Improving Parallel Shear-Warp Volume Rendering on Shared Address Space Multiprocessors

Dongming Jiang and Jaswinder Pal Singh

Department of Computer Science
35 Olden Street
Princeton University
Princeton, NJ 08544

{dj, jps}@cs.princeton.edu

Abstract

This paper presents a new parallel volume rendering algorithm and implementation, based on shear warp factorization, for shared address space multiprocessors. Starting from an existing parallel shear-warp renderer, we use increasingly detailed performance measurements on real machines and simulators to understand performance bottlenecks. This leads us to a new parallel implementation that substantially outperforms and out-scales the old one on a range of shared address space platforms, from bus-based centralized memory machine to hardware-coherent distributed memory machines to networks of computers connected by page-based shared virtual memory. The results demonstrate that real time volume rendering is promising on general purpose multiprocessors, and illustrate the utility of tool hierarchies in conjunction with algorithmic and application knowledge to understand memory system interactions and improve parallel algorithms.

1 Introduction

Many computer graphics applications are important candidates for multiprocessing, because they desire real time rendering and animation which is difficult to achieve on uniprocessors. However, graphics computations tend to have highly irregular and unpredictable patterns of data access, communication and synchronization. This makes them challenging for both parallel programming and performance, as well as useful case studies in parallel algorithms and programming. Volume rendering is one such application.

Volume rendering is an important tool in the graphical visualization of three-dimensional (3D) data. In many applications, a sequence of frames from different viewpoints to the volume is rendered, the goal being to render them in real time (30 frames per second) or interactive time (10 or 15 frames per second). While much research has been done in fast rendering algorithms, it is still not possible to render data sets of interesting size at interactive rates on serial processors, and the costs grow quickly as data set sizes increase. For example, an optimized ray-casting renderer was measured to take about 5 seconds to render a single frame for a 256^3 data set on a 150MHz SGI Indigo2 R4400 workstation, and a recent shear-warp algorithm takes 0.8 seconds to render the same frame on the same platform. The computational complexity of both scales linearly with the size of the volume data set.

The widespread availability of small to moderate scale multiprocessors makes them viable platforms for speeding up volume rendering. Successful parallel implementations of ray casting renderers have been developed for both centralized and distributed shared memory multiprocessors [8]. Parallel implementations of the faster shear-warp method have also been developed [4, 12], but these have not scaled well beyond 8 to 12 processor systems, or performed well on systems with physically distributed memory. We first attempt to understand why, studying the performance, scalability and characteristics of the existing parallel shear-warp renderer. We do this through detailed measurement using performance tools and instrumentation on real systems, as well as through simulation.

Based on the understanding obtained, we develop an improved partitioning method that has much better communication and data locality properties for both centralized and distributed shared memory machines, while still preserving load balance. We show that this new method performs substantially better on five very different types of shared address space platforms: a Silicon Graphics Challenge (a centralized shared memory machine), a Stanford DASH (a distributed shared memory machine), a simulated next-generation distributed shared memory machine, a Silicon Graphics Origin2000 (a modern, scalable distributed shared memory machine) and a system that supports a coherent shared address space at page granularity using shared virtual memory. The magnitude of the improvement increases as we move along these types of systems, i.e. as more locality issues arise and communication becomes relatively more expensive. We present the detailed memory system interactions and locality behavior of the different schemes, so that performance can be better understood and extrapolated to other systems.

The use of performance debugging tools and simulators was an interesting aspect of our experience. Going into successively deeper details of locality and performance characteristics ended up pointing us to a much higher level algorithmic restructuring to obtain better performance. Coarse understanding of whether data access or load imbalance is the major bottleneck were possible to glean on the real systems we used, but understanding the locality properties we needed required either more detailed tools or simulation. All levels of the "tool hierarchy" were useful, but the ultimate solution took deeper algorithmic knowledge. Tools on real machines that can monitor memory interactions and track detailed interaction sources would have been very valuable, and we hope that experiences such as this will guide us to the design of tool hierarchies for real systems.

We begin by briefly describing the sequential shear-warp algorithm. In section 3, we present detailed measurements of the performance, scalability and memory system interactions of the existing parallel method. The new parallel partitioning technique is presented in section 4, and its performance and system interactions are illustrated in section 5. Finally, in section 6 we summarize and conclude the paper.

2 Serial Volume Rendering Algorithms

Volume rendering algorithms consist of three basic steps: assigning a color and an opacity to each sample in a 3-dimensional input array, projecting the sample onto an image plane, and then blending the projected samples. The algorithms can be divided into two types, image-order and object-order, depending on whether the outer loop in rendering a frame is over image pixels or object voxels (volume elements). Volume rendering by ray casting [8] is an image-order algorithm: the outer loop iterates over the pixels in the two-dimensional image, and for each pixel fires a ray (or set of rays) into the volume. The ray is sampled as it proceeds through the volume, and a color and opacity is computed for the pixel. On the other hand, the shear-warp algorithm is an object-order algorithm containing two phases: a compositing phase and a warp phase. The right side hand of Figure 1 illustrates the two phases in a parallel projection. The viewing transformation is factored into a 3D shear space parallel to the volume slices. The compositing phase first streams through the 3-D volume data and projects the volume to form a distorted intermediate (composited) image. The composited intermediate image is then transformed into a final undistorted image by a 2D warp during the warp phase. The sizes of the intermediate and final images usually differ a lot due to the shear in the compositing phase. The compositing phase is $O(n^3)$ for an n-by-n-by-n voxel volume, and is the dominant phase in sequential execution. A detailed description of this algorithm can be found in [4].

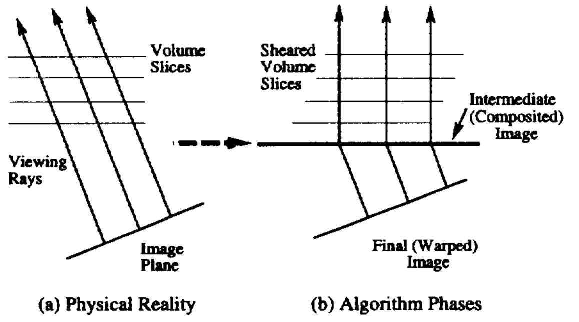


Figure 1: The serial shear-warp volume rendering algorithm.

The shear-warp algorithm is reported to be the fastest volume rendering algorithm so far that does not compromise quality on a serial workstation, almost 4-7 times faster than a functionally "equivalent" efficient ray-casting algorithm. It combines the linear volume traversal advantage of object-order algorithms with the optimizations traditionally performed by image-order algorithms. Let us see how.

For many types of volumes, particularly medical images, 70% to 95% of the voxels are found to be transparent. It is therefore useful to avoid rendering these transparent voxels. So called coherence-accelerated schemes employ spatial data structures for this purpose, which encode the presence or absence of high-opacity voxels in the volume. One such a data structure is the run-length encoding that the shear-warp algorithm uses for both the volume and image data [4]. (Ray casting algorithms use an octree representation of the volume for this purpose, so interesting regions of the volume can be easily found).

Another optimization is to omit those regions of the volume that are not visible to the viewer because the rays that reach them are already saturated with opacity. This is called early ray termination (for example, ray casting algorithms can terminate the traversal of a ray through the volume when it has accumulated too much opacity). In the shear-warp algorithm, 2-D slices through the volume, parallel to the intermediate image, are processed in front-to-back order (see Figure 1). To facilitate early ray termination, if the opacity of a pixel in the intermediate image exceeds a threshold, the pixel is marked as opaque and will be skipped when processing the rest of the slices.

Given a data set, shear-warp and ray-casting algorithms use an almost identical number of actual compositing and resampling operations. However, two data accessing properties make shear-warp outperform ray casting on a uniprocessor. First, in a ray caster the looping time—which is the time spent on control overhead and traversing coherence data structures while searching for the next voxel to process—dominates the total rendering time: It traverses an octree data structure that represents the space once for each ray, and has to address each voxel along that ray. In contrast, the shear-warper traverses the run-length coherence data structures in scanline order for both the volume and the intermediate image when compositing the latter. The data structures are traversed only once, and the linear traversal reduces addressing overhead. In addition, in a ray caster the traversal order through voxels is not the same as their storage order in memory, so spatial locality is poor. The shear-warper, however, traverses the data in the same order as the storage order, so exploits spatial locality well on both the object and image. Figure 2 demonstrates these differences [4].

3 Analysis of Existing Parallel Algorithms

Parallel ray casting volume renderers have been shown to perform well on cache-coherent systems [8]. Their spatial and temporal locality properties have also been analyzed [3, 13]. While spatial locality on volume data is poor, as discussed above, temporal locality across rays is high. More recently, parallel versions of the shear-warp algorithm have also been implemented on similar systems [5, 12]. The parallel shear warper achieves interactive frame rates on small-scale, centralized shared-memory multiprocessors for moderate sized data sets (256^3 voxels). However, its

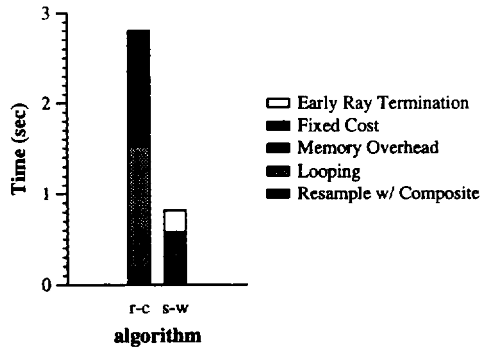


Figure 2: Breakdown of rendering time for the ray caster (r-c) and the shear warper (s-w) on an SGI Indigo2 R4400 workstation for a 256 x 256 x 167 MRI brain.

scalability with increasing number of processors is not very good, and nor is its performance with physically distributed rather than centralized memory. These issues limit parallel performance and are not well understood.

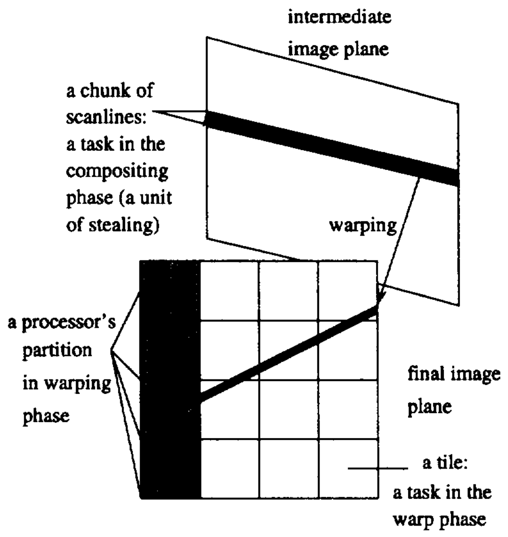
In the rest of this section, we analyze the parallel shear warper of [5, 12] in detail on cache-coherent shared address space architectures. We examine its locality properties, spatial and temporal, as well as its concurrency and load balance. We use a detailed simulator as a performance diagnosis tool to understand performance characteristics and bottlenecks, which leads us to a more efficient parallel shear warp algorithm. Let us begin by introducing the parallel algorithm and the platforms we use.

3.1 Existing Parallel Implementation

It is natural to divide the parallel implementation into two phases, compositing and warping, separated by a global barrier. In the compositing phase, we can choose to partition the object (volume data) or the intermediate image. If we partitioned the slices of the volume among processors, different processors would have to write some of the same scanlines in the intermediate image, for two reasons. First, a given scanline of the volume affects two scanlines of the image, so two processors assigned adjacent volume scanlines would write-share image scanlines. Second, due to the shear of the volume with respect to the intermediate image, it is quite likely that different processors would be assigned scanlines that are in the same plane perpendicular to the intermediate image, but in different slices. This implies not only write-sharing of the image but also that synchronization would be needed for mutual exclusion on these writes (unless we use event synchronization between slices, which is too conservative).

The alternative is to partition the scanlines of the intermediate image among processors, and have the processors read-share the volume scanlines that they need to update their assigned image scanlines. This is better since it avoids write-sharing and synchronization. To preserve the good spatial locality of the uniprocessor shear-warp algorithm, and to simplify programming in the presence of run-length encoding, the original parallel implementation chooses as a task a set or "chunk" of complete scanlines. For load balance, chunks of scanlines are initially assigned in an interleaved manner to processors, and task stealing is used when a processor becomes idle while other processors still have work to do. The size of each task, i.e. the number of scanlines in a chunk, is a combination between spatial locality and load imbalance, and is determined empirically for a given data set, number of processors, and platform.

In the warp phase too, we would like only one process to write a given pixel, for similar reasons, so the algorithm partitions the final image rather than the intermediate image among processors (here too, each pixel of the intermediate image affects more than one pixel of the final image). Due to the change in orientation during the warp (see Figure 1), at the granularity of small chunks of scanlines, there is little relationship or overlap between a chunk read and that written in the warp. Contiguity of partitions is important for locality, but making chunks larger would cause load balance to be a problem. There is little computation in the warp phase, so task stealing for load balancing may not be worthwhile either. The original parallel algorithm chooses to compromise by dividing the final image into fixed-size square tiles, and assigning tiles in a round-robin interleaved manner to processors, thus allowing load balance without task stealing. The warp is illustrated in Figure 3.



intermediate image plane
a chunk of scanlines: a task in the compositing phase (a unit of stealing)
warping
a processor's partition in warping phase
final image plane
a tile: a task in the warp phase

Figure 3: Partitions and tasks of the original parallel shear warp algorithm, assuming 4 processors.

3.2 Experimental Environment

In this subsection, we describe the three platforms we used to analyze the original parallel program. The SGI Origin2000 and the shared virtual memory system we used later will be described in section 5.5.

DASH The Stanford DASH prototype [2] we use is a 32-processor, distributed shared-memory system with a 2-D mesh network. Each node in the mesh is a four-processor bus-based multiprocessor, with four, now-dated 33 Mhz R3000 processors. Each processor has separate 64KB first-level instruction and data caches, and a unified 256KB second-level cache. The cache line size is small, 16-bytes, even in the second-level cache, and the maximum node-to-network communication bandwidth is 120MB/sec. Processor caches within a node are kept coherent by a snoopy bus protocol, and across nodes by a distributed directory protocol.

Challenge The SGI Challenge is a bus-based symmetric shared-memory multiprocessor. The machine we use has sixteen 150 Mhz processors, each with separate 16KB first-level instruction and data caches and a unified 1MB second-level cache. The second-level cache line size is 128 bytes, and the bus bandwidth is 1.2GB/sec.

Simulator This simulates a more "pure" and modern cache-coherent shared address space multiprocessor, with physically distributed memory and one processor per node. Every processor has a single-level cache, and caches are kept coherent by a distributed directory protocol similar to that of DASH. To represent a modern high-performance multiprocessor, we set the uncontended cost of a locally satisfied cache miss to 70 cycles, and that of a remotely satisfied miss to 210 or 280 cycles, depending on whether the miss led to two protocol hops or three [2]. The default settings are 1Mbyte cache size, 64 byte line size, 4-way set associativity. Peak node-to-network communication bandwidth is 400MB/sec. The simulator interfaces with the Tango-Lite reference generator [6], and models buffering and contention in detail everywhere except in the network links and routers.

3.3 Input Data Sets

The primary volume data used as inputs to analyze the algorithms are a set of MRI scans of a human brain with different resolutions: 128 × 128 × 128 voxels, 256 × 256 × 167 voxels, and 511 × 511 × 333 voxels. The so-called 128^3 and 256^3 sets are common in volume rendering today. The 512^3 data set is larger, but likely to be common in the near future.

In order to explore different algorithms more broadly, we also used a higher resolution of the above data set at 640 × 640 × 417 voxels, as well as a set of CT scans of a human head at resolutions of 128^3 voxels, 256^3 voxels and 511^3 voxels as supplementary inputs. To generate the 512^3 and 640^3 data sets, we used a resampling tool to up-sample the 256^3 raw data along each dimension.

3.4 Performance Analysis

The parallel efficiency of the renderer is strongly dependent on task size. We therefore empirically determine and use the optimal task size for each configuration in our measurements.

3.4.1 Speedups

While the shear warper is very fastest sequentially, it does not obtain nearly as good self-relative speedup on multiprocessors as a ray caster [12]. The speedups for the parallel shear warper on the 512^3 data set are shown in Figure 4 for the different platforms.

To understand why speedups fall off with increasing processor count, we used performance tools to obtain coarse breakdowns of execution time. On real machines, we used the Pixie program to profile execution time by basic-block counting. This gives us, on a per-processor basis, the number of cycles that would have been spent executing instructions had there been no cache misses. The per-processor

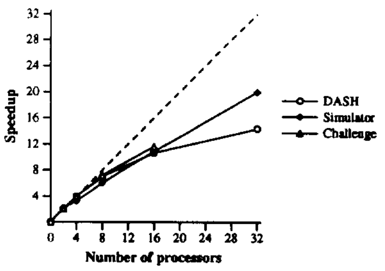


Figure 4: Speedups of the parallel shear warper with the 511 × 511 × 333 voxel data set on different platforms.

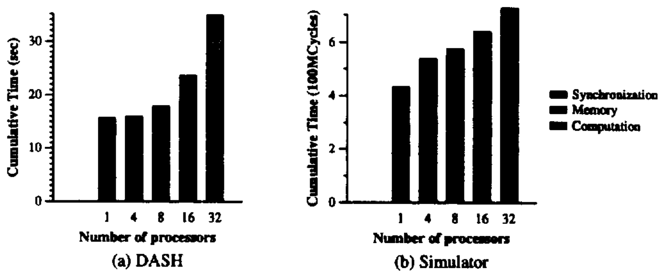


Figure 5: Breakdown of the cumulative rendering time of the parallel shear warper with the 511 × 511 × 333 voxel data set on distributed shared memory architectures.

waiting time at synchronization events was measured separately by executing the program with manually inserted timing calls before and after each synchronization event. From these and the execution time of the parallel program (without Pixie or synchronization timing instrumentation), the time spent stalled on the memory system can be easily computed. Figure 5 shows the results for the 512^3 data set. It is clearly the time spent stalled on the memory system that dominates the decline in parallel performance. About 50% of the execution time is spent in the memory system on DASH with 32 processors, compared with 18% on a single processor. The memory stall fraction is a little smaller on the other systems, but is still the major reason for performance decline in parallel execution.

To investigate the memory system overhead further, we examine how the execution time breakdown changes with data set size. Figure 6 shows the speedup curves for the three MRI data sets on different machines. Speedups on the distributed-memory DASH are in all cases considerably smaller than on the centralized-memory Challenge. Moreover, the speedup on DASH is best for the intermediate 256^3 data set and degrades for the smaller and larger ones. Concurrency grows with data set size and task stealing is used, so the reason for this ought not to be load imbalance. The problem is likely to be memory system interactions caused by communication and lack of data locality. Here, the performance debugging tools on the real systems we use run out of steam. With some manual effort, they are able to tell us that memory overhead grows with processors in both the compositing and warp phases. However, they are not able to tell us exactly where or why, or even whether the overhead is due to high miss rates (local or remote), or contention (large miss costs on some misses). We cannot determine whether the misses are due to inherent (or false) sharing of data or due to capacity, conflict or cold misses, or even whether it

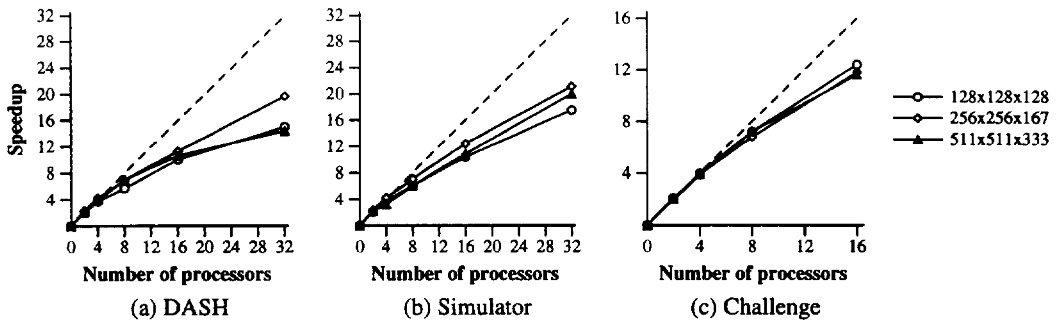


Figure 6: Speedups of parallel shear warper for different data sets on different architectures.

is spatial or temporal locality that is causing problems [1] To understand the memory system interactions, our next level of exploration is to study the communication and locality behavior through simulation.

3.4.2 Cache Miss Breakdown

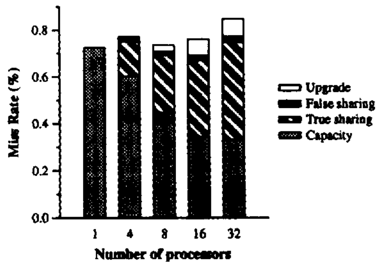


Figure 7: Breakdown of the memory overhead vs. number of processors of the parallel shear warper with the 511 × 511 × 333 voxel data set on the simulator.

Our first step is to see how the miss rate changes with the number of processors and problem size, and particularly what types of misses are substantially affected. Omitting cold misses, Figure 7 shows the detailed breakdown of the types of cache misses suffered for the 512^3 data set on the simulator, using the memory system configuration described in section 3.2 and the classification of misses exactly as described in [13].

Most of the cache misses are due to replacements and true sharing of data. (A true sharing miss is one that occurs because the word being referenced has been written by another processor since it was last referenced.) As the number of processors increases, true sharing misses dominate, indicating a rapid increase in inherent interprocessor communication. Capacity misses decrease, due to larger aggregate cache space for the same data set. Interestingly, the net result is that the overall miss rate does not increase very quickly with the number of processors. However, the simulator shows that a much larger fraction of the misses are remote (not satisfied locally), and the misses undergo significantly greater contention.

The compositing phase itself should not have much true communication, since the volume data are read-only and the image data are not actually shared. After more detailed instrumentation and examination of the algorithm, obtaining the above information for different phases, it turns out the major source of inherent communication is at the interface between the compositing and warp phases.

Even with tiles as tasks, a processor in the warp phase is likely to not use the intermediate image scanlines that it wrote and hence brought into its cache during the compositing phase, but rather to read data that other processors wrote in their caches (see Figure 3), leading to true sharing. The mismatch and communication grows as the number of processors increases and as tasks are made smaller, and is more expensive relative to processor speed and local access on machines with distributed memory. Also, while the capacity miss rate decreases, more of the capacity misses are satisfied remotely on these machines as the number of processors increases, increasing their cost (owing to the unpredictability of the viewing position, and the fact that it changes across frames, it is very difficult to place data appropriately in local memories, so pages of data are initially distributed round-robin across memories). This helps explain all our results so far, except for two: the speedup being best with the intermediate, 256^3 data set on DASH, and the speedups on the simulator being better than on DASH despite the faster processors modeled by the simulator. To understand these, let us look at the application's spatial and temporal locality properties.

3.4.3 Spatial Locality

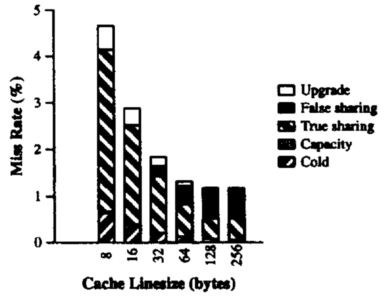


Figure 8: Breakdown of the memory overhead vs. the size of the cache line of the parallel shear warper with the 511 × 511 × 333 voxel data set on the simulator with 32 processors.

Figure 8 shows that the parallel shear-warp program retains the good spatial locality of the sequential program. The

[1]More recent processor/cache systems provide a few performance counters that can count the number of misses during the execution on a real machine (e.g. SGI Origin2000), as we shall discuss in section 5.5.1.

true sharing miss rate as well as the cold and capacity miss rates drop quickly with increasing line size, at least up to 256 bytes, and false sharing does not become a particularly major issue. (A false sharing miss is one that occurs because a different word that happens to fall on the same cache line as the word being referenced has been written by another processor since this word was last referenced.) The cache line size on DASH is much smaller than on the Challenge or the simulated machine, and this is one reason why DASH experiences a much higher miss rate than those systems in both uniprocessor and multiprocessor executions. The combination of small cache line size and distributed memory (non-local misses) makes the memory overhead higher on DASH than on the Challenge, particularly as the number of processors grows. While the simulated machine also has distributed memory, its longer cache lines and hence lower miss rates help it compared to DASH in this regard. What remains is to understand why the intermediate data set performs best on DASH. This is probably due to a tradeoff between capacity and communication, so we examine temporal locality and working sets of the program.

3.4.4 Temporal Locality and Working sets

We measure the size and scaling of the algorithm's working sets, and compare them with those of the ray caster. For a fixed problem size and number of processors, we measure working sets by running the parallel program on the simulator with different per-processor cache sizes, varying the cache sizes in powers of two between 1KB and 1MB. The cache sizes at which knees occur in the miss rate vs. cache size curve represent well-defined working sets, fitting which in the cache can make a big difference to performance [3].

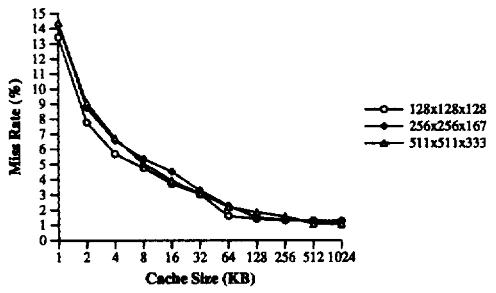


Figure 9: Miss rate vs cache size for different data sets on the simulator with 32 processors.

Figure 9 shows the results for different data sets. The cache organization assumed in this experiment is a 64-byte line size and 4-way associativity. This approaches the inherent working sets of the algorithm (best measured with fully-associative caches and a one-word line size) but is still realistic [13]. The results show that the size of the important working set of the parallel shear warp program is quite independent of the number of processors used, but grows with the size of the data set. Cache with smaller associativity, as on the machines we use, would have to be significantly larger to hold the working set than the sizes shown in Figure 9.

While the shear warper's working set is much larger than a ray caster's for the same volume [12], the total storage it uses for a given data set is much smaller since the data set is run-length encoded and greatly compressed. The larger working set is thus due to the algorithm itself. The parallel shear warper streams through the run-length encoded volume (or the portion of it that a processor touches) once in a frame, and therefore does not exploit much macro-scale temporal locality like the ray caster does across consecutive rays. The working set turns out to be roughly proportional to the size of a plane through the volume data perpendicular to the intermediate image, since there is some reuse of neighboring voxels across such plane when moving from compositing one scanline of the intermediate image to the next. It is thus proportional to n^2, where n is the number of voxels along one edge of the volume. The fact that the intermediate image data a processor writes in the compositing phase are mostly not read by it in the warp phase causes a loss of reuse and data locality as well as increasing communication. In contrast, the size of the important working set in a ray caster depends on the height of the octree that represents the volume and the length of a ray, so it grows proportionally to $\log n$ with a larger constant and proportionally to n with a smaller constant, a much smaller growth rate overall.

This explains why the parallel shear warper speeds up best for the intermediate data set size (256^3) on DASH. Inherent communication and load imbalance drop with increasing data set size, but capacity misses increase, causing artifactual inter-node communication to increase. On the centralized-memory Challenge, the difference in cost between communication and capacity (including local) misses is very small, so the differences across data set sizes are smaller as well.

4 A New Parallel Shear-warp Algorithm

Given this understanding, our major goals are to reduce true-sharing communication at the interface of the compositing and warp phases, and to try to reduce capacity misses, yet not to compromise load balance. We now describe a different partitioning method for the shear-warp algorithm that does this.

4.1 Approach

We would like that a processor in the warp phase read the same intermediate image scanlines that it wrote in the compositing phase, one way to do this by partitioning the intermediate rather than the final image in the warp phase as well, in exactly the same way as done in the compositing phase, and having processors write-share the relevant portions of the final image. While this reduces intermediate image communication, it would cause substantial problems with write-sharing and synchronization on the final image at the borders between processors that own adjacent chunks of the intermediate image. This is especially true with the current partitioning of the intermediate image into small interleaved chunks of scanlines. So this is not a good solution. Furthermore, while the current interleaved chunk assignment reduces initial load imbalance in the compositing phase, it increases both read-sharing of volume data as well as false write-sharing on the intermediate image in that phase itself. With long cache lines and high compression of data through run-length encoding, there is potential for false sharing at every boundary between scanline chunks that are assigned to different processors.

Ideally, we would like a partitioning scheme in which processors are assigned not interleaved chunks of scanlines but rather entirely contiguous partitions of the intermediate image scanlines in the compositing phase, and then they use the same partitions in the warp phase, i.e. a processor reads its assigned intermediate image scanlines and writes them to the appropriate portions of the final image. This has many advantages in the compositing phase, both read-sharing of the volume and write false-sharing of the intermediate image are minimized since a processor's assigned portion of the intermediate image now has only two borders with other processors, rather than two per chunk it is assigned. In the warp phase, a processor reads the same lines of the intermediate image that it wrote, increasing reuse and reducing communication. Spatial locality is enhanced since contiguous partitions are bigger. And write-sharing and synchronization on the final image are also small, since the warp transposes a contiguous portion of the intermediate image to a still contiguous portion of the final image: With the bilinear interpolation used in the algorithm, the only write sharing on the final image is at the scanline-wide boundaries between the contiguous partitions that processors end up writing, i.e. the scanlines that adjacent processor partitions share, instead of at the boundary of every small chunk. This synchronization and write-sharing can now in fact be eliminated, as we shall see.

The problem, of course, is obtaining a contiguous assignment of intermediate image scanlines that is load balanced or nearly load balanced. We could start from any contiguous assignment and then steal small chunks of scanlines, but stealing will violate all the nice properties above so it is important that we minimize the need for stealing. Instead, we appeal to application-level insights to develop a nearly load balanced, contiguous initial assignment, and then augment it with task stealing. We focus on balancing the compositing phase, since it is by far the more time-consuming (particularly when the communication problem between phases is solved). It turns out that a load balanced assignment for the compositing phase is good enough for the warp phase as well, in that even without stealing, the resulting load imbalance in warp does not hurt overall performance much.

Our approach takes advantage of the observation that most often volume rendering is done as an animation; that is, by rendering many successive frames from different viewpoints along a progression. The angle between successive viewpoints is typically small, to give the illusion of continuity. This means that the relative work associated with an intermediate image scanline in one frame is a good predictor of the relative work associated with it in the next frame. We can therefore dynamically profile the work done for each scanline during the compositing phase for the current frame, and use the profiles to construct contiguous, predictively balanced initial assignments for the next frame. In the rest of this sections we will describe how we do the profiling, how we compute the initial assignment for a frame based on the profiles, how we steal, and finally how we manage parallelism in the warp phase.

4.2 Profiling the Computation

It is important that the profiling be very inexpensive compared to the computation itself; otherwise, the overhead of profiling might outweigh any benefits obtained from better load balancing.

We insert profiling code in the application to count the instructions executed for compositing each scanline. In particular, we insert profiling instructions at every statically identified basic block. As a first optimization, the profiles show that several scanlines at the top and bottom of the intermediate image plane are almost always not worth processing since the portion of the volume that overlaps with them is empty (see Figure 10). While the existing parallel shear-warper blindly composites the intermediate image from the very beginning to the end, we first determine the region of the written intermediate image plane to composite, eliminating the unnecessary computation and avoiding profiling overhead for the empty portions.

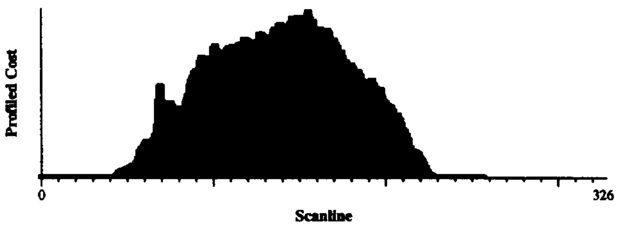


Figure 10: Profiling for rendering a frame of the 256x256x167 voxel MRI brain. The intermediate image size is 326 × 326 in the sheared space.

Even with our optimizations in inserting profiling code, profiling adds 10% to 15% overhead to the compositing time for a frame. However, experiments show that the profiles do not change very much until the viewpoint changes quite a bit. We therefore do not profile every frame but every k frames. The choice of k is a tradeoff between the cost of profiling and the predictive accuracy of the profiles; for our data sets, we found it appropriate to choose k such that profiles are computed once every 15 degrees of rotation.

4.3 Computing a Balanced Assignment

Particularly since the costs are very nonuniform across scanlines, simply recording the relative cost of a scanline is not enough to allow processors to compute their assignments efficiently. The problem is that a processor does not know where its partition should begin until the partition for the previous processor has been determined, so computing the partitions becomes a completely serial process. (In fact doing this increases the original compositing time by 50% compared with the old shear warper.)

A profile distribution like Figure 10 can be viewed as circumscribing a curve, the area under which is equal to the cumulative profiled cost for all the scanlines. The task partitioning problem is a problem of partitioning the area under the curve. Based on this observation, we construct a cumulative profile in which the entry for a scanline contains the cumulative profile cost of all the scanlines before it (including itself), using a parallel prefix operation. The relationship between the old and new profile distributions is shown in Figure 11.

Task partitioning can then be done as follows: According to the number of processors used, the partition boundaries in the cumulative cost curve can be computed by dividing the value of the last entry in the new profile into equal parts. The partition boundaries in the intermediate image are set to the scanlines whose cumulative profile costs are the closest to the boundary values assigned to this processor. Finding these boundary scanlines can be done using a fast search algorithm into the cumulative profile array. The larger and

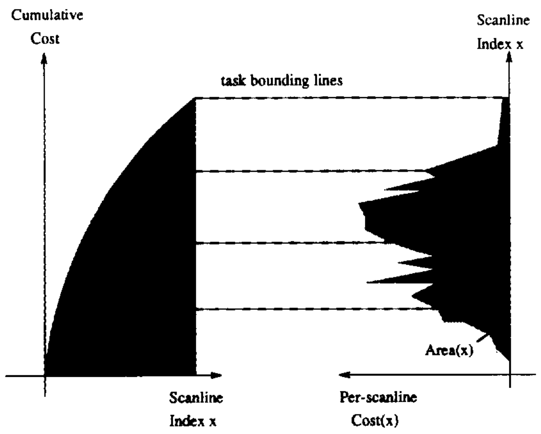


Figure 11: Partition tasks in parallel with the profiling information of previous frame. (assuming 4 processors).

more condensed the data set, the better the load balance obtained by this partitioning approach.

4.4 Dynamic Stealing

To account for possible imbalances in work load, we allow dynamic task stealing when processors become idle. Since each processor has only one contiguous partition—there are no chunks in the initial assignment—we initially set the unit of task stealing to a single scanline. However, this resulted in a synchronization overhead for task stealing about 10 times higher than that in the original parallel algorithm. We therefore steal chunks of scanlines instead of one, like in the old parallel algorithm. The difference is that the stealing chunk size has nothing to do with initial assignment; it is determined by the size of the data set, the number of processors, and the size of the cache lines.

4.5 Parallelizing the Warp Phase

As discussed earlier, the warp phase uses the same intermediate image partition as the compositing phase; each processor reads the scanlines in its partition (which it just composited) and writes the relevant pixels in the final image. Two processors may wish to write the same final image pixels only at the boundary of their partitions, so the two scanlines of the intermediate image at each boundary are assigned to one processor (the one which is assigned fewer lines) thus eliminating write-sharing and synchronization. There is no stealing in this phase, which would complicate this write-sharing solution.

5 Performance of the New Parallel Program

Results comparing the new shear warper with the old one are provided below. In addition to the data sets we used in previous sections, we ran the two parallel shear warpers on more data sets as well, to investigate the executions of the new algorithm broadly. These data sets include the 640 × 640 × 417 MRI scan voxels, and the computed tomography (CT) scans mentioned in section 3.3. We also use two new platforms in section 5.5.

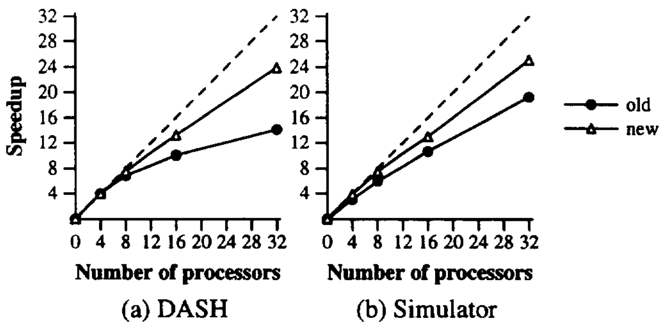


Figure 15: Speedup comparison of both the old and new parallel shear warpers with 511 × 511 × 510 voxel CT head data sets on distributed.

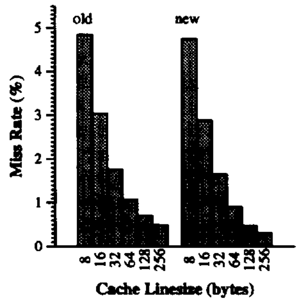


Figure 17: Comparison of the spatial locality between the two parallel shear warpers with the 511 × 511 × 333 voxel MRI brain on the simulator.

5.1 Speedups

The comparative speedup curves for the two parallel shear warpers for the MRI brain data sets are shown in Figure 12 and Figure 13 on DASH and the simulator respectively. The new algorithm's speedups are better, and especially so for larger data sets and larger number of processors. Figure 14 compares the new and the old parallel algorithms (figure (a) and (c) are actually copies from Figure 5). It shows that major difference between the old and new parallel programs is in data access stall, which no longer dominates the overhead, not only for the simulated architecture with its larger cache lines but also for DASH. At the same time, load balance is preserved in the new parallel shear warper as well.

Figures 15 shows that the results are similar to the CT human head data sets as well. Let us quickly examine the communication and locality characteristics of the new algorithm and compare with the old one.

5.2 Cache Miss Breakdown

Figure 16 shows that the new algorithm greatly decreases the sharing misses, particularly true sharing misses. This indicates that the approach achieves the goal of reducing communication between the compositing and warping phases. False sharing misses are also reduced somewhat, since the large contiguous partitions lead to fewer intermediate image borders being shared by processors.

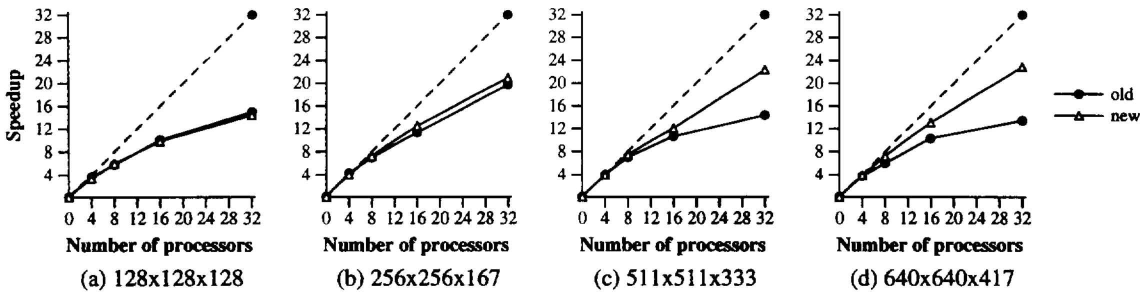


Figure 12: Speedup comparison of both the old and new parallel shear warpers with MRI brain data sets on DASH.

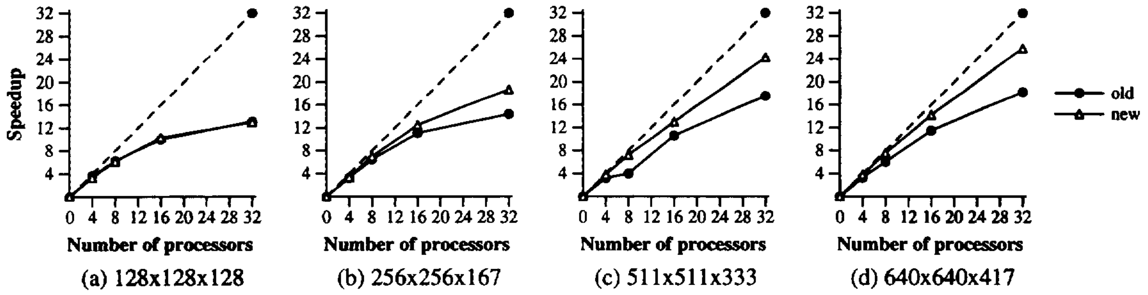


Figure 13: Speedup comparison of both the old and new parallel shear warpers with MRI brain data sets on the simulator.

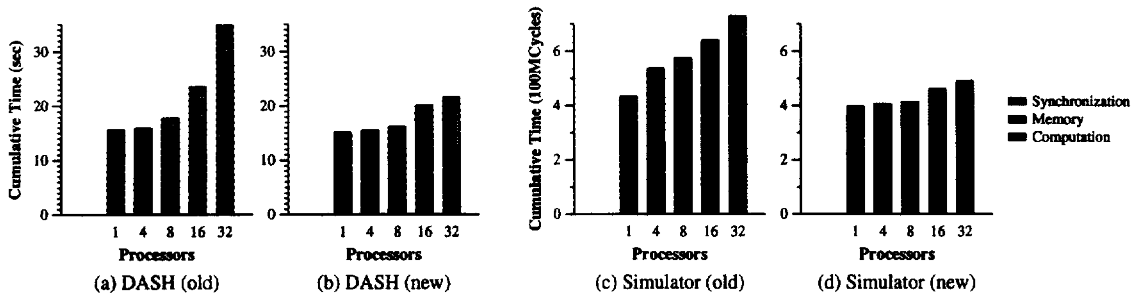


Figure 14: Breakdown of the cumulative rendering time of both the old and new parallel shear warper with the 511 × 511 × 333 voxel MRI brain on distributed shared-memory architectures. Figure (a) and (c) are for the old parallel program, and Figure (b) and (d) are for the new parallel program.

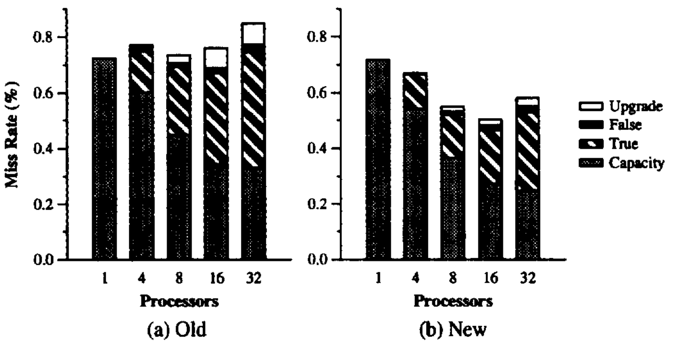


Figure 16: Breakdown of memory overhead of both the old and the new parallel shear warpers with the 511 × 511 × 333 voxel MRI brain on the simulator. Figure (a) is for the old parallel program, which is identical to Figure 5, and Figure (b) is for the new parallel program.

5.3 Spatial Locality

Figure 17 shows the new algorithm benefits even more from even longer cache lines, because a processor processes more contiguous scanlines of the intermediate image.

5.4 Temporal Locality and Working Set

Figure 14 showed that the new program reduces not only communication but also capacity misses. And Figure 15 showed that unlike the old program, this program speeds up better on DASH for bigger data sets. These results suggest that the new program has tighter working sets than the old program. Figure 18 illustrates this, particularly by comparison with Figure 9.

Let us examine how the working set scales. Figure 18(a) shows that unlike in the old program, the working set of the new program decreases (slowly) with increasing number of processors. There is reuse of intermediate image data across phases and more reuse of volume data as well. Since the size of a processor's assigned block of scanlines contracts when the number of processors increases, less cache is needed to maintain enough temporal locality and the working set is smaller. For example, Figure 18(b) shows the working sets for the MRI brain data sets for the new algorithm on the simulator with 32 processors. Even for the data set as big as 512^3 voxels, the working set is as small as 64K bytes.

5.5 Performance on Other Platforms

We have also ported the parallel programs to other platforms. In this subsection, we present the results on two of these: a hardware coherent SGI Origin2000 machine, and a software-coherent shared virtual memory platform that provides coherence at page granularity.

5.5.1 SGI Origin2000

The SGI Origin2000 is a recent scalable, cache-coherent shared address space multiprocessor with physically distributed memory. It can be expanded to connect up to 128 195Mhz R10000 processors by a hypercube (or fat-cube, beyond a point). We use a 16-processor system with four routers connected by the CrayLink interconnect. Each router holds two nodes. Each node has two processors, and each processor has separate 32KB first-level instruction and data caches and a unified 4MB second-level cache. The second level cache is 2-way set associative with a cache line size of 128 bytes. The raw memory bandwidth on a node board is 780MB/sec, and the CrayLink interconnect provides a raw node-to-network bandwidth of 780MB/sec in each direction as well. Processor caches are kept coherent by a distributed directory-based protocol [7].

Figure 19 shows the performance of the old and new parallel programs on the Origin2000. The results validate those we have seen so far on other systems and on the simulator. The new algorithm outperforms the old one significantly. The Origin2000 provides performance debugging tools that use the hardware performance counters on the MIPS R10000 microprocessor [11]. These tools help characterize the dynamic behavior of the multi-level cache hierarchy, by sampling the frequencies of specified events such as cache misses at each level. They were able to tell us that a large amount of execution time was spent on cache misses for the original parallel program. However, they couldn't provide more detailed information; for instance, whether these cache misses

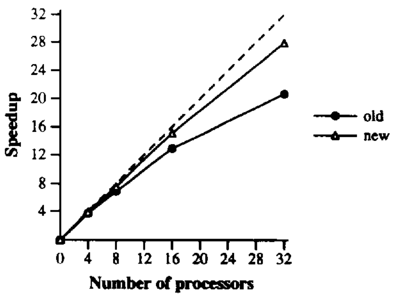


Figure 19: Speedups of both the old and new parallel shear warpers with the 511 × 511 × 333 voxel MRI brain data sets on SGI Origin2000.

are due to capacity misses or conflict misses, and whether the high cache miss overheads are due to high miss rates or contention. The tools can help understand miss rates at a per-procedure level, which is very useful, but they do not provide further insights.

5.5.2 Shared Virtual Memory

Our second platform is a page-based shared virtual memory (SVM) [9] system. While the performance of SVM on real applications is not well understood, it is allows us to support shared address space in software on networks of PCs and workstations without hardware support.

We use a simulated SVM platform that implements an all-software home-based lazy release consistency (HLRC) protocol [10]. It models an architecture of SMP nodes connected by a commodity Myrinet-like interconnection. Each node has 4 processors, and one I/O bus network interface on the I/O bus. It assumes 200MHz 1 CPI processors, 400MB/sec memory buses and 100MB/sec I/O buses. The width of the memory and I/O buses are 8 bytes and 4 bytes respectively. The page size is 4KB. The data cache hierarchy consists of a 8KB first-level direct mapped, write-through cache and a 512KB second-level 2-way set-associative cache, each with a line size of 32 bytes. Contention is modeled at all levels except the network links. More information about this platform can be found in another paper in these proceedings [1].

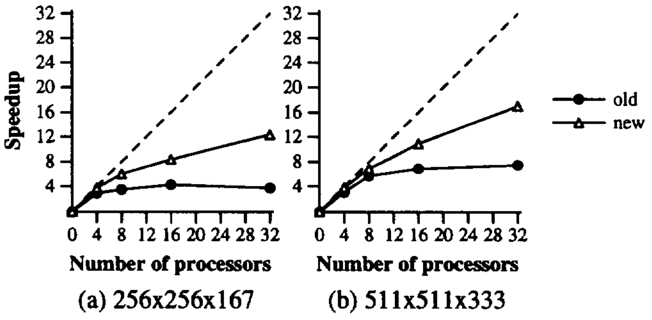


Figure 20: Speedups of the two parallel shear warpers on the SVM platform with the MRI brain data sets.

Figure 20 demonstrates that the new algorithm substantially outperforms the old one on the SVM platform. Figure 21 reveals that on SVM platform, the old parallel program has extremely high data and barrier wait time. Data

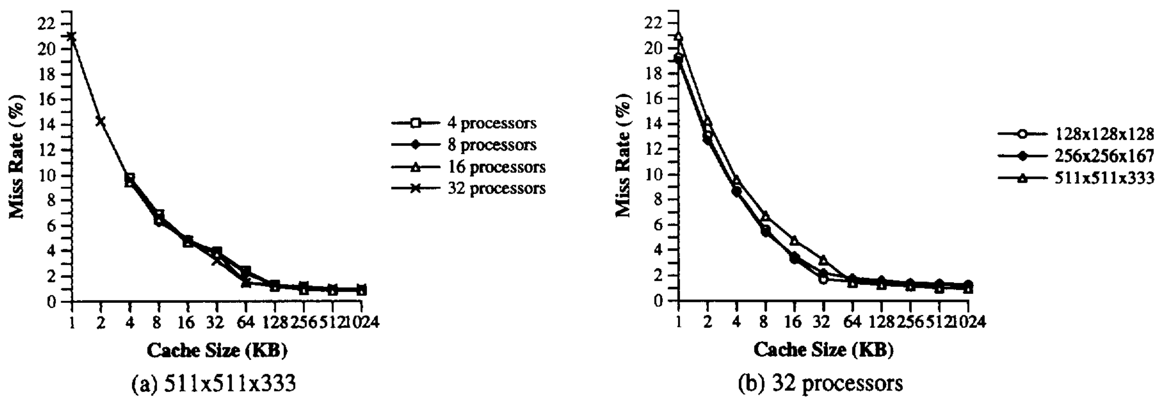


Figure 18: Working set for the new parallel shear warper with the MRI brain data sets.

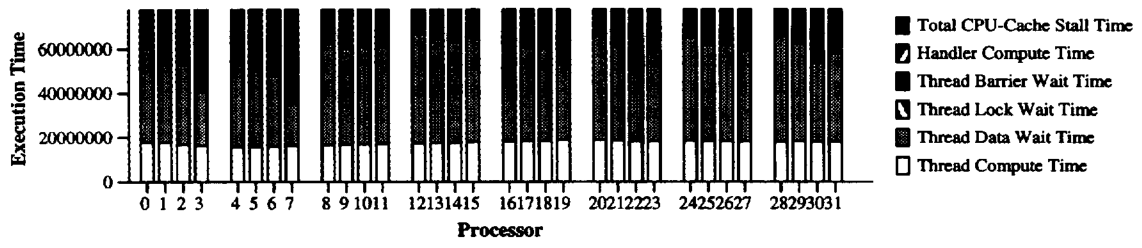


Figure 21: Breakdown of the execution time of the old parallel shear warper on the SVM platform with 511x511x333 voxel MRI brain.

wait time is the time spent waiting for data to arrive at remote page faults, i.e. the time spent waiting for communication. Barrier wait time is the time spent waiting at barriers including the wait time and the overhead of the synchronization event.

With SVM's coherence and communication granularity being a page, interleaved chunks that are smaller than pages artificially introduce more communication traffic, due to false sharing and fragmentation of communication, in addition to the inherent communication between phases. The barrier between the compositing and warp phases is very expensive, not due to the barrier operation itself and or the computational load imbalance, but due to the contention induced by communication. Detailed investigation through the simulator shows that this contention causes the memory bus in a node to delay delivery of the synchronization messages of the barrier, further increasing its overhead.

In addition to reducing inherent communication, the new parallel algorithm makes a processor's access patterns much more coarse grained. This is much more important here with the larger granularity of coherence and communication and the higher cost of communication. Also, the identical partitioning of the intermediate image for both the compositing and warp phases *eliminates* the barrier between them.

Figure 22 verifies that the new parallel shear warper greatly reduces communication and contention, resulting in less data and barrier wait time. Even though the lock overhead is a bit higher, resulting from the possibly smaller chunks of stealing, the overall performance on SVM is improved significantly.

6 Conclusion

We have presented a new parallel volume rendering algorithm based on shear warp factorization. It assigns a contiguous block of scanlines of the intermediate image to each processor, addressing the load balance problem through a combination of profiling and task stealing. This same partitioning scheme is applied across both the compositing and warp phases, so that data locality is improved and inherent interprocessor communication is greatly reduced. The nature of the new partitioning also reduces the working set, further reducing non-inherent communication.

By substantially improving communication and data locality, our parallel algorithm achieves good speedups and fast rendering times on a range of very different shared address space multiprocessors, including bus-based and distributed hardware cache-coherent machines, and vastly improved speedups on software page-based shared virtual memory platforms. The new algorithm outperforms the old one, with the magnitude of improvement increasing as more locality issues arise on the platform and communication becomes more expensive. We used a hierarchy of increasingly detailed performance diagnosis experiments on real machines and simulators to obtain a systematic understanding of performance issues. This helped us identify the causes of the performance bottlenecks in the existing parallel program and led to the design of the new version. The performance diagnosis tools we used either gave us very coarse-grained data on real machines or were slow simulators that don't model real machines exactly. Nevertheless, each gave us useful information of a different type, and we were able to use them efficiently in conjunction with algorithmic and application knowledge.

Although simulators do not model any particular system

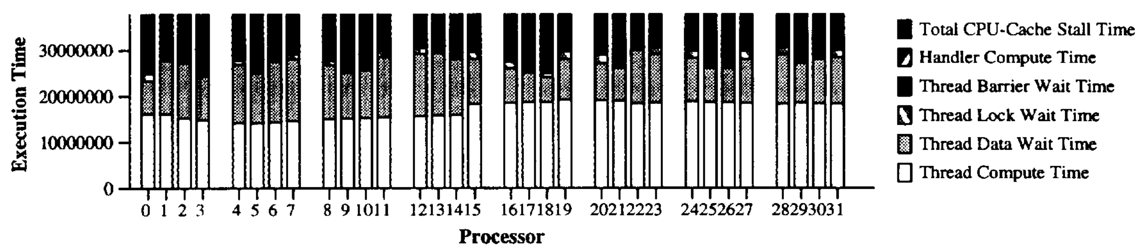


Figure 22: Breakdown of the execution time of the new parallel shear warper on the SVM platform with 511x511x333 voxel MRI brain.

exactly and can lead to different interleaving of accesses, the types of detailed characteristics they can measure were extremely useful (e.g. that it was true sharing rather than false sharing or other misses that dominated, whether it was miss rates or contention that dominated, and where in the code the problems were encountered). Better integrated and easy-to-use tool hierarchies on real systems would be very useful, even if some of them perturb the execution significantly since many of the key characteristics are not affected very much by perturbation. More efficient support for user-level runtime profiling of fine-grained computations, without inserting too many extra instructions, would have been useful as well, to support profile-based dynamic load balancing.

This resulting parallel shear warp algorithm demonstrates that real time volume rendering is promising on general-purpose shared address space multiprocessors. We would like to examine how it scales to even larger data sets and systems, as well as how to develop performance diagnosis tool hierarchies—hardware and software—for real machines that would have helped us come to the desired insights and improvements more rapidly.

Acknowledgment

We would like to thank Philippe Lacroute for his help on the source codes of the original shear-warp program, and providing experiment data sets.

References

[1] Jiang D, Shan H., and Singh J.P. Performance Portability of Applications and Optimizations Across Shared Address Space Multiprocessor. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 1997.

[2] Lenoski D., Laudon J., Joe T., Nakahira D., Stevens L., Gupta A., and Hennessy J. The DASH Prototype: Implementation and Performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 92–103, May 1992.

[3] Rothberg E., Singh J. P., and Gupta A. Working sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 14–25, May 1993.

[4] Lacroute P. G. *Fast Volume Rendering Using a Share-Warp Factorization of the Viewing Transformation.* PhD thesis, Stanford University, 1995.

[5] Lacroute P. G. Real-Time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp Factorization. In *Proceedings of 1995 Parallel Rendering Symposium*, 1995.

[6] S.A. Herrod. *TangoLite; A Multiprocessor Simulation Environment.* Computer Systems Laboratory, Stanford University, 1994.

[7] Laudon J. and Lenoski D. The SGI Origin: A ccNUMA Highly Scalable Server. In *To appear Proc. Intl Conference on Computer Architecture 1997*, 1997.

[8] Nieh J. and Levoy M. Volume rendering on scalable shared-memory MIMD architectures. In *Proceedings of the 1992 Workshop on Volume Visualization*, pages 17–24, 1992.

[9] Li K. and Hudak P. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986.

[10] Iftode L., Singh J. P., and Li K. Scope Consistency: a Bridge Between Release Consistency and Entry Consistency. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[11] Zagha M., Larson B., Turner S., and Itzkowitz M. Performance Analysis Using the MIPS R10000 Performance Counters. In *Supercomputing'96*, 1996.

[12] Singh J. P., Gupta A., and Levoy M. Paralle Visualization Algorithms: Performance and Architectural Implications. *Computer*, 27:45–55, 1994.

[13] Woo S., Ohara M., Torrie E., Singh J.P., and Gupta A. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 1995.