To appear in Proceedings of the 1995 Parallel Rendering Symposium.

Real-Time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp Factorization

Philippe Lacroute
Computer Systems Laboratory
Stanford University

Abstract

This paper presents a new parallel volume rendering algorithm that can render 256^3 voxel medical data sets at over 10 Hz and 128^3 voxel data sets at over 30 Hz on a 16-processor Silicon Graphics Challenge. The algorithm achieves these results by minimizing each of the three components of execution time: computation time, synchronization time, and data communication time. Computation time is low because the parallel algorithm is based on the recently-reported shear-warp serial volume rendering algorithm which is over five times faster than previous serial algorithms. Synchronization time is minimized by using dynamic load balancing and a task partition that minimizes synchronization events. Data communication costs are low because the algorithm is implemented for shared-memory multiprocessors, a class of machines with hardware support for low-latency fine-grain communication and hardware caching to hide latency.

We draw two conclusions from our implementation. First, we find that on shared-memory architectures data redistribution and communication costs do not dominate rendering time. Second, we find that cache locality requirements impose a limit on parallelism in volume rendering algorithms. Specifically, our results indicate that shared-memory machines with hundreds of processors would be useful only for rendering very large data sets.

CR Categories: D.1.3 [Concurrent Programming]: Parallel Programming; I.3.3 [Computer Graphics]: Picture/Image Generation—Display Algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism.

Additional Key Words: Volume Rendering, Scientific Visualization, Medical Imaging, Image Partition, Shear-Warp Algorithm.

1 Introduction

Real-time rendering is an important goal in visualization applications. In order to enable researchers to interactively explore their data, applications must provide smooth user interfaces with sub-second update rates. 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, allowing interactive manipulation of the data but eliminating many perceptual cues [11]. The second approach is to rely on special-purpose hardware, with the disadvantage that algorithms implemented in hardware lose flexibility [26, 2, 8, 20].

The third approach is to parallelize a brute-force volume rendering algorithm for a large multiprocessor such as a massively-parallel SIMD machine [3, 22, 24, 7, 25]. SIMD architectures perform best when communication patterns are regular and uniform, so the best speedups are achieved with simple brute-force rendering algorithms. Brute-force algorithms for large MIMD machines have also been reported [14, 5, 16]. 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 to reduce the computational costs [19, 18]. 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 most performance 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 paper.

We present 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 a new fast serial rendering algorithm [10]. The rendering algorithm exploits coherence in both the volume and the image and uses a shear-warp factorization of the viewing transformation to enable an efficient scanline-order traversal of the coherence data structures. Compared to the serial version of the octree-accelerated ray caster used by Nieh [19] the cost of the serial version of our rendering algorithm is lower by a factor of five or more. 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 [19]. 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 [18, 16, 5].

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 mul-

---

Author's Address: Center for Integrated Systems, Stanford University,
Stanford, CA 94305-4070
E-mail: lacroute@graphics.stanford.edu
World Wide Web: http://www-graphics.stanford.edu

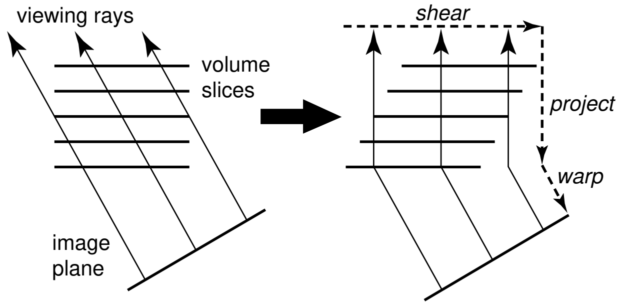

Figure 1: Instead of projecting voxels directly from the volume to the image as shown on the left, we first shear the volume by translating each slice as shown on the right. This step makes projection simple and efficient.

tiprocessor. 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 [26] and exceed the best reported results for massively-parallel SIMD machines [7, 25] and much larger MIMD message-passing machines [18]. 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 [19].

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 [19]. 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.

In Section 2 of this paper we give an overview of the serial rendering algorithm. Section 3 describes the parallel algorithm and the design tradeoffs we considered. Section 4 describes the architectures of the Challenge and DASH multiprocessors and the details of our implementation for these machines. We then present our performance results in Section 5 and discuss the limitations on parallelism. Finally we discuss related work and our conclusions.

2 Serial Algorithm

The new multiprocessor algorithm is a parallelization of the serial rendering algorithm described in [10]. The rendering algorithm is an object-order algorithm based on a factorization of the viewing transformation matrix into a 3D shear parallel to the slices of the volume, a projection to form a distorted intermediate image, and a 2D warp to form an undistorted final image.

Figure 1 illustrates the factorization for a parallel projection (although the method extends to perspective). The horizontal lines in the figure represent slices of volume data sampled on a rectangular grid viewed in cross-section. We choose the shear transformation such that the viewing rays become perpendicular to the slices of the volume. The shear is implemented by translating and resampling each slice of volume data. Projection is then trivial: the resampled slices are composited together in front-to-back order using the "over" operator to form an intermediate image. Finally, the intermediate image must be transformed into the correct final image by applying an affine 2D warp. The warp is relatively inexpensive because it operates on 2D images that are much smaller than the 3D volume data.

The advantage of projecting volume slices to the intermediate image instead of directly to the final image is that the scanlines of volume data are always aligned with the scanlines of the intermediate image. It is therefore possible to stream through both the volume data and the intermediate image simultaneously in scanline order. No scaling or rotation is necessary, resulting in simple addressing arithmetic and simple resampling.

To take advantage of coherence we use a run-length encoding of the volume data to skip over transparent voxels [17] and a run-length encoding of the intermediate image to skip over pixels with high accumulated opacity (since the corresponding voxels must be occluded) [21]. This second optimization is equivalent to early ray termination in a ray caster [13]. The run-length encoded volume must be precomputed, and three copies are required (one for each of the principal viewing axes). The run-length encoded intermediate image is computed on-the-fly.

3 Parallel 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. We begin by focusing on the projection of the volume data into the intermediate image since that dominates the cost of the serial algorithm, and then we consider data distribution and the complete parallel algorithm.

3.1 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 to resample and composite [18, 14]. The partial results from each processor must then be composited together to form the image. In contrast, in an image partition each processor's task is to compute a specific portion of the image [19, 5]. Each image pixel is computed by only one processor, but the volume data must be moved to different processors as the viewing transformation changes.

Neumann [18] 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 [19] 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 [13], 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 as shown by Challinger [4]. The binary-swap image compositing algorithm proposed by Ma [14] 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 to use an image partition, which requires no synchronization until the entire volume has been composited.

3.2 Task Shape

There are several choices for the unit of work in an image partition: individual pixels, scanlines of pixels [4, 26], or rectangular tiles [19]. Generally, tile-shaped tasks are best in order to maximize temporal cache locality [19, 5]. However, two factors unique to the run-length encoded data structures in this 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. 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 5.

3.3 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 make it difficult to predict performance.

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 [19]. 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. 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.

3.4 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 hotspots. To avoid hotspotting we use a round-robin distribution of all large data structures to the different physical memories, and 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.

3.5 Overall Algorithm

The following pseudo-code shows the flow of control for each processor in the parallel algorithm. First, each processor computes 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 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.

```
procedure ParallelRender()
    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
```

4 Implementation

4.1 Hardware Architectures

We have implemented the algorithm for two architectures, the Silicon Graphics Challenge and the Stanford DASH Multiprocessor.

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 Gbits/sec.	960 Mbits/sec.

Table 1: Characteristics of the parallel architectures used for the performance measurements. Both machines are shared-memory multiprocessors with hardware cache-coherence.

Table 1 lists the characteristics of the machines.

The 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. 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 [12].

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.

4.2 Software Implementation

We implemented the algorithm using the C language and the ANL parallel programming macro package [1]. Because of the shared-memory programming model the parallel code is almost identical to the original serial code. 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 hotspotting, 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 grayscale 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, but the transparent voxels are not stored so the total size is usually smaller than the original volume.

We shade the voxels using the Phong illumination model implemented with an 8192-entry lookup table [9]. 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.

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 2: Rendering rates and tasks sizes on the SGI Challenge.

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)	0.3	0.7	1.3	2.1	3.4	5.9
Task Size (scanlines)	32	20	12	12	8	3

Table 3: Rendering rates and task sizes on the Stanford DASH Multiprocessor.

5 Results

We tested our algorithm using two data sets: a 256x256x225 voxel CT scan of a human head (Figure 2) and a 128x128x109 voxel MR scan of a head (Figure 3, top left). Both data sets are from the Chapel Hill Volume Rendering Test Dataset. Medical data sets such as these have significant coherence: with the classification functions we used to produce the images in the figures, 5% of the voxels in the CT data set are non-transparent and 11% of the voxels in the MR data set are non-transparent. We used an early-ray-termination opacity cutoff of 95%. The images are grayscale and contain 256x256 pixels. Note however that the rendering times are relatively independent of the image size which only affects the cost of the 2D warp. The total amount of physical memory required for the three copies of the classified volume and the image data structures is 4 Mbytes for the MR data set and 12 Mbytes for the CT data set.

5.1 Rendering Rates

Tables 2 and 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 and GL) 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 each frame. The fastest rates are achieved with the Challenge: 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 [19] on DASH, 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. In contrast to brute-force approaches, the optimizations in the algorithm are data-dependent and do not work well for data sets that are both dense and entirely semi-transparent. However, in our experience such data sets generally produce uninformative visualizations. Proper choice of the classification function results in both increased visual clarity and a high degree of coherence, so coherence optimizations are useful in practice.

Figure 3 includes three color renderings produced using the multiple-material shading model described in [6]. Renderings take about twice as long with this shading model due to the additional

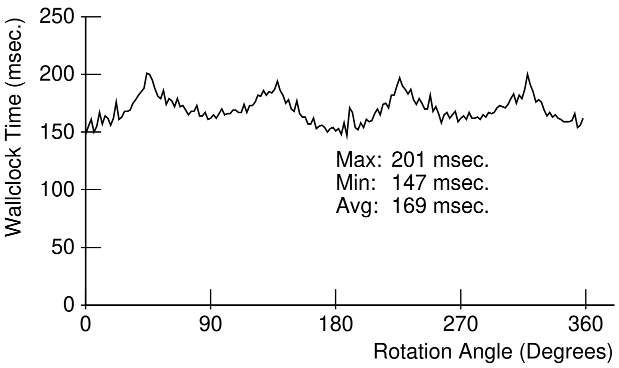

Figure 4: Rendering time for the 256^3 grayscale head on DASH with 32 processors as the viewing angle changes.

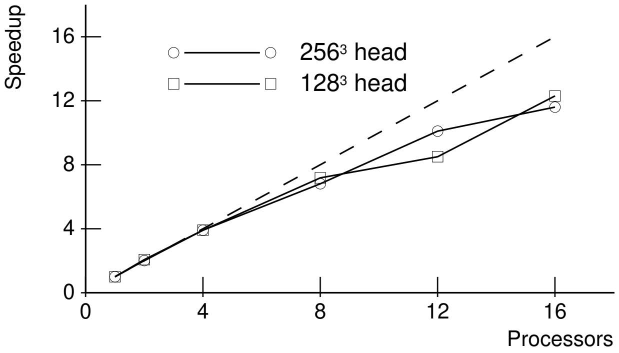

Figure 5: Speedup curve for the Challenge multiprocessor.

resampling for the two extra color channels, and additional table lookups to perform the shading. On DASH with 32 processors, the color CT head (256x256x225 voxels) classified with semitransparent skin was rendered at a rate of 2.0 Hz. Compared to the grayscale rendering of the same data set, 30% of the time increase is due to the classification and 70% is due to the shading model. The CT engine (256x256x110 voxels) was rendered at a rate of 2.8 Hz, and the CT abdomen (256x256x159 voxels) was rendered at a rate of 2.4 Hz.

Figure 4 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.

5.2 Performance Limits on the Challenge

Figure 5 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 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 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 (wallclock

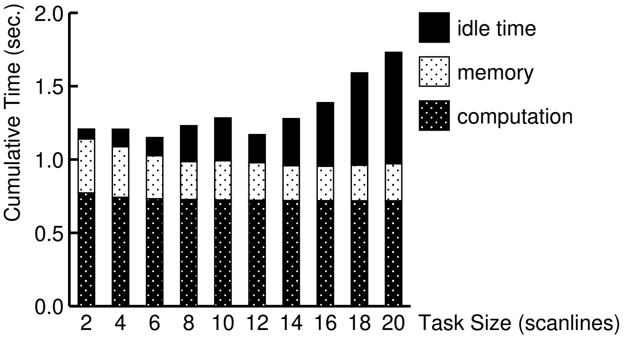

Figure 6: Breakdown of execution time into computation cycles, memory overhead, and idle time due to synchronization on a 16-processor Challenge as the task size varies. The time is cumulative over all processors. There is a tradeoff between the two types of overhead.