Chapter 6

A Multiprocessor Volume Rendering Algorithm

The new volume rendering algorithms described in the previous chapters enable interactive visualization applications on current-generation workstations, but they are not fast enough to provide real-time frame rates (10-30 Hz). Real-time rendering allows smooth user interfaces and instantaneous feedback when exploring a data set. Furthermore, a fast update rate during a rotation sequence provides strong depth cues that can greatly aid the human visual system to understand a data set.

Four approaches have been proposed in the literature to achieve high frame rates for volume rendering. The first approach is to trade off image quality for speed, for instance by using low-resolution versions of the data during user interaction [Laur & Hanrahan 1991]. Such methods allow rough manipulation of the data but the loss of resolution eliminates many perceptual cues. The second approach is to rely on special-purpose hardware designed either for graphics applications or specifically for volume rendering. Recent work in this area includes volume rendering algorithms for the Pixel-Planes 5 graphics architecture [Yoo et al. 1992] and for 3D texture mapping hardware [Cabral et al. 1994], a custom accelerator board for volume rendering [Knittel & Straßer 1994], and the Cube-3 architecture [Pfister et al. 1994]. Designs such as these can deliver high-quality images at high frame rates; however, algorithms that are implemented in hardware lose flexibility.

The third approach is to parallelize a brute-force volume rendering algorithm for a large multiprocessor. Many volume rendering algorithms have been reported for massively-parallel SIMD multiprocessors [Cameron & Undrill 1992, Schröder & Stoll 1992, Vézina et al. 1992, Hsu 1993, Wittenbrink & Somani 1993]. These architectures perform best when communication patterns are regular and uniform, so the best speedups are achieved with simple brute-force rendering algorithms. Brute-force parallel algorithms for large MIMD machines have also been reported [Ma et al. 1993, Corrie & Mackerras 1993b, Montani et al. 1992]. This solution requires a large number of processors to achieve acceptable frame rates.

The fourth approach is to parallelize a fast rendering algorithm that relies on algorithmic optimizations, such as spatial data structures, to reduce the computational costs [Nieh & Levoy 1992, Neumann 1993]. A MIMD multiprocessor must be used since intricate algorithms with many special cases do not map well onto SIMD arrays. The disadvantages of this approach are that the optimizations are data-dependent and preprocessing is often necessary. However, less-expensive hardware is required to achieve the same frame rates as with the other approaches, so using an optimized algorithm is more cost effective. This is the approach we adopt.

In this chapter we describe a new parallel volume rendering algorithm that performs well due to three factors. First, the computational costs are lower than previously-published parallel algorithms because the algorithm is based on the serial shear-warp rendering algorithms described in Chapter 4. Furthermore, we have designed our parallelization using an image partition that takes full advantage of the optimizations in the rendering algorithm.

Second, we control synchronization and idle waiting costs by using the dynamic load balancing method described by Nieh & Levoy [1992]. Our task partition also minimizes the number of synchronization events, thereby making load balancing easier.

Third, our algorithm has low data communication costs because it is implemented for cache-coherent shared-memory multiprocessors. This type of parallel architecture supports fine-grain low-latency communication, unlike message-passing machines which require expensive operating-system intervention for every message and therefore demand careful data distribution [Neumann 1993, Montani et al. 1992, Corrie & Mackerras 1993b].

We have implemented our algorithm for two different MIMD shared-memory architectures: the Silicon Graphics Challenge multiprocessor, which is a bus-based architecture, and the Stanford DASH Multiprocessor, a scalable distributed shared-memory multiprocessor. Our implementation on a 16 processor Challenge renders a 256^3 volume at 13 Hz and a 128^3 volume at 31 Hz. These rates are comparable to results reported for special-purpose graphics hardware [Yoo et al. 1992] and exceed the best reported results for massively-parallel SIMD machines [Hsu 1993, Wittenbrink & Somani 1993] and much larger MIMD message-passing machines [Neumann 1993]. On a 32-processor configuration of DASH the algorithm renders a 256^3 volume at 5.9 Hz, six times faster than the results for DASH reported by Nieh & Levoy [1992].

Our performance results lead to several conclusions. First, as others have noted, data redistribution and communication costs do not dominate rendering time on a shared-memory machine [Nieh & Levoy 1992]. Second, we find that cache locality requirements impose a limit on parallelism in volume rendering algorithms. Caches hide latency by exploiting locality of reference, but locality is destroyed if the task size is too small. In our experiments our algorithm approaches this limit, and we believe that for any volume rendering algorithm based on an image partition shared-memory machines with hundreds of processors would be useful only for very large data sets.

Section 6.1 of this chapter describes the multiprocessor algorithm and the design tradeoffs we considered. Section 6.2 describes the architectures of the Challenge and DASH multiprocessors and the details of our implementation for these machines. We then present performance results for the parallel algorithm in Section 6.3 and discuss the limitations on parallelism. Finally we discuss related work and our conclusions.

6.1 Multiprocessor Rendering Algorithm

Designing a parallel algorithm for a shared-memory multiprocessor consists of choosing a partition of the computation into tasks for each processor and choosing appropriate synchronization mechanisms. The serial algorithm contains three phases, each of which must be parallelized: computing the shading lookup table, projecting the volume into the intermediate image, and warping the intermediate image. We begin by focusing on the projection of the volume data since that phase dominates the cost of the serial algorithm, and then we consider the issue of data distribution and the complete parallel algorithm.

6.1.1 Image and Object Partitions

The two general types of task partitions for parallel volume rendering algorithms are object partitions and image partitions. In an object partition each processor is assigned a specific subset of the volume data [Neumann 1993, Ma et al. 1993]. The processors first resample and composite the voxels assigned to them, and then the partially-computed image pixels must be composited with results from other processors to form the final image pixel values. In contrast, in an image partition each processor's task is to compute a specific portion of the image [Nieh & Levoy 1992, Corrie & Mackerras 1993b]. Each image pixel is computed by only one processor, but the volume data must be moved to different processors as the viewing transformation changes. The choice of an image partition versus an object partition is independent of the choice of an image-order or object-order rendering algorithm.

Neumann [1993] shows that object partitions require less data redistribution than image partitions because only the image is communicated, while the much larger 3D volume is never redistributed. On a message-passing machine the cost of communication is high due to operating-system involvement in every message, so object partitions appear attractive. However, on shared-memory machines the cost of communication may not be dominant [Nieh & Levoy 1992] and therefore other factors are also important. One disadvantage of object partitions is that early ray termination becomes less effective as the number of processors increases. An increasing percentage of the processors are assigned occluded portions of the data and therefore either perform unnecessary computation or have no work to do. Early ray termination can result in 2-3x performance improvements as we saw in Chapter 5, so 50% or more of the processing power of a large parallel machine may be wasted with an object partition.

A second potential disadvantage of object partitions is that increased synchronization is necessary. In an object partition multiple processors must write to the same portion of the image, so the processors must explicitly synchronize these writes to avoid erroneous results. Less synchronization is required in an image partition because only one processor writes to a given portion of the image; the only shared data structure is the volume data, which is read-only and therefore requires no synchronization.

Algorithms that require fine-grain synchronization often suffer from high percentages of idle waiting time and load-balancing difficulties. For example, in an early attempt to parallelize the shear-warp algorithm we used an object partition in which each processor was assigned a portion of each slice of voxels [Agrawala 1993]. To maintain the required compositing order all of the processors were required to synchronize at a barrier after each slice had been composited into the intermediate image. The result was poor speedup due primarily to idle waiting time at the barriers. Load balancing could not compensate because of the small amount of work between each synchronization event. Challinger [1992] reports results for an object partition on a shared-memory machine that agree with these conclusions.

The binary-swap image compositing algorithm proposed by Ma et al. [1993] also appears to have high synchronization overhead: the image-compositing phase suffers from low speedups. Ma's overall parallel volume rendering algorithm has good speedups because the cost of compositing is far less than the cost of rendering, but the lower cost of rendering in our algorithm makes the cost of compositing more significant.

We therefore choose an image partition, which requires no synchronization until the entire volume has been composited.

6.1.2 Task Shape

There are several choices for the unit of work in an image partition: individual pixels, scanlines of pixels [Challinger 1992, Yoo et al. 1991], or rectangular tiles [Nieh & Levoy 1992]. Generally, tile-shaped tasks are best in order to maximize temporal cache locality [Nieh & Levoy 1992, Corrie & Mackerras 1993b]. However, two factors unique to the run-length encoded data structures in the shear-warp algorithm lead us to partition the intermediate image into groups of contiguous scanlines. First, the run-length encoded volume data structure makes it difficult to find the voxels associated with an image tile. Finding the data associated with an arbitrary voxel requires decoding the run-length encoding for all of the voxels preceding it. In a scanline-based task partition we can precompute pointers to the beginning of each scanline of voxel data to eliminate all decoding overhead, as mentioned in Section 4.1.2. In a tile-based task partition pointers would be required for every voxel since in an object-order rendering algorithm an image tile can potentially map to any voxel (depending on the view transformation).

The second reason for choosing scanline-shaped tasks instead of tiles is to maximize spatial locality in both the intermediate image and the run-length encoded volume data. If the compression ratio in the run-length encoded volume is high then several scanlines of voxels may fit in a single cache line and it is advantageous to process a contiguous group of scanlines as a unit. The optimal number of scanlines in a task depends upon the cache line size and will be discussed further in Section 6.3.

6.1.3 Load Balancing

Given that the fundamental unit of work is a group of contiguous scanlines of the intermediate image, we must choose how to assign the scanlines to processors while minimizing load imbalances. We consider three options: a static contiguous partition, a static interleaved partition, and a dynamic partition.

In a *static contiguous partition* the image is divided into large fixed blocks of scanlines, one block per processor. Such a scheme works well only if the execution time for each block can be accurately predicted so that each processor has the same amount of work. However, view-dependent variations in the effectiveness of the coherence optimizations makes it difficult to predict performance. Our experiments with a simple measure of work, the number of nonzero voxels in a scanline, showed it to be an effective predictor of execution time [Agrawala 1993].

In a *static interleaved partition* the image is divided into small blocks of scanlines and the blocks are distributed in round-robin order to the processors. This scheme introduces a tradeoff between load balancing and memory performance. If the image is divided into a small number of large tasks then it is likely that some processors will be assigned more expensive tasks than others and the load balance will be poor. On the other hand, if the image is divided into a large number of small tasks then spatial locality is reduced and memory overhead rises.

In a *dynamic partition* tasks may be reassigned to different processors during rendering to optimize the load balance. This method results in the best load balance, but care must be taken to avoid overhead caused by redistributing tasks. The method we consider employs a distributed task queue and dynamic task stealing [Nieh & Levoy 1992]. The image is first partitioned using one of the static partitioning methods and the tasks are placed on task queues associated with each processor. If a processor completes all of its tasks then it may "steal" a task from one of the other queues. Distributed queues are used rather than a single centralized queue in order to avoid a bottleneck.

We implemented all of the above methods and found that dynamic task stealing with a moderately interleaved initial task distribution performed best, as our performance results will show. The interleaved partition increases the probability of a good initial load balance, and dynamic task stealing compensates in cases where the initial partition is not optimal.

6.1.4 Data Distribution

Explicit data distribution is a difficult problem when an image partition is used because the portion of the volume required by a particular processor depends on the viewpoint. One solution is to replicate the data on every processor's memory, but this design severely limits the maximum size of the volume. However, explicit data distribution is not necessary on a shared memory machine with caches. Because all of the processors share the same address space, data in any memory can be accessed transparently by any processor. The performance penalty for remote accesses is small provided the cache hit rate is sufficiently high and there are no memory hot spots. To avoid hot spotting we use a round-robin distribution of all large data structures to the different physical memories. On a shared-memory machine the hardware caches automatically replicate the subset of the data required by multiple processors. The memory performance of the algorithm will be analyzed in the results section.

6.1.5 Overall Algorithm

The pseudo-code in Figure 6.1 shows the flow of control for each processor in the parallel algorithm. First, each processor selects a portion of a lookup table used for shading voxels (parallelized with a static contiguous task partition) and then waits at a barrier for the other processors to finish. Next, each processor computes a portion of the intermediate image (image_scans) by choosing a task from one of the task queues (FindTask()). Each task entails

```
procedure MultiRenderVolume()
    ComputeShadingLookupTable(); { static contiguous partition }
    barrier
    while (not done)
        image_scans = FindTask(); { dynamic partition }
        if (image_scans ≠ NIL)
            Clear(image_scans);
            foreach (voxel_slice from front to back)
                voxel_scans = Intersect(image_scans, voxel_slice);
                image_scans = image_scans over Resample(Shade(voxel_scans)));
            end
        else
            done = 1;
        endif
    end
    barrier
    image = Warp(); { static interleaved partition }
    barrier
    Display(image);
end
```

Figure 6.1: Pseudo-code for the multiprocessor algorithm.

looping through the slices of the volume, finding the voxel scanlines (voxel_scans) that intersect the assigned portion of the intermediate image, and then resampling and compositing the voxel scanlines. When all of the tasks have been completed each processor synchronizes at the second barrier, warps a portion of the image (parallelized with a static interleaved partition consisting of rectangular tiles of the final image), and synchronizes again. Then the image is displayed.

Apart from the three barriers, the only synchronization required is a set of locks to govern access to the task queues in the FindTask() function. Each queue has its own lock, and processors request tasks from their own queue except when stealing, so the contention for the locks is negligible.

Architecture	SGI Challenge	Stanford DASH
Nodes	16	32
Processor	150 MHz R4400	33 MHz R3000
1st Level D-Cache	16 Kbytes	64 Kbytes
2nd Level D-Cache	1 Mbytes	256 Kbytes
2nd Level Line Size	128 bytes	16 bytes
Interconnect Type	Global Bus	2D Mesh
Interconnect Bandwidth	1.2 Gbytes/sec.	960 Mbytes/sec.

Table 6.1: Characteristics of the parallel architectures used for the performance measurements. Both machines are shared-memory multiprocessors with hardware cache-coherence.

6.2 Implementation

6.2.1 Hardware Architectures

We have implemented the algorithm for two architectures, the Silicon Graphics Challenge and the Stanford DASH Multiprocessor. Table 6.1 lists the characteristics of the machines.

The SGI Challenge is a bus-based symmetric shared-memory multiprocessor. A global bus allows any processor to access any memory bank with the same latency. DASH is an experimental prototype of a scalable distributed shared-memory architecture. The machine consists of processor clusters connected by a 2D mesh network (Figure 6.2). Each cluster contains four CPUs and a memory module on a local bus. The processor caches are kept consistent by the hardware using a distributed directory-based protocol [Lenoski et al. 1993]. The mesh network in this design is scalable, whereas in a bus-based design the bus saturates if too many processors are attached.

From the parallel programmer's perspective, both architectures implement a shared-memory model. However, the memory hierarchies are significantly different. The secondary cache on the Challenge has a long cache line size (128 bytes) so that programs with good spatial locality benefit from a prefetching effect which helps to mask the latency of main memory accesses. The short cache lines on DASH lessen the benefit of spatial locality. The caches on DASH are also much smaller. Finally, since DASH is a distributed shared-memory machine the memory latency increases if an access must be serviced by a remote memory.

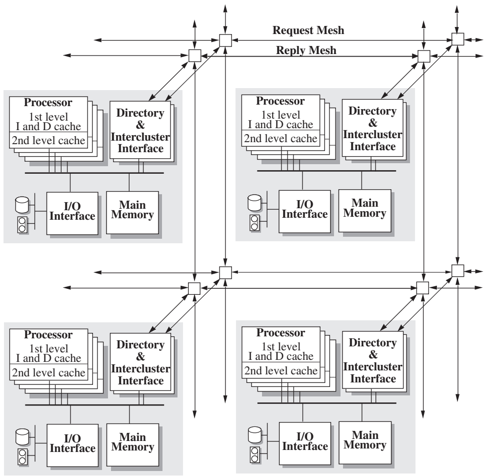


Figure 6.2: Block diagram of a 2 x 2 DASH system (courtesy of Daniel Lenoski, Computer Systems Laboratory, Stanford University).

6.2.2 Software Implementation

We implemented the algorithm using the C language and the ANL parallel programming macro package [Boyle et al. 1987]. Because of the shared-memory programming model the parallel code is almost identical to the original serial code. The only additions are functions to compute task partitions and a few calls to the synchronization primitives. The same source code runs without modification on both the Challenge and DASH architectures and is portable to other shared-memory machines.

All of the large data structures, including the run-length encoded volume, the intermediate image, the final image, and the shading lookup table are allocated in the shared address space. In order to avoid hot spotting, on DASH we distribute the shared data structures to the different memories by assigning pages (4096 byte blocks) to memories in round-robin fashion. On the Challenge the memory system is eight-way interleaved, so round-robin distribution of the data structures is performed automatically at cache-line granularity.

The memory requirements of our implementation are four bytes per non-transparent voxel (including a precomputed opacity and normal vector for shading) and twelve bytes per pixel for a gray scale intermediate image (including an intensity and an opacity stored in floating point format, and storage for the run lengths). Three copies of the volume data are stored in memory, one for each possible transpose, as in the serial algorithm.

We shade the voxels using the Phong illumination model implemented with an 8192-entry lookup table (further described in Chapter 7). The table is recomputed at the start of each frame to allow view-dependent directional lighting effects.

The task sizes for the compositing phase and the warping phase of the algorithm are parameters which can be changed at run-time. The task sizes must be adjusted to find the optimal tradeoff between load balancing and memory overhead on each architecture.

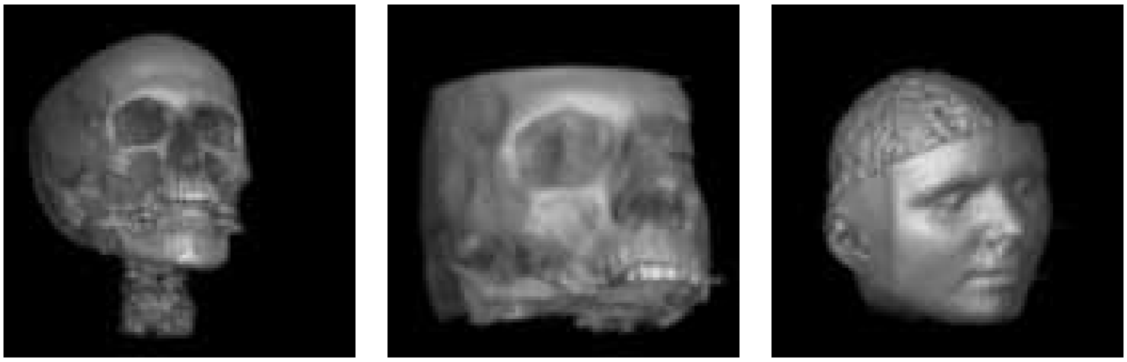
Our current implementation only supports parallel projections and preclassified volumes. The perspective version of the rendering algorithm and the fast classification algorithm can be parallelized using the same method described here.

6.3 Results

We tested our algorithm using two data sets: a 256x256x225 voxel CT scan of a human head and a 128x128x109 voxel MR scan of a head. These are the "head" and "brainsmall" data sets described in Table 5.1. Figure 6.3 shows renderings of the test data sets. The images are gray scale and contain 256^2 pixels (although the rendering times are relatively independent of the image size, just as in the serial algorithm). On the Challenge we parallelized the image warp using tiles with 8x256 pixels and on DASH we used 16x16 tiles. We used an early ray termination opacity cutoff of 95%. The total memory usage of the parallel algorithm is the same as the serial algorithm (Table 5.3). Figure 6.4 illustrates a graphical interface that allows the user to rotate the volume in real time by dragging on the rendered image.

6.3.1 Rendering Rates

Tables 6.2 and 6.3 give our performance results on the Challenge and on DASH. These results do not include image display and preprocessing to compute the run-length encoded volume. On the Challenge our display routine (implemented with Tcl/Tk [Ousterhout 1994] and GL [Silicon Graphics 1991]) requires 3 msec. to display a 256^2 image. The preprocessing time is 65 sec. on a single processor for the 256^3 volume. The frame rates in the tables are averages for a 180-frame animation with a two-degree rotation of the volume between



(a) (b) (c)

Figure 6.3: Test data sets for the multiprocessor algorithm: (a) CT head data set (256 x 256 x 225 voxels) (b) subvolume extracted from the CT head data set (128 x 128 x 128 voxels) (c) MR brain data set (128 x 128 x 109 voxels).

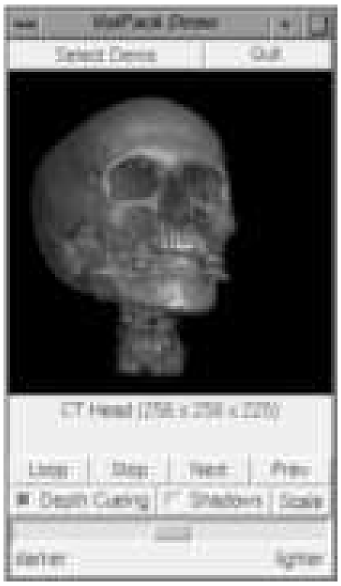


Figure 6.4: The interactive user interface for the multiprocessor algorithm. The user interface allows the volume to be rotated in real time by direct manipulation with a mouse. Sliders (not shown) allow interactive adjustment of shading and lighting parameters.

Data set	Processors				
	1	2	4	8	16
CT Head (128x128x128)					
Frame Rate (Hz)	2.5	5.3	9.9	18	31
Task Size (scanlines)	256	8	8	8	8
CT Head (256x256x225)					
Frame Rate (Hz)	1.1	2.2	4.3	7.5	13
Task Size (scanlines)	512	20	20	20	13

Table 6.2: Rendering rates and tasks sizes on the SGI Challenge. The task size is the number of intermediate image scanlines per task for the projection stage of the parallel algorithm.

Data set	Processors					
	1	2	4	8	16	32
MRI Brain (128x128x109)						
Frame Rate (Hz)	1.2	2.4	4.4	7.0	12	18
Task Size (scanlines)	32	20	12	12	5	3
CT Head (256x256x225)						
Frame Rate (Hz)	0.33	0.67	1.3	2.1	3.4	5.9
Task Size (scanlines)	32	20	12	12	8	3

Table 6.3: Rendering rates and task sizes on the Stanford DASH Multiprocessor.

each frame. The fastest rates are achieved with the Challenge (which has more total computational power than DASH): 13 Hz for the 256^3 volume.

The speedup curves for the algorithm are not ideal, as will be discussed later in this section. Nevertheless, these rendering rates are among the fastest reported. In comparison to the results reported by Nieh & Levoy [1992] on the DASH architecture, our parallel algorithm is faster by a factor of six for a 32-processor machine. It is also competitive with results for much more expensive massively parallel processors and special-purpose hardware. The multiprocessor shear-warp algorithm achieves these results only for data sets with high coherence, but as we argued earlier in Section 5.3.4 classification functions that produce low coherence also produce poor visualizations.

Color renderings take roughly twice as long as gray scale renderings, just as in the serial algorithm. We rerendered the color images shown in Figure 5.5–5.7 on DASH with 32 processors. The CT head (256 x 256 x 225 voxels) was rendered at a rate of 2.0 Hz, the CT engine (256 x 256 x 110 voxels) was rendered at a rate of 2.8 Hz, and the CT abdomen (256

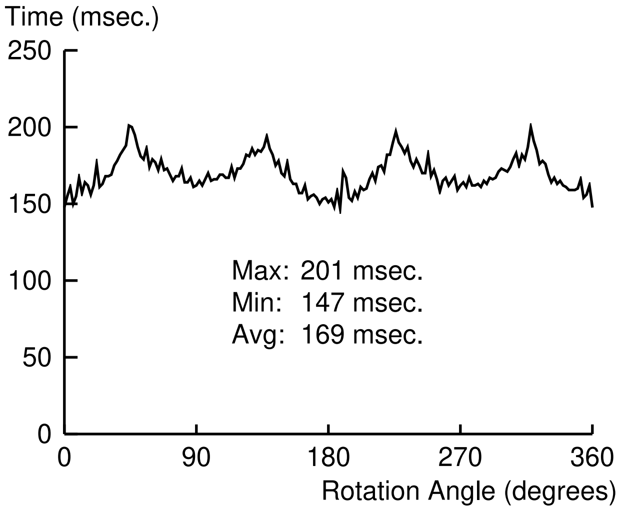


Figure 6.5: Rendering time for the 256^3 gray scale head on DASH with 32 processors as the viewing angle changes.

x 256 x 159 voxels) was rendered at a rate of 2.4 Hz.

Figure 6.5 shows how the rendering time varies with viewing angle during a continuous 360-degree rotation sequence. The variations are due to data-dependent changes in image complexity. There is no discontinuous jump in rendering time when the volume is rotated past a 45-degree point, even though the renderer must switch to a different transposed copy of the run-length encoded volume. Neither is there any visual discontinuity in the rendered images. The rendering time does not change if the rotation angle between frames is increased, indicating that the performance results do not depend on cache coherence between frames.

6.3.2 Performance Limits on the Challenge Multiprocessor

Figure 6.6 shows the speedup curve for the Challenge multiprocessor. While the algorithm achieves high frame rates, the speedup for 16 processors is only about 12 (an efficiency of 75%). There are two potential sources of overhead leading to the non-linear speedup: memory stall time caused by cache misses, and processor idle time caused by load imbalances.

Figure 6.7 illustrates how the task size can be adjusted to minimize the total overhead by balancing the tradeoff between memory stall time and processor idle time. The figure

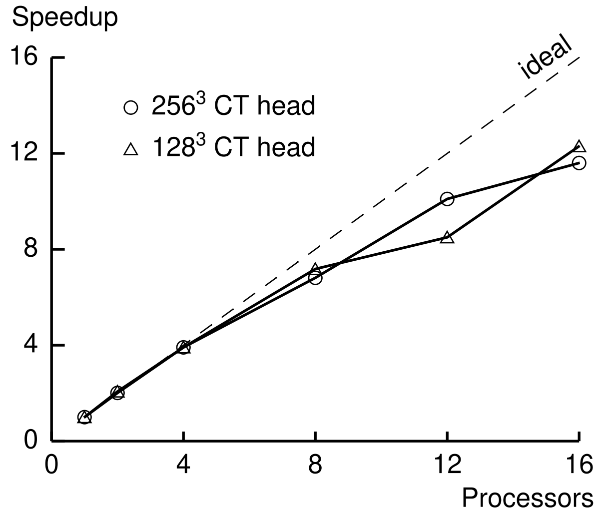


Figure 6.6: Speedup curve on the Challenge multiprocessor.

shows a breakdown of the overheads for a fixed number of processors (16) as the task size is varied. The total length of each bar represents the sum of the time spent by all of the processors to render a single frame (wall clock time would be found by dividing the length of the bar by the number of processors). Each bar is broken down into sections representing computation time, memory stall time, and idle time (synchronization overhead). Idle time was measured using high-resolution interval timers at each barrier, computation time was measured using a cycle-counting profiler (Pixie [Smith 1991]), and memory stalls account for the remaining time. The figure shows that small tasks result in larger memory overhead while large tasks result in poor load balancing.

The optimal task size depends on the data set and the number of processors (see Table 6.2). Figure 6.8 shows the magnitudes of the overheads for the 256^3 CT head data set with the optimal task size on a range of machine sizes. There are two observations to make from the figure. First, with optimal task sizes the memory overhead for the 16 processor run is only 9% more than the overhead for the uniprocessor algorithm. This result indicates that data redistribution and communication costs do not dominate rendering time.

The second observation is that the overhead due to idle time does increase. But from Figure 6.7 we know that the load balancing algorithm is not the problem since the idle time can be reduced to a negligible amount by reducing the task size. In fact, the decreasing efficiency

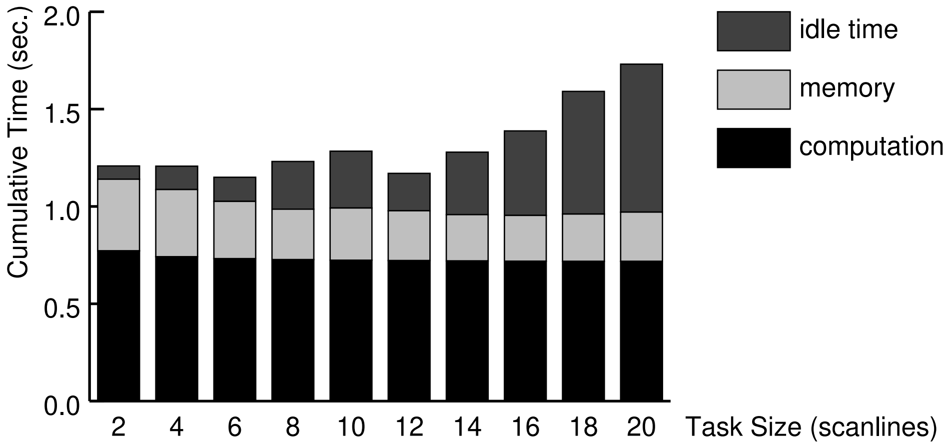


Figure 6.7: Breakdown of execution time into computation cycles, memory overhead, and idle time due to synchronization on a 16-processor Challenge as the task size varies. The time is cumulative over all processors. There is a tradeoff between the two types of overhead.

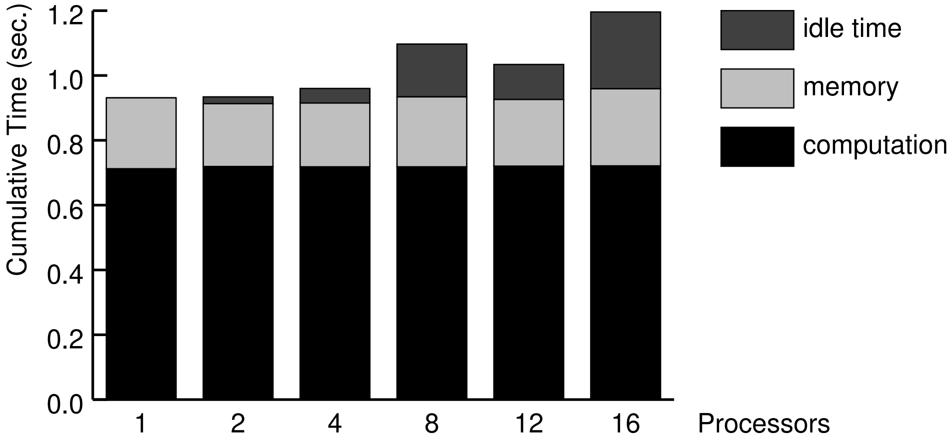


Figure 6.8: Breakdown of execution time into computation cycles, memory overhead, and idle time on the Challenge as the number of processors varies.

is due to a limit on the available parallelism in our task partitioning algorithm. As more processors are added the intermediate image must be partitioned into a larger number of tasks to achieve good load balancing, but as a result the tasks are smaller and memory overheads rise. At some point there is no net gain from making the tasks smaller. Beyond that point, since the tasks are not made smaller the overhead due to load imbalances increases as more processors are used, and eventually the speedup curve becomes flat. Therefore, cache locality requirements impose a limit on the maximum useful number of parallel tasks.

We suspect that other algorithms for volume rendering on shared-memory machines have similar limits. For example, Nieh's parallel ray caster benefits from good temporal locality since adjacent rays access data from the same cache lines [Nieh & Levoy 1992, Singh et al. 1994]. The algorithm uses an image partition, so as the number of parallel tasks increases the size of the image tiles assigned to a processor must decrease. If a large number of processors is used the likely result is a degradation in temporal locality and an increase in memory overhead.

However, our goal is to achieve interactive rendering rates, and on the Challenge our algorithm achieves that goal for moderately-large data sets. The limit on available parallelism is less of a problem for larger data sets since they can be divided into more tasks, so larger machines are useful with our algorithm provided that larger data sets are rendered. Another factor affecting the available parallelism is the architecture of the memory system, which determines the minimum practical task size. DASH, which we consider next, has a much smaller optimal task size and therefore a larger number of processors can be used. We will discuss memory performance in more detail later in this section.

6.3.3 Performance Limits on the DASH Multiprocessor

Figure 6.9 shows the speedup curve for the DASH multiprocessor. Compared to the results for the Challenge, the efficiency on DASH is much lower (60% for 16 processors, and about 50% for 32 processors). The source of the overhead, shown in Figure 6.10, is also different: memory stall time is the dominant overhead. In the rest of this section we will show that on DASH the memory overhead increases due to an increase in remote memory references,

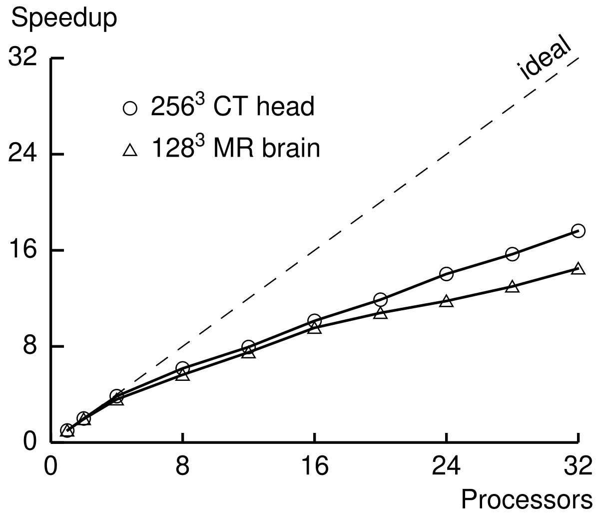


Figure 6.9: Speedup curve on the DASH multiprocessor.

indicating that the caches are not completely hiding the latency of memory. In the next section we will analyze the cache behavior in more detail and show that the performance on the Challenge is better because the caches are larger, so the working set fits, and the cache line size is longer, so the benefit of spatial locality is greater.

The first step to understanding the increased memory overhead on DASH is to recall the structure of the memory hierarchy: the machine consists of clusters of four processors connected by a bus to a local memory, and accesses to memory on remote clusters require communication via a 2D mesh network. The latency for remote accesses is roughly four times longer than for local accesses. This fact explains the jump in memory overhead from the four-processor run to the eight-processor run: the average latency for memory accesses rises abruptly. As the number of processors increases further, a larger fraction of the data structures required by a particular processor reside on remote memory and so the average latency increases even further. We used a hardware performance monitor to count the number of remote references and found that it does increase slowly: when the number of processors is doubled from 16 to 32 the number of remote memory references increases by about 10%. To prove that the memory overhead is due to an increasing fraction of remote references and not to an increasing total number of memory references, we moved all data structures to remote clusters. In this experiment the memory latency is almost constant since almost

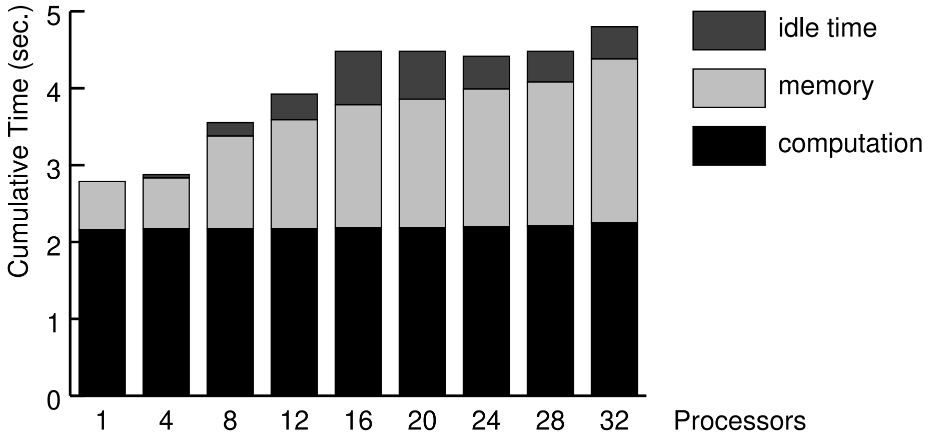


Figure 6.10: Breakdown of execution time into computation cycles, memory overhead, and idle time on DASH as the number of processors varies.

every reference is remote. As expected, we found that the memory overhead also became nearly constant (the algorithm achieved efficiencies of 85-90% for 16 processors).

Unlike our results on the Challenge, the memory overhead cannot be significantly reduced by increasing the task size. The reason is that on DASH the cache line size is much shorter (16 bytes instead of 128 bytes on the Challenge), so spatial locality is less important. The optimal task sizes shown in Table 6.3 are significantly smaller than for the Challenge. However, because the optimal task sizes are smaller, more parallelism is available and more processors can be used.

From this line of reasoning we conclude that the sub-linear speedup on DASH is caused by the caches not completely hiding the latency of remote memory accesses, and by the increasing average latency as the data structures are spread over a larger number of processors. Next, we consider the reasons for sub-optimal cache performance.

6.3.4 Memory Performance

Nieh & Levoy [1992] report that memory overhead for an optimized ray caster on shared-memory multiprocessors such as DASH is a small fraction of execution time, partly due to the effectiveness of the caches and partly due to the high ratio of computation to memory accesses. Our algorithm eliminates a considerable fraction of the computation time associated with traversing data structures, so we might expect the impact of memory overhead to be more important. To investigate further, and to explain the different results on the two machines, we used a memory system simulation tool [Martonosi et al. 1992] to qualitatively understand the memory behavior. The simulator does not model all of the details of the memory system so it cannot be used to predict miss rates on the real machines, but it does allow us to determine which data structures miss most often and the dominant causes of the misses.

With the cache parameters adjusted to roughly approximate the caches at the cluster level on DASH, the simulator indicates two dominant causes of cache misses. The first cause is communication (invalidation misses) resulting from the redistribution of the intermediate image and to a lesser extent the shading lookup table. These misses result from inherent communication in the parallel algorithm. The intermediate image is redistributed during the 2D warp since each processor is responsible for assembling rectangular pieces of the final image from pieces of the intermediate image computed by other processors. It is difficult to reduce this cost since a warp inherently requires moving pixels to different parts of the image. The communication cost slowly increases as the number of processors increases because smaller fractions of the intermediate image and the shading table are computed on the same processor as they are used.

When the cache line size is increased to 128 bytes (to match the line size on the Challenge) the simulator reports that the number of misses due to communication drops by more than a factor of five. This result indicates that our algorithm has significant spatial locality. Therefore, on architectures with long cache lines the cost of communication misses can be masked by the caches.

The second source of misses on DASH is from capacity misses, which occur when the cache is not large enough to capture the working set. The second-level caches on DASH (256 Kbytes) are small by today's standards. Increasing the cache size in the simulation to 1 Mbyte per processor (to match the second-level cache size on the Challenge) causes a significant drop in the number of capacity misses, so they should have a smaller effect on the Challenge architecture. The simulator also shows that as the number of processors increases the number of capacity misses decreases due to the increased amount of cache in the machine.

To summarize the analysis for DASH, when the increasing cost of communication and the decreasing cost of capacity misses are combined the net result is that the total number of remote memory references increases slowly as the number of processors increases. This result from the simulator matches the increase in remote references observed with the hardware performance monitor. The memory overhead increases at a slower rate than the processing power, so the speedup continues to rise as more processors are used, but the efficiency goes down. In comparison to Nieh's parallel ray caster [Nieh & Levoy 1992], our algorithm traverses the volume data in a different order that results in a reduction in temporal locality and a larger working set.

On the Challenge, the combination of larger caches (which can capture a larger working set) and longer cache lines (which capitalize on spatial locality to reduce the number of misses) result in good memory performance. The current trend in processor design is towards increasing memory latency with longer cache lines to amortize that latency over many references. Scanline-based algorithms with good spatial locality, such as the shear-warp algorithm, can effectively make use of the prefetching effect of long cache lines.

As a final experiment to determine the impact of volume data redistribution, we eliminated the cost of redistribution by replicating the 128^3 MR volume on all of the clusters of DASH. The rendering time decreased by less than 5% with two processors and by less than 1% with 12 or more processors, indicating that volume redistribution costs are not the limiting performance factor on shared-memory multiprocessors. The hardware handles communication and caching of the volume data automatically with no performance penalty. Unfortunately the caches are less effective at hiding the latency of inherent communication in the algorithm, i.e. redistribution of the intermediate image, so the memory overhead is non-negligible despite the low cost of accessing volume data.

6.3.5 Load Balancing

Table 6.4 compares the performance of the different task partitioning algorithms we implemented. The load imbalance is measured by timing the resampling and projection computation (between the first and second barriers in Figure 6.1) on each processor and then

Task Assignment Algorithm	Load Imbalance	Rendering Time (msec.)
round-robin		
stealing	7%	149
no stealing	18%	160
contiguous		
stealing	32%	187
no stealing	169%	342

Table 6.4: Comparison of load balancing algorithms for the 256^3 head on DASH with 32 processors. Round-robin task assignment with dynamic stealing performs best.

computing the average time and the maximum deviation from the average time. The numbers reported in the table are the maximum deviations as a percentage of the average time. A value of 0% indicates that all processors take the same amount of time, and a value of 100% indicates that at least one processor takes 100% longer than the average time.

The best results are obtained with a round-robin assignment and stealing (dynamic repartitioning); the load imbalance is almost negligible. When stealing is disabled the load balance increases significantly, although overall rendering time increases by only 7%. This result indicates that the initial round-robin assignment is close to optimal, but stealing is easy enough to implement on a shared-memory machine that even a small increase in performance makes it worthwhile. The contiguous partition is a sub-optimal initial partition, so stealing makes a much bigger difference. But even with stealing, the contiguous partition is worse than the round-robin partition. We theorize that because the round-robin partition reduces the number of tasks that are stolen there is somewhat more temporal locality and fewer cache misses.

6.3.6 Related Work

We have argued that an image partition coupled with hardware caching results in low memory overhead. Most volume rendering algorithms for message-passing machines use object partitions to minimize communication costs [Neumann 1993]. However, Corrie & Mackerras [1993b] propose an algorithm with an image partition that uses software caching to exploit data reference locality on a message-passing machine. The target application is non-interactive rendering of very large data sets, so the granularity of communication is large (data blocks of 8 Kbytes or larger) and the overhead of software caching is reasonable. But for moderately-sized data sets and real-time rendering rates the granularity of communication is necessarily smaller and the overhead of software-controlled caching is likely to be high. Furthermore, the method does not reduce the increase in message traffic incurred if a dynamic load balancing algorithm is used. In contrast, the hardware caches on a shared-memory multiprocessor automatically hide most of the cost of communication in our algorithm.

Our results corroborate the conclusions of Nieh & Levoy [1992] and Singh et al. [1994], both of which demonstrate the success of image partitions on shared-memory multiprocessors.

6.4 Chapter Summary

The multiprocessor shear-warp algorithm is capable of rendering 256^3 volumes at rates exceeding 10 Hz on readily-available hardware. We have achieved these speeds by using an optimized rendering algorithm with low computational costs, and an image partition which can take full advantage of the optimizations and which has lower synchronization costs than object partitions. The memory performance of the algorithm is good because we use cache-coherent shared-memory architectures. On the Challenge, which has a long cache line, the spatial locality in our task partition results in negligible memory overhead. Even on DASH, with a much shorter cache line and small caches, the memory overhead is acceptable. Shared-memory architectures work well for volume rendering because they efficiently support fine-grain communication, and have the added benefit of a simple programming model: the parallel code is virtually identical to the serial code.

One disadvantage of the algorithm described in this chapter is the preprocessing step required to run-length encode the volume whenever the user reclassifies the volume (i.e.

reassigns voxel opacities). However, the same parallelization can be used for the fast-classification algorithm, allowing real-time classification and rendering. The perspective projection algorithm can also be parallelized in the same way. Neither of these extensions has yet been implemented, but we expect a factor of 2-3x slowdown compared to the results presented here.

Another interesting area for future work is a more detailed analysis of the memory performance of volume rendering algorithms for shared-memory multiprocessors. Unfortunately, such studies are difficult because the shared-memory model encourages implicit communication through shared variables rather than explicit calls to message-passing routines. While it takes little effort to produce a correct, efficient program, it is difficult to analyze the source of performance bottlenecks deep in the memory system.

In the coming years, small-to-medium-scale shared-memory multiprocessors based on fast microprocessors are likely to become commonplace. The parallel shear-warp algorithm is well-suited to this type of architecture and enables a cost-effective solution for real-time volume rendering.