after start up. The data distribution is probably quite poor, and a great number of rays are likely to converge on a single unlucky processor. The processors must be subdued from their haste in generating rays.

The first strategy, where the number of rays allowed in the system is limited by some sort of token system, fails because the number of tokens is limited by the resources of a single processor to 50. However 50 tokens spread over 32 processors is not enough to allow even basic operation since a single ray will typically spawn several offspring.

The second strategy of frequently propagating the buffer status across the machine was rejected because of the large message overhead required to implement it. Perhaps this information could have been piggybacked onto other messages, but this wasn't tried.

The third strategy of sending out panic messages at a high water mark was tested and performed very poorly. The panic messages had to be sent out before the buffers were even half full because of the latency and backlog of messages in the system. Once panics started inhibiting execution, the whole machine would soon grind to a halt until the paniced processor caught up on its backlog of rays. The whole machine would stutter along with horrible performance. Occasionally misfortune would still conspire to deadlock the machine.

The fourth strategy was to have a global rate at which all processors could generate rays. For example each processor might be allowed to generate rays at the rate of 25 rays per second. The global performance was monitored, and if all of the processors were sitting idle for some fraction of time because they weren't allowed to generate enough rays, the global generation rate would be increased. If on the other hand a processor was overworked and couldn't keep up with its ray generation quota, the global generation rate would be decreased. The net effect of this management technique is that the most heavily loaded processor would be fully utilized and the other processors would be less fully utilized in proportion to their respective loads. Since all processors generate rays at the same rate, they will all complete at the same

time. The goal of load balancing becomes to match the loads on the processors so that they can all be fully utilized. Unfortunately, this technique also suffers from instability. Occasionally coincidental events will require a rapid decrease in the generation rate to avoid buffer overflow. Adding damping to the system slowed down its recovery but wasn't able to prevent instability problems.

The fifth strategy was to limit the number of rays that a processor could be waiting upon for results. The limit started out small but was slowly increased as results were successfully returned. When a panic message was received from a processor, the pending ray limit was frozen so that a result would have to be returned before the next ray was sent out. This, in a sense, provides acknowledgement for free. As each result is returned, it signifies that the buffer that had held it is now empty. Processors give preference to processing rays passed from neighbors rather than generating rays of their own. This keeps input buffers constantly drained, but whenever a processor finishes its passed rays, it can start generating rays of its own and thus never needs to sit idle except when the machine is congested. This technique worked well with 16 processors but occasionally deadlocked with 32 processors because buffer space was inadequate.

Figure 4.2 shows a performance graph for a run where the load is well balanced. This is for a 16 processor configuration. Each individual graph shows a processor's utilization as a function of time. Note that the processors are fully utilized until a point near the end of the run. At this point their utilization drops dramatically. Figure 4.3 shows the progress that each processor makes towards completing its assigned block of rays. The falloff in utilization corresponds to the point at which the processor can no longer generate rays to keep itself busy but must wait for rays passed from neighbors. The occasional small dips in performance correspond to congestion problems.

The scene used for figures 4.2 and 4.3 is an artificial scene designed to be well load balanced. Figures 4.4 and 4.5 are the performance graphs for the Arch scene. This is a very poorly load balanced scene. Note that some processors (lower left corner) finish all of their rays in 5% of the total runtime, and then sit nearly idle. Other processors (upper right corner) sit nearly idle but make only slow progress towards finishing their work. This suggests that the rays they generate are sent to another processor for computation while the sender waits. Only one processor (lower left middle) is fully utilized. This processor is greatly overburdened and spends all of it's time processing rays passed from neighbors. Not until near the

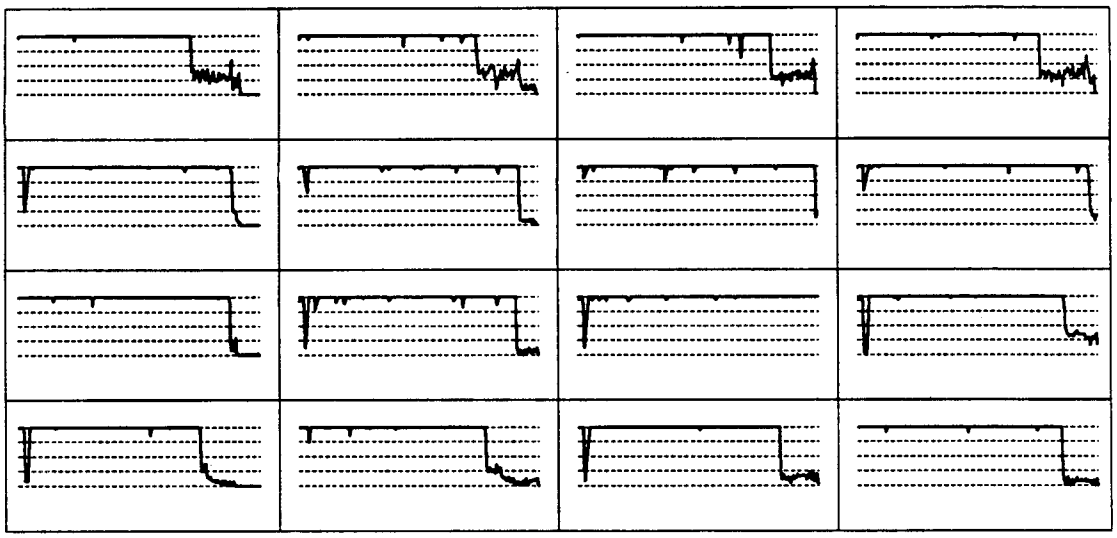

Figure 4.2. Processor Utilization (0 to 100%)

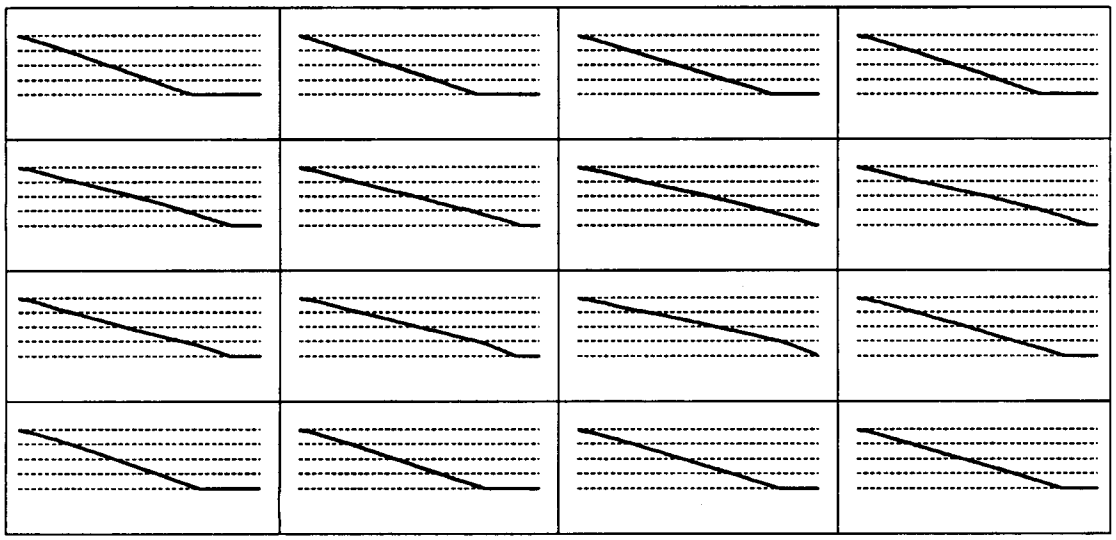

Figure 4.3. Remaining Work (0 to 100%)

end of the run when it has finished the neighbors' rays, is it able to make progress on its own rays.

4.3. Adjustment Granularity

The two level spatial subdivision structure was originally developed to save memory when storing the scene. The large cells, consisting of a 4 by 4 by 4 block of small cells, also form a convenient unit for load balancing. Each processor maintains a map of where the large cells are located and routes messages according to that map.

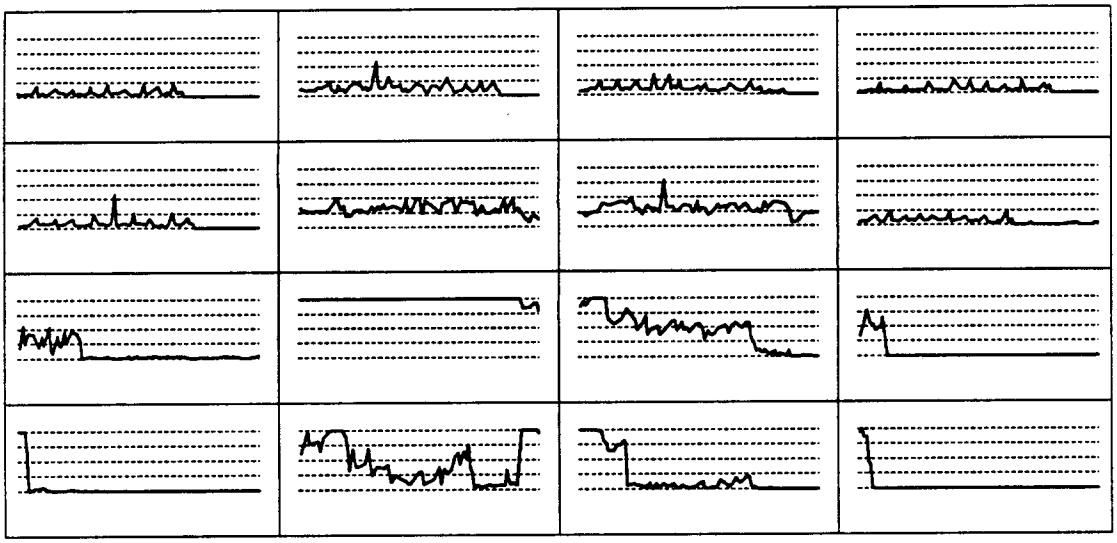

Figure 4.4. Processor Utilization (0 to 100%)

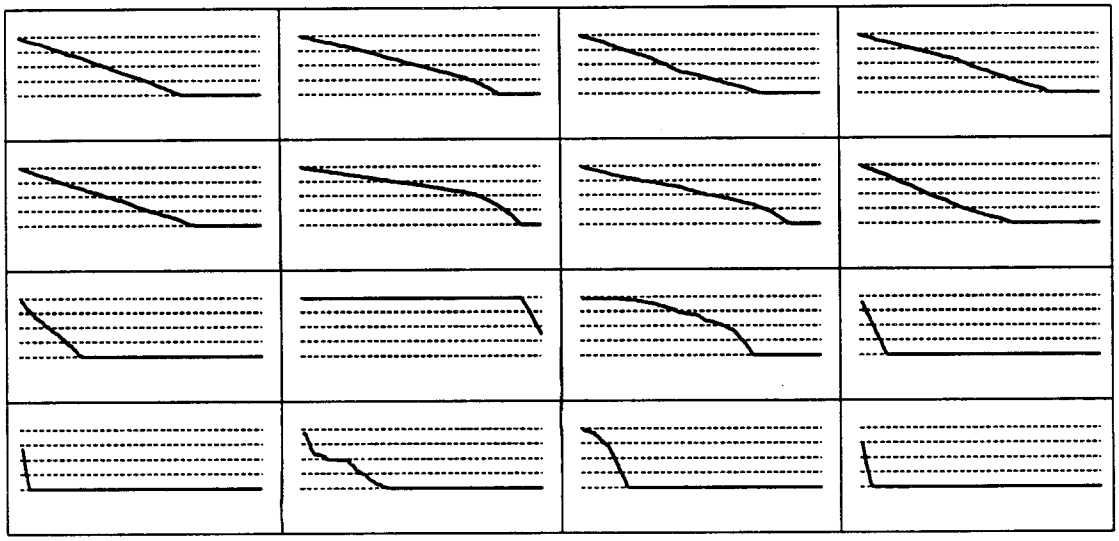

Figure 4.5. Remaining Work (0 to 100%)

The analysis in the section 3 was based on simple averages of statistics. Averages can look good, however, while the actual distributions can be disastrous.

Two distributions of interest are the amount of data contained in a cell, and the percentage of computation occurring in a cell. A few big data cells can clog the load balancing mechanism, and a few heavy computational cells can become bottlenecks. Cleary if there exists a cell that can fill up a processor's memory, then whichever processor gets the cell will be stuck with whatever load is required by that cell.

Sufficiently small size granularity is needed for load balancing.

Likewise if there exists a cell that requires more computation than one processor's share of the total, that cell will be a bottleneck limiting the machine's performance. This can be dealt with by duplication, but that requires more complicated mapping to allow multiple ownership.

Even if cells are small enough so that the average computation per cell over the entire run is sufficiently small, there are only a limited number of rays active in the machine at any particular point in the computation. For load balancing at any particular instant, only the computations required by the set of active rays is relevant. If the load presented by the set of active rays falls onto a small number of cells, then the load balancer will once again be unable to distribute the load evenly. Rather than generate rays in scanline order, rays must be distributed across the image to make sure they don't overuse any cell.

Super Cell Size Distribution						
Scene	80K-60K	60K-40K	40K-20K	20K-10K	10K-5K	5K-0K
Tetra	0	0	0	35	270	263
Arch	0	0	0	10	6	1315
Arch+Gumby	0	1	0	7	19	2868
Gearbox	4	11	20	28	37	985

Table 4.1

Sub Cell Size Distribution						
Scene	80K-60K	60K-40K	40K-20K	20K-10K	10K-5K	5K-0K
Tetra	0	0	0	0	0	8980
Arch	0	0	0	0	6	20618
Arch+Gumby	0	0	0	2	7	45113
Gearbox	0	0	2	66	185	17683

Table 4.2

After the operating system, the program, and the message buffers take their share of each node's memory, there is only 100K bytes left for storing data. Tables 4.1 and 4.2 show the distribution of cell sizes for super cells and sub cells respectively. The vast majority of the cells contain less than 10K of data, however for the more complex scenes there are a few super cells larger than 40K. These cells must be broken up. Table 4.2 shows that even the sub cells can be large, but none of them exceeded 40K in the scenes

studied.

Tables 4.3 and 4.4 show a metric of the amount of computation occurring in the cells. Again, the majority of the cells are well behaved but a few contain a large fraction of the total computation. If a single cell contains 1% of the total computation, speedup is clearly limited to 100 regardless of the number of processors available. Since there are many sub cells that contain more than 1% of the computation, data replication techniques will clearly be required for large machines. Even with only 32 processors, there are single sub cells that constitue several percent of the total computation, and thus will severly restrict the ability to load balance.

Super Cell Usage (% of total intersections)						
Scene	15%-10%	10%-5%	5%-2%	2%-1%	1%-0.5%	0.5%-0%
Tetra	0	0	0	4	38	508
Arch	1	6	7	4	4	182
Arch+Gumby	1	4	7	14	9	649
Gearbox	0	2	11	17	16	913

Table 4.3

Sub Cell Usage (% of total intersections)						
Scene	15%-10%	10%-5%	5%-2%	2%-1%	1%-0.5%	0.5%-0%
Tetra	0	0	0	0	0	7201
Arch	0	0	4	20	28	1134
Arch+Gumby	0	0	1	14	24	2971
Gearbox	0	0	0	2	20	9944

Table 4.4

For reasons of both size granularity and computational granularity, some super cells must be broken into sub cells for load balancing. For space reasons, the super cell structure is required. The conclusion is that both level of cells will have to be used as load balancing units. This suggests that a full oct-tree structure would be desirable and perhaps not much more complex. Oct trees would provide greater flexibility to subdivide until sufficiently fine spatial and computationl granularity is achieved.

4.4. Load Balancing

The results from the previous section shows that load balancing is necessary except for specially constructed artificial scenes. To do load balancing, one needs both a policy and a mechanism. The policy is used to decide when and what to load balance. The mechanism must be capable of implementing the policy decision.

A load balancing policy should be stable. This means that under constant load, the system will reach a point where load balancing is not needed. This is only possible within the precision of the load balancing unit. If large portions of the load must be shifted, any shift will necessarily leave the load unbalanced.

The load balancing mechanism selected is the shifting of responsibility for cells in the subdivision. When a cell is moved between processors, the global routing tables become invalid. There are two simple approaches to deal with this. The most obvious approach is to broadcast the ownership change to all processors. The other approach is to allow the routing tables to become inconsistent and simply forward messages to the correct processor. The sequence of previous owners of a cell will form a chain that leads to the current owner. The broadcasting approach is wasteful since many of the distant processors will never send a message to the cell and therefore never need to know that the cell was moved. The forwarding approach is wasteful since each time a cell is migrated, all future references to it will require an extra message forwarding operation.

A compromise approach is to forward messages, but also notify the ray originator of the changed ownership. This limits updates to only those processors that need to know that a cell has migrated. In practice this works extremely well. The new and old owners of a cell automatically know that it has moved. On average only 0.6 relocation messages per migrated cell are sent to other processors.

4.4.1. Local Policy

My original experiments with load balancing policies were with local policies. Each processor tried to balance its load against the loads of its neighbors. Since all processors are trying to balance their loads with their neighbors, after several iterations a balance across the entire machine should be achieved. Each processor monitored its own utilization and occasionally compared it against the utilization of its neigh-

bors. When a processor noticed a significant difference in loads between itself and a neighbor, it would arrange to balance their loads.

A fundamental problem arose with this method that was unsolvable. When two processors rendezvoused to transfer cells between them and attempt to balance their loads, the rest of the processors would continue to send them rays. Selecting just a single cell and performing the transfer could take long enough that incoming messages from other processors could congest the machine and cause deadlock. The local load balancing policies eventually had to be abandoned in favor of a global policy.

4.4.2. Global Policy

The global policy is to make a global decision that load balancing is needed, stop all of the ray-tracing activity, adjust the load distribution and then resume tracing rays. Since all processors are stopped together, it was felt that a single global redistribution of data should be used instead of the many local redistributions that would otherwise be required.

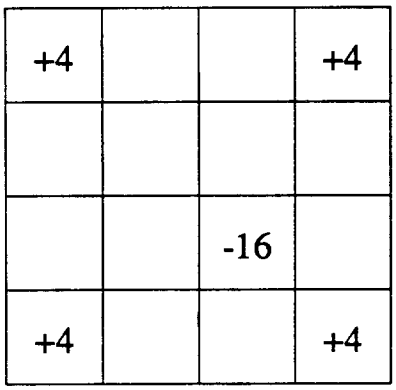

Figure 4.6a. Desired Data Changes

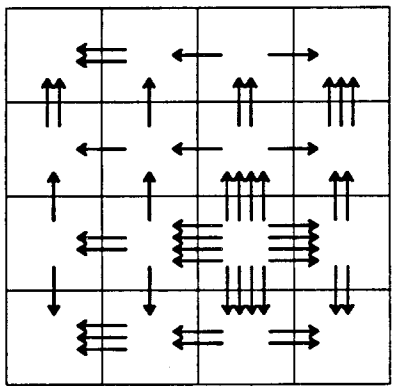

Figure 4.6b. Resulting Transfers

The goal of load balancing is for all processors to finish their share of the work at the same time and to stay busy for the entire run. At each pause for load balancing, the processors send information about their workload, progress toward completion, and cpu utilization to a central node (node 0 was chosen arbitrarily). The central node calculates how the workload on each processor should change to bring about a balanced load and how to redistribute the workload around the machine. The results of this calculation are

then sent to the processors, and the processors arrange with their neighbors to transfer the data between them. Figures 4.6a and 4.6b show an example. Figure 4.6a shows a redistribution map where each square represents a processor and the numbers indicate the *amount of data* to be transferred. Figure 4.6b shows the transfer map that would be calculated.

The *amount of data* to be transferred can be measured in many ways. Each processor determines by what percentage the workload should change and it transfers that portion of its cells which it believes will provide the desired workload change. Ideally one would like to measure cell workload by the amount of computation that uses each cell. This would require sub-sampling the image at uniform density and gathering usage statistics for every cell. This is a difficult statistic to measure. Instead, one can make the simplifying assumption that all cells are used equally, or better, that all objects are used equally, or better yet, that all objects are used in proportion to their size. All of these assumptions are inaccurate but easy to measure. I chose to measure usage in proportion to object size, and scaled this separately for each processor by the processor's utilization.

One can make a comparison between static and dynamic load balancing. With a perfect load metric, dynamic load balancing reduces to static load balancing because the load will be balanced by a single load balancing phase. For a less accurate metric dynamic load balancing has the additional ability to measure the effectiveness of its previous attempts and try again. I hoped that after several attempts a poor but easy to measure load metric could be used to balance the load. The results were disappointing.

4.4.3. Results

Compare figure 4.7 to figure 4.4. Figure 4.4 shows the Arch scene run without load balancing. Because many of the processors sit idle the speedup relative to the serial program was only 2.9. When load balancing was added (figure 4.7) the speedup was improved to somewhere between 4.0 and 4.8. The larger figure results if one ignores the time spent for load balancing. This is valid since a small image was used for testing purposes. On a full size image the time spent for load balancing will stay the same, but the time spent ray-tracing will increase by a factor of 25. The time spent load balancing becomes negligible. These results also ignore start up costs for the same reason.

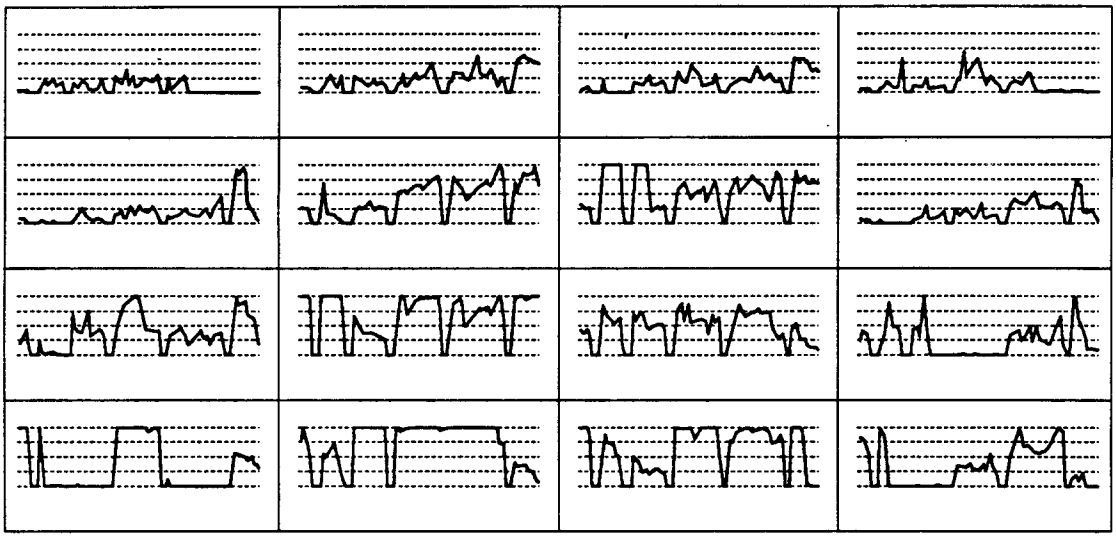

Figure 4.7. Processor Utilization - With Load Balancing

In figure 4.7 load balancing was performed five times. These phases can be identified because the utilization (as far as ray-tracing is concerned) drops to 0. The five dips are easily visible for the processors in the center of the chart. Notice that the load shifts around but never becomes well balanced. In the process of load balancing, the memory of some processors fills up before an adequate workload is obtained.

The arch is the smallest of the test scenes. Memory is clearly inadequate for the other scenes. There are actually 32 processors on our hypercube, which if all used would provide twice as much memory as 16 processors provide. Unfortunately, as the number of processors is increased, more buffer space is needed to avoid deadlock. The memory that would have been gained by adding more processors is lost to provide sufficient buffers. Substantially more memory is needed per node.

4.5. Conclusions

This section has described my attempts at parallelizing a sophisticated ray-tracer. Dealing with realistic scenes that have non-uniform scene density provides many challenges. The chaotic flow of rays through the scene is inherently deadlock prone. An adequate ray regulation method was found which avoids deadlock, but it requires substantial buffer space. Load balancing was only moderately successful because of the limited memory available. Given the complexity of dynamic load balancing, static load balancing seems to hold more promise.

The data caching approach was not implemented. It remains an open problem to determine if this approach will work when adequate memory is available, and to determine how much memory is needed relative to the scene size.

5. Conclusions

The first conclusion that can be drawn from this thesis is that parallel ray-tracing is a much harder problem than it is generally believed to be. Toy implementations with simple artificial scenes do not encounter the difficulties that occur when realistic scenes are used. Real scenes have both substantial complexity and non-uniformity. Production quality ray-tracers are even more complex than the one used in this study. Typically they have fancier antialiasing, more types of primitive objects, and texture mapping. Dealing with the large amount of texture data is a serious problem which still needs to be studied. Diversity in object types and surface properties presents a serious problem for SIMD implementations [Del88].

Second, shared memory machines are much easier to use than distributed memory machines (at least for applications where a large shared data base is involved.) Converting the program to run on the Sequent took a few months compared to nearly a year spent on the Hypercube version. The original ray tracer was 10,000 lines of code. The Sequent version had 400 lines of code modified or added. The hypercube version had 6,000 lines modified or added. The Sequent version is robust and achieves linear speedup, and it has been used for producing thousands of images for an animated video. The hypercube version is unreliable and achieves only moderate speedups (due in part to insufficient memory).

Third, it is unreasonable to require a programmer to manage a complex distributed data structure. The code to pull apart linked structures, package them into messages, and unpackage them at the recipient was laborious. Adding failure recovery when memory filled made this coding even more difficult. Providing a shared memory model to a programmer is acceptable; providing a distributed memory model is not.

Finally, load balancing on a shared memory machine through dynamic scheduling of many small jobs is a simple and reliable technique. To balance workloads on a distributed memory machine, as attempted on the Hypercube, requires extensive programming as well as careful measurement, and tuning.

The sole advantage of distributed memory machines is that they can be scaled to large numbers of processors at almost linearly increasing cost. If large shared memory machines were available, ray-tracing would be a well suited application.

Ray-tracing on a shared memory machine should be considered a nearly solved problem. Three issues not included in this research are higher order surface primatives, texture mapping, and stochastic

antialiasing. Using higher order surface primatives will increase the computation time, but will not otherwise affect performance. Dealing with large amounts of texture data will probably not be any more troublesome than it is for uniprocessors. Scheduling stochastically antialiased image samples is the most difficult open issue.

Ray-tracing on a distributed memory machine remains unsolved. The attempts at load balancing tried in this research were unsuccessful. Whether other approaches will work remains to be seen.

6. References

[Amd67] G. M. Amdahl, Validity of the Singe Processor Approach to Achieving Large Scale Computing Capabilities, *Proceedings AFIPS 1967 Spring Joint Computer Conference*, April 1967, 483-485.

[BKT87] B. Beck, B. Kasten and S. Thakkar, VLSI Assist For A Multiprocessor, *ASPLOS II 15*, 5 (October 1987). published as part of Computer Archtecture News.

[Bel85] C. G. Bell, Multis: A New Class of Multiprocessor Computers, *Science 228* (April 26, 1985).

[CWB86] J. G. Cleary, B. M. Wyvill, G. M. Birtwistle and R. Vatti, Multiprocessor Ray Tracing, *Computer Graphics Forum 5*, 1 (March 1986), 3-12.

[Del88] H. Delany, Ray Tracing On A Connection Machine, *1988 International Conference on Supercomputing*, July 1988, 659-667.

[DiS84] M. Dippe and J. Swensen, An Adaptive Subdivision Algorithm and Parallel Architetcure for Realistic Image Synthesis, *SIGGRAPH '84 18*, 3 (July 1984), 149-158.

[DoM87] G. B. Doshi and E. V. Munson, Benchmarking the Cubes, *CS252 Spring 1987 projects I* (Spring 1987).

[Gus88] J. L. Gustafson, Reevaluating Amdahl's Law, *Communication of the ACM 31*, 5 (May 1988), 532-533. Technical Note.

[HaG86] E. A. Haines and D. P. Greenberg, The Light Buffer: A Shadow-Testing Accelerator, *IEEE Computer Graphics and Applications 6*, 9 (September 1986), 6-16.

[Hyp86] Hypercube, *iPSC Program Development Guide*, Intel Corporation, 1986.

[JoB86] K. Joy and M. Bhetanabholta, Ray Tracing Parametric Surface Patches Utilizing Numerical Techniques and Ray Coherence, *SIGGRAPH '86 20*, 4 (August 1986), 279-285.

[Lia87] C. Liao, *Parallel Bundle Tracing*, University of New Hampshire, December 1987. Masters Thesis.

[Mar87] D. Marsh, *UgRay - An Efficient Ray-Tracing Renderer for UniGrafix*, U. C. Berkeley, May

1987. Masters Thesis.

[NOK83] H. Nishimura, H. Ohno, T. Kawata, I. Shirakawa and K. Omura, LINKS-1: A Parallel Pipelined Multimicrocomputer System for Image Creation, in *Proceedings of the 10th Symposium on Computer Architecture*, vol. 11 , SIGARCH, 1983, 387-394.

[Pet87] J. W. Peterson, *Distributed Computation for Computer Animation*, University of Utah, June 1987. Tech. Rep. UUCS 87-014.

[Pot88] M. Potmesil, *personal communication*, AT&T Pixel Machines, February 1988.

[PrB88] T. Priol and K. Bouatouch, Experimenting with a Parallel Ray-Tracing Algorithm on a Hypercube Machine, *Eurographics*, 1988, 243-259.

[Sat85] H. Sato, Fast Image Generation of Constructive Solid Geometry Using a Cellular Array Processor, *SIGGRAPH '85 19*, 3 (July 1985), 95-102.

[Sei85] C. L. Seitz, The Cosmic Cube, *Communications of the ACM 28*, 1 (January 1985), 22-33.

[Seq86] Sequent, *Balance Guide To Parallel Programming*, Sequent Computer Systems, Inc., 1986.

[SDB85] R. L. Speer, T. D. DeRose and B. A. Barsky, A Theoretical and Empirical Study of Coherent Ray-tracing, in *Proceeding of Graphics Interface '85*, Montreal, May 1985, 1-8.

[TPP85] J. Tuazon, J. Peterson, M. Pniel and D. Liberman, CALTECH/JPL MARK II Hypercube Concurrent Processor, *Proceedings of the 1985 Internation Conference on Parallel Processing*, 1985, 666-673.

[Ull83] M. K. Ullner, *Parallel Machines for Computer Graphics*, California Institute of Technology, January 1983. Computer Science Technical Report 5112.

[Whe85] D. S. Whelan, *A Multiprocessor Architecture for Real-Time Computer Animation*, California Institute of Technology, 1985. Ph.D. Thesis.

[Wol89] R. W. Wolff, *Stochastic Modeling and The Theory of Queues*, Prentice Hall, 1989.

7. Appendix

Example Ray-Traced Images			
Scene	Faces	Sequential Run Time (1 MIP processor)	Comments
Tetra	4096	8 minutes	no reflections not antialiased
Arch	375	17 minutes	flat faces
Arch+Gumby	558	27 minutes	smooth shaded faces (similar to curved faces)
Gearbox	7169	13 hours	glass with many internal reflections