A Parallel Splatting Algorithm With Occlusion Culling

Jian Huang[+], Naeem Shareef[+], Roger Crawfis[+], Ponnuswamy Sadayappan[+] and Klaus Mueller[*]

[+]Department of Computer and Information Sciences, The Ohio State University, Columbus, OH
[*]Computer Science Department, SUNY-Stony Brook, Stony Brook, NY

Abstract

Splatting is a volume rendering technique that projects and accumulates voxels onto the screen. It is able to incorporate a variety of reconstruction kernels without extra computational overhead, as well as reduce computational and storage costs using a sparse volume representation. Previous splatting algorithms suffered from artifacts because they incorrectly separate volume reconstruction and volume integration. The IASB (Image-Aligned Sheet-Based) splatting overcomes these problems by accumulating voxels onto sheets aligned to be parallel with the image plane. In addition, it introduces a novel approach for splatting to cull occluded voxels using an opacity map, called an occlusion map, that provides a substantial speedup in serial implementations.

Parallel approaches to volume rendering are able to overcome the enormous amount of computation required. We present a parallel solution to the IASB splatting approach that leverages its image-aligned volume partition construction. The occlusion culling technique poses difficulties in developing a parallel solution due to its inherent serial nature. Our approach assigns processors to render data closest to the screen first and propagates an opacity map to avoid rendering occluded data. Data communication is substantially reduced making use of occlusion culling, as well as view coherence. We utilize a dynamic load balancing scheme where tasks are partitioned in image space. Our implementation runs on both NOW (network-of-workstations) and MPP platforms. We present results and analysis on a NOW cluster of Pentium II's.

1. Introduction

Volume rendering has been widely applied to visualize volume data sets in such fields as computational science, engineering, and medicine. A volume is a tessellation of a bounded region in 3D space by volume cells, called voxels (or volume elements). Volume rendering is an enormous computational task where data set sizes of 512^3 are now commonplace. Algorithms can be classified as *image order, object order, or hybrid* approaches. To accelerate volume rendering, researchers have reported using parallel solutions and specialized hardware. Parallel systems, such as MPP's [21] and NOW's [2][3], provide flexibility as well as scalability which are useful to many practitioners, while specialized hardware are currently only capable of interactive renderings of small sized datasets.

Splatting [23] is an object order approach. The flexibility in splatting to choose reconstruction kernels, with no extra costs, distinguishes splatting from the other methods. In addition, it allows for a sparse volume representation that holds only non-transparent voxels. Previous splatting approaches suffer from artifacts, that are overcome by the Image-Aligned Sheet-Based (IASB) splatting approach [16]. IASB splatting accumulates voxels onto image-aligned sheets that span the volume. In addition to the image quality improvements achieved, an extension to the algorithm [18] introduced a novel mechanism to cull occluded data in rendering. This acceleration provides substantial speedups, especially for opaque data sets. For example, to rendering a 256^3 volume data set (Fig. 3b) takes 145 sec/frame on a PC with 300MHz Pentium II. With occlusion culling, the time is reduced by an order of magnitude to 17 sec/frame.

In this paper, we present a parallel solution to the IASB splatting with occlusion culling. The methods presented in the existing parallel splatting literature [6][11][13][14] can well be extended to perform IASB splatting for cases where occlusion is light (transparent data sets or data sets that branch heavily like a nerve neuron) with good scalability. Here, we instead focus on a more challenging case, that is, the case where heavy occlusion is available. For the sample data set (Fig. 3b), assuming perfect scalability, without occlusion being considered, one needs 10 nodes to render the data set in parallel to achieve a rate of one single node using occlusion culling. A meaningful parallelization needs to use occlusion acceleration in such a case. However, due to its serial nature, occlusion culling is non-trivial in a parallel solution.

We use a view dependent object-space data partition and an image-space task partition. We utilize a dynamic load balancing scheme where workload is estimated on a per tile basis. Our algorithm keeps an occlusion map to cull occluded data. Only visible data need to be communicated within a single view. Between multiple views, data communication is further reduced by taking advantage of view coherence. Our results show scalable speedups over multiple frames on a NOW cluster with 16 Pentium II 300 MHz nodes. For the data set just mentioned, our implementation renders at 1.5 sec/frame using all 16 nodes.

In Section 2, we briefly describe previous parallel approaches to volume rendering. Next, we describe the IASB splatting algorithm in Section 3. Our parallel solution to IASB splatting is presented in Section 4. We discuss the system architecture, results and analysis of our parallel implementation in Section 5. Finally, Section 6 concludes the paper and discusses future work.

{huangj, shareef, crawfis, saday}@cis.ohio-state.edu, 395 Dreese Lab, 2015 Neil Ave., Columbus, OH 43210, USA.
{mueller}@cs.sunysb.edu, Stony Brook, NY 11794-4400, USA.

To appear in the 3rd Eurographics Workshop on Parallel Graphics and Visualization, September 2000, Girona, Spain.

2. Parallel Volume Rendering

Much research towards parallel ray-casting has been reported in the literature, primarily due to the simplicity of the algorithm. To avoid volume data redistribution costs, researchers have proposed the distribution of data to processing nodes, where each node processes its own data for all frames or views. Each node generates a partial image with its data, which are then accumulated and composited into the final image [8][12][13][15].

Researchers have also investigated partitioning screen space into square tiles or contiguous scanlines, to be used as the basic task to be sent or assigned to processing nodes. For better load balancing, the task queue can be ordered in decreasing task size, such that the concurrency gets finer until the queue is exhausted[4]. Load balancing can also be achieved by having nodes steal smaller tasks from other nodes, once they have completed their own tasks [20][25]. Finally, timeout stamps for each node can be added, such that if the node can not finish its task before the timeout, it takes the remnant of the task, re-partitions it and re-distributes it [5].

Shear-warp factorizes the viewing matrix into a 3D shear, a projection and a 2D warp to the final image. Shear-warp has problems with resampling, but is fast in nature [9]. In [10], a parallel shear-warp implementation on shared-memory architectures has been reported with decent timing benchmarks. Amin et. al [1] ported the shear-warp algorithm onto a distributed memory architecture, by partitioning in sheared volume space and using an adaptive load balancing. [22] improves the parallel shear-warp implementation on distributed memory architectures by dividing the volume data after the shear operation into subvolumes parallel to an intermediate image plane of the shear-warp factorization.

Splatting and cell projection methods have also been parallelized using a sort-last paradigm. The community has researched parallel splatting algorithms [11] that do not utilize occlusion-based acceleration. The volume data is distributed in either slices (axis-aligned planes) [6] or blocks [11] to processing nodes. Those are then rendered, in parallel, to partial images which are composited for the final image by the master node. Speed-ups can further be achieved by only passing the non-empty parts of the partial images [6] or by parallelizing the final compositing stage using a screen space partitioning [11]. Hierarchical data structures such as a k-d tree can be applied to facilitate prompt compositing and occlusion culling [13]. Machiraju and Yagel [14] report a parallel implementation of splatting, where the tasks are defined by a subvolumes. Each processor is assigned a subvolume. The images are composited together in depth-sort order, also performed in parallel.

For both splatting and shear-warp, the amount of work is directly related to the number of relevant voxels. Balanced partitions are difficult, and complicated adaptive load-balancing schemes are needed. While with ray-casters, distributing data in a balanced fashion is often easy, though load balancing remains an issue.

3. IASB Splatting With Occlusion Culling

Splatting, first presented by Westover [23], projects voxels in depth-sorted order. Volume reconstruction and volume integration are performed by placing a 3D reconstruction kernel, e.g. a 3D Gaussian, at each voxel, and then accumulating projections of these kernels onto the image plane with 2D kernels called *splats*. Computationally, each splat is represented by a discrete *footprint* table which holds the integration of the 3D reconstruction kernel and can be pre-computed. A voxel is accumulated onto the image by first shading it, and then resampling its footprint, weighted by the voxel's color and opacity, at each pixel within its extent. Westover's [23] first algorithm individually projects voxels and accumulates them onto the screen. This algorithm is prone to color bleeding, as shown in [16]. To address this problem, Westover [24] introduced an algorithm that first accumulates voxels onto axis-aligned sheets that partition and span the volume, and then composite these sheets into the final image. Unfortunately, this approach introduces severe popping artifacts in animations when the axis alignment changes abruptly as the view moves [16].

To overcome these problems, recently the IASB splatting algorithm [16] was introduced. Here, sheets are aligned to be parallel with the image plane. Given a view, sheets are uniformly distributed along the view direction within the z-extent of the volume. The spacing between two neighboring sheets is the *sheet thickness*. The number of sheets is computed by dividing the *z*-depth extent of the volume by the sheet thickness. An improvement in image quality was achieved by slicing the 3D reconstruction kernel into an array of 2D voxel sections, each representing a partial integration through the voxel. A footprint table is an array of footprints, whose size is determined by dividing voxel extent by the sheet thickness. It is indexed by which subsection of a voxel falls between two sheets. In addition, when using FTB traversal, it also allows for the ability to cull occluded voxels from rendering. An occlusion map, or buffer, holds the accumulated opacity at each pixel during rendering. This map is successively updated and passed to the next sheet. In sum, the IASB splatting algorithm performs the following steps:

Pre-compute footprint table array;
For each view **do**
  Initialize the occlusion map to zero opacity;
  Transform each voxel (v_x, v_y, v_z) to eye space (s_x, s_y, s_z);
  Bucket-toss the voxels by *z*-location into an array of buckets, i.e. one bucket per sheet. A bucket represents the area between two neighboring sheets. A voxel is inserted into a bucket if its extent overlaps it. Depending on the sheet thickness, a voxel may be inserted into more than one bucket.
  For each sheet **do**
    For each voxel in bucket **do**
      If this voxel is not occluded **then**
        For each pixel within extent of the footprint **do**
          Resample the footprint and update the pixel;
          Update the occlusion map with the pixel opacity;
        End;
    End;
    Composite the sheet onto the final image;
  End;
End;

4. Parallel IASB Splatting

4.1 Overview

Most parallel approaches to splatting have used an object-space partitioning of the volume, assigning these subvolumes to processing nodes. It is shown that such data partition minimizes data communication [19]. To parallelize the IASB splatting algorithm in the same manner, each node will compute multiple sheets, which must then be composited in order. When occlusion is not considered or there is little occlusion, e.g. highly transparent or very lightly

occluded data sets, such an approach would very likely offer scalable parallel implementations. While this is very useful in practice, we opt to focus ourselves on the more interesting and challenging case, the case when there is heavy occlusion. As we presented in Section 1, even with a linear speed up, this simple approach needs 10 nodes to render the sample data set at a rate that one single node can achieve with occlusion culling. In order for a meaningful parallel speed up under such a scenario, this pure object-space approach can not leverage the speedups gained by occlusion culling. On the other hand, while image-space partitioning will be able to utilize occlusion culling, it is prone to inferior load balance as well as expensive data redistribution costs. We designed a hybrid data partitioning approach to take advantage of both object space partitioning and image space partitioning.

The sheet thickness corresponds to the step size in a Riemann sum approximation to the volume integration. To define work based on sheets is too fine grained. Instead, we group successive sheets together to define a larger image-aligned region, called a *slab*. All sheets within a *slab* are composited to an image of this *slab*, called a *slab image*. The final image is obtained by compositing the *slab* images. One straightforward parallel scheme to IASB splatting is to simply assign a slab per processor and then accumulate the slab images. As with simple object-space task partitioning schemes, this approach is not able to exploit inter-slab occlusion in a parallel fashion. The occlusion culling makes the IASB algorithm serial. In order to determine if data within a slab is occluded, all the slabs in front of it must be computed first.

In the algorithm we devised, data is partitioned into bricks. All processors work on a single *slab* before the next *slab* image is computed. We use a task partitioning scheme that divides image space, i.e. the slab images, into image tiles. The screen space partitioning is performed dynamically, by building a quadtree for each slab of each view. The actual tiles, termed *macrotiles*, assigned to rendering nodes are usually at a non-leaf level in the quadtree. We balance loads by estimating the workload per *macrotile* with a heuristic based on bricks. An occlusion map, initialized at the frontmost slab, is propagated with the task assignments and updated upon each task's completion. In the following sections we describe our algorithm in more detail.

4.2 Volume Partition & Brick Culling

We subdivide the volume into *bricks*, of size $v \times v \times v$. This grouping of voxels allows the definition of a work load on a macro scale rather than at each voxel. In addition, data communication favors larger message sizes due to the constant per-message startup costs. From experiments, we choose v to be 8 in our implementation. As mentioned before, splatting allows for a sparse representation, and thus each brick contains only relevant, i.e. non-transparent, voxels. Each voxel is represented by its location as three floats (x,y,z), an 8-bit density value, and an 8-bit per component gradient (n_x, n_y, n_z). This amounts to 16 bytes per voxel.

A summed area table (SAT) computed over the occlusion buffer is used to cull away bricks directly [7]. For each brick, we construct a tight bounding box of the sparse voxel list within it. Given a viewing direction, we quickly project the 3D bounding box into the framebuffer. We then calculate a 2D bounding box of this projection. After retrieving the values in the SAT occlusion map, corresponding to the four corners of the 2D bounding box, we can easily compute the average opacity in this region. If the average opacity is already 1.0, then this entire brick is occluded.

4.3 Image-Space Partition & Load Balancing

We employ an image-space task partitioning scheme. An image region can be one of these types: *empty*, *fully opaque* or *non-fully opaque*. An empty image region has no voxels in the sparse volume representation that project to it, and is not rendered. Fully opaque regions will not be rendered because all pixels will have full opacity. Only non-fully opaque image regions have to be rendered. Of course, an image tile can be of different types in different slabs, however, fully opaque tiles remain fully opaque.

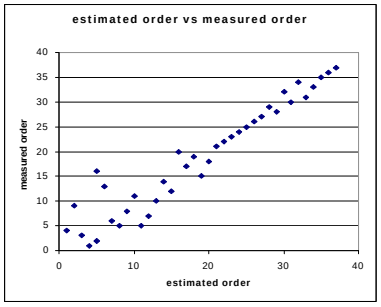
We need a heuristic for workload estimation, such that load balancing schemes can be devised on this basis. One method is to use the number of voxels projecting to a region as the estimation. But this approach does not take into account the rendering work that won't occur because of occlusion. Instead, the heuristic function we use is the number of non-opaque pixels that are overlapped by one or more bricks within a image region. This is straight-forward to calculate by using the bounding boxes of bricks' projections. The reason for this heuristic is that when a pixel has reached full opacity, it is not updated any more in IASB splatting, whereas, empty pixels will never require work. With an opaque data set, a pixel reaches full opacity within a small number of updates. When occlusion is to take place, the number of voxels to be rendered does not relate well to the total amount of work needed. As a result of image-space partitioning, we obtain a set of non-fully opaque image regions with an estimated per region workload. This set of image regions is sorted, in decreasing order of estimated workload, into a queue. We then assign the image regions in the queue to rendering nodes in a round robin fashion, starting from the head of the queue. After a rendering node completes its task, it requests another task from the queue. If no more tasks are left in the queue, this rendering node idles, waiting for the start of the next slab.

For load balancing, we desire the partitioned tasks to be fine grained. Smaller image regions are favored. But as we will show in Section 5.1, we would also like to control the total number of tasks. The reason is that splatting employs large reconstruction kernels, which provide better interpolation. Unfortunately, this also leads to heavy overlapping across partition boundaries. To ensure correctness, any splats and bricks overlapping a boundary must be duplicated at and processed by all nodes processing image regions neighboring that boundary. Hence, the total amount of work increases. When the screen extents of a splat or brick are comparable to the size of an image tile, this overlapping effect considerably increases the collective total amount of work, leading to a decrease in parallel speedup. Therefore, we desire larger image tiles for efficient speedups. These two opposing needs form a trade-off. Furthermore, this trade-off is view and data set dependent. We can only search for the optimum in the trade-off via experimentation. To achieve an optimum in this trade-off, one must look for the image region size that is *just* small enough for acceptable load balance. Therefore, we allow the user to input a parameter defining the 'desired' ratio of number of image regions (tasks) vs number of rendering nodes. By tweaking this ratio parameter, an optimal trade-off with respect to a data set can be found. To achieve good speedup for our data sets, the ratio should be between 2.0 and 3.0.

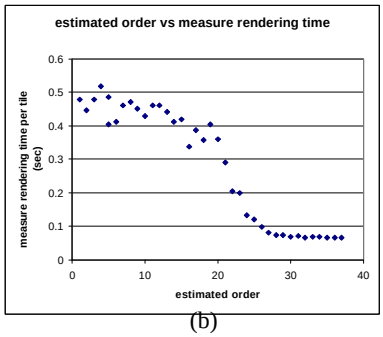
The image-space could be partitioned statically, which is simple and efficiently indexed. But to allow for easy search for the optimum in the trade-off, we need the flexibility to vary region sizes. A quadtree hierarchy is implemented in our algorithm, with the leaf nodes being small static non-empty tiles. The tasks are assigned at a level higher than the leaf tiles, i.e. *macrotile*. We choose the *mac-*

rotile size by building and inspecting the image tile quadtree of each *slab* image. The leaf nodes of the tree are 16×16 pixels in our implementation. We partition a slab image by first computing the desired number of *macrotiles* from the ratio: (# of *macrotiles*)/ (# of nodes), supplied by the user. The quadtree is then built in a bottom-up fashion, until a level is reached where the number of tiles found to be rendered on that level is more than the desired number of *macrotiles*. From our experiments we found that typically, a full quadtree is not built. Rather, we usually observed only around three levels. *Macrotiles* are then placed onto a task queue, sorted and assigned to processors.

In Fig. 1, we present an analysis our workload estimation scheme on real data. An arbitrary view of the Brain data set (Fig. 3b) was rendered in parallel with 9 rendering nodes. For the first slab, our algorithm decided to use a *macrotile* size of 64, and found 38 such *macrotiles* to render. It estimated the work load, and sorted the *macrotile* queue in decreasing order. During rendering, we recorded the actual time the rendering nodes took to render each *macrotile*, as well as the order in which each *macrotile* is completed. In Fig. 1a, we compare the order based on estimated workload with the recorded order in which *macrotiles* got rendered, using each order as one axis. The ideal case would show a linear 45 degree line. We see this pattern for *macrotiles* after the 16th position in the estimated queue. We see a rather unorganized pattern below the 16th position because the first 16 *macrotiles* are completely covered by bricks. The estimated order among the first 16 *macrotiles* was less determined, as shown by Fig. 1b, where we show the estimated order vs. the measured rendering time. The first 16 *macrotiles* took about the same amount of time with minor variations. After the first 16 *macrotiles*, the rendering times show a sorted order.



(a)



(b)

Figure 1: (a) Sorted order based on estimated workload vs measured order. (slab 1 of 3-slabs setup, UNC Brain). (b) Sorted order based on estimated workload vs. measured per *macrotile* rendering time (sec).

4.4 Data Distribution & View Coherence

Using image-space partitioning, data distribution is un-avoidable. When the view changes, the image space data redistribution involves a large number messages. The small message sizes usually incurred are inefficient for communication, due to the constant per message start-up overhead. Effective load balancing schemes are limited with such data distribution schemes as well.

Alternatively, we implemented a master/slave framework. The slave nodes are the rendering nodes. They collaborate on each slab, rendering all the *macrotiles*. The master node plays the role of data server and task scheduler. It loads in the sparse volume data, and in a slab-by-slab fashion, builds up the task queue, *broadcasts* the un-occluded bricks to the rendering nodes. Then, the master assigns *macrotiles* to rendering nodes. After all the *macrotiles* are completed, the server updates the occlusion map with the results sent back, culls occluded data portions in the subsequent slab, and starts the cycle to render the subsequent slab.

Using the slab-by-slab approach and the broadcast operations is suitable for our goals. First, with *slabs*, the occlusion results of the slabs in the front can cull away the occluded portions in the following slabs from being processed. The total amount of data communicated is a close approximation to the un-occluded portion of the total volume, and can be small in size and locally cacheable by all the rendering. Second, the large number of point-to-point or multicasts messages for data re-distribution are replaced with one single optimized collective communication operation, *Bcast*. Third, when all the rendering nodes locally cache the un-occluded volume portions, any flexible load balancing scheme can be adopted with no additional data redistribution. Finally, with caching enabled on the rendering nodes, newly appearing volume portions are incremental. The communication is affordable and fast. Efficient multi-frame rendering is supported.

Fig. 2 further illustrates this broadcast based data distribution scheme. Fig. 2a shows an initial view and the IASB algorithm using 4 volume slabs. The white areas represent the parts of the volume that are occluded, thus not broadcast, while the small pieces in black are. When the viewpoint changes, as shown in Fig. 2b, the areas shown in grey represent bricks that were not broadcast in the previous view. If the rendering nodes are able to cache the data received for a previous view, less amount of data communication is required, and essentially, each rendering node would then possess locally the non-occluded portion of the volume. This data is discovered on the fly.

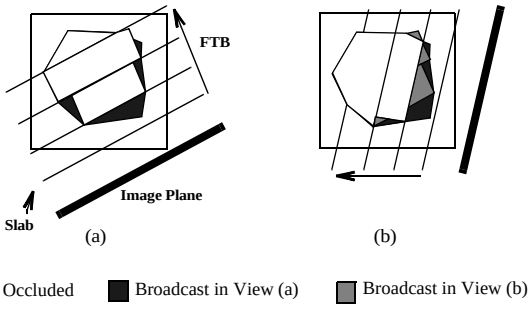


Figure 2: Data broadcast for two nearby views for an opaque object. (a) A volume rendered from an initial view. The white option is occluded and is not broadcast. The bricks in the dark pieces are broadcast and rendered. (b) The data (grey pieces) to be broadcast for a next (close-by) view.

Our experiments in Section 5.2 show that for a practical data set (assuming 5 degree/frame movement), this incremental amount of newly un-occluded volume is only 1% of the total sparse volume, amounting to less than 0.5% of the total raw volume size. The data resident on each rendering node is only ~15% of the total volume. Hence, at little communication costs, we achieve the flexibility to use most load-balancing schemes to adopt occlusion accelerations.

4.5 Summary

In summary, our basic parallel algorithm is as follows.

Pre-compute a brick partition structure over the volume (relevant voxel list in each brick);

Server node

Initialize occlusion map/image;
For each slab in FTB order **Do**
  Construct quadtree of non-empty/opaque tiles of this slab;
  Build quadtree and collect macrotiles
  Build a queue sorted by estimated workload
  Broadcast unoccluded bricks to all rendering nodes;
  For each **macrotile** in **queue**
    Assign to a rendering node;
    Receive/Combine the result into the slab image;
    Update occlusion map for full image;
  End
End;

Rendering nodes

Receive assigned macrotile
For each sheet in current **macrotile do**
  Apply occlusion test for voxel and brick culling
  If exist un-occluded voxels
    Splat into sheets with occlusion testing;
    Collect computed tiles
    Update occlusion map,
    Composite the **tile** to image;
  Endif;
End;
Send resulting macrotile to server node

Of course the actual implementation overlaps communication and computation as much as possible. This is not illustrated in the pseudo code to aid in understanding.

5. Results & Analysis

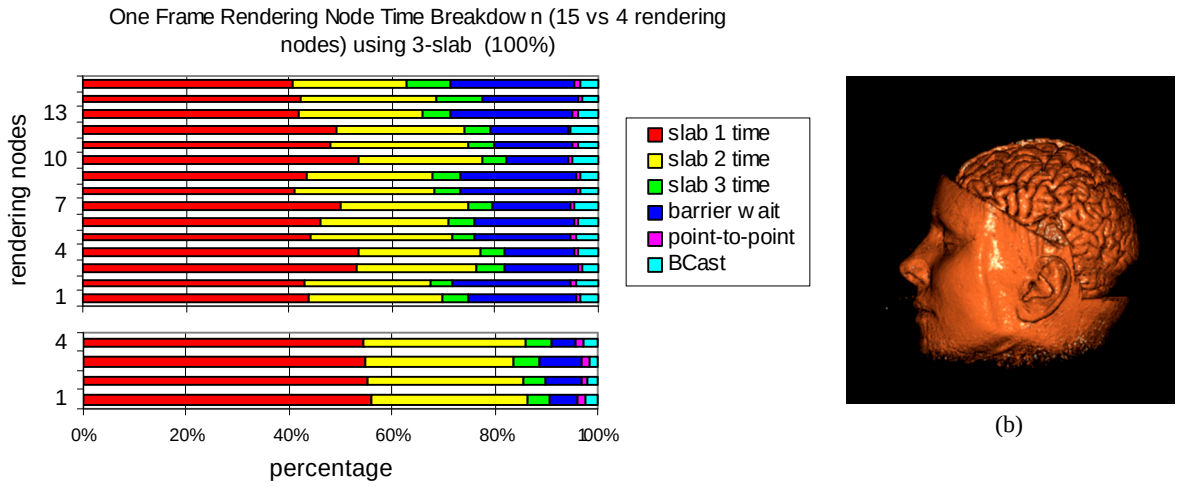
In this section, we evaluate some important aspects of the parallel algorithm that we have developed. These aspects include load balancing, data communication for new frames, the amount of data resident on each node, rendering timings using 3, 5, 9, 13 and 16 nodes, as well as the speed up curves and a scalability analysis.

We implement our parallel solution on a NOW cluster consisting of 16 Pentium II 300 MHz CPU's connected by a Myrinet/GM (1.2Gbps full duplex) interconnection network, running MPI. We dedicate one node as the Master, while the rest are assigned as Slave nodes whose sole job is to render the *macrotiles*. The sample data set we used for a detailed analysis is the UNC Head MRI data set, which has a $256 \times 256 \times 256$ voxels. Each voxel contains pre-computed gradients and position, amounting to 256 MBytes of data. Removing any voxels associated with air, reduces this total to 80MBytes. We use a 3D Gaussian with a 2.0 object space radius. The image resolution is 512×512. No graphics hardware is used. We simulate the user interaction by rotating the user around the center of the volume at 5 degrees/frame.

5.1 Load-Balance

The brick organization adds to the problem described in Section 4.3, where small slab-tile sizes actually increase the total amount of work to be performed. Reducing the brick size results in a trade-off between more overhead in data management/communication and better load balancing. Due to the space limit, we do not present results pertaining to varying brick sizes.

The communication with each slave rendering node involves receiving a broadcast, Bcast, update for the un-occluded data in the current slab, and performing point-to-point communication with the master node to obtain a macrotile to work on. The master node sends the slave a *macrotile*, the slave node renders into this *macrotile*, when done, sends the *macrotile* back to the master via point-to-point communication. The slave then waits for a next *macrotile* in the slab. If no *macrotiles* are left in the queue, the slave waits until all the other nodes finish and the master node broadcasts data for the next slab. There are 4 types of working states that a slave can be in: rendering, participating in Bcast, point-to-point communications, or waiting. A break down of the timings for an arbitrary



(b)

Figure 3: (a) Break down of a typical view in the frame sequence, rendering with 16 nodes (upper, 15 slave rendering nodes), and 5 nodes (lower, 4 slave rendering nodes), using the 3-slabs setup. The slab rendering time is shown separately for each slab. (b) Sample image of this frame.

frame (3-slab) using 16 nodes (15 slave nodes) and 5 nodes (4 slave nodes) is shown in Fig. 3a. The 16-node break down is shown at the top, with the 5-node results at the bottom.

In both cases, the communication time, both Bcast and point-to-point, is minor, comprising only about 5% of the total time on average. However, the barrier wait consumes about 15% to 20% of the total time with 16 nodes, where the level in the image plane quad tree is at a *macrotile* size of 64 pixels. For tiles of this size, the macrotiles on the boundary of the human head's projection can become significantly smaller than the ones at the interior of the projection. Therefore, although we are using a tile workload estimation scheme and building a sorted task queue based on the estimation, the lack of smoothness in the workload variation from the front towards the end of the queue degrades the load balancing. Constrasting this to the 5-node case, the *macrotiles* are of 128 size and much better load balancing is achieved. The total rendering time (summation of the rendering time for slabs 1, 2, 3) takes up more than 90% of the total spent time per frame.

Parallel computing researchers often use a dynamic load balancing scheme in which the tasks are dynamically partitioned to guarantee proper load balancing. We implemented this, but did not see an improvement in the overall timing. This is due to the overlapping effects discussed in Section 4.3. We are left with the predicament that larger tile sizes reduce the amount of overhead, but smaller tile sizes produce better load balancing. Our timing shows that with 64 sized macrotiles, as shown in Table 1, the summation of the rendering time on all the 15 slave rendering nodes is 16.6 seconds. When the macro-tile size is reduced to 32 pixels, the summed rendering time almost doubles to 27.88 seconds, but the wait time is cut in almost half.

TABLE 1. Macro-Tile Size vs. Rendering Time (sec) Increase and Load-balancing Improvement

Macrotile Size	64 pixels	mix 64/32 pixels	32 pixels
Summed Rendering Time	16.6	22.24	27.88
Max Wait Time	0.28	0.21	0.15
Avg Rendering Time/Node	1.4	1.66	2.00

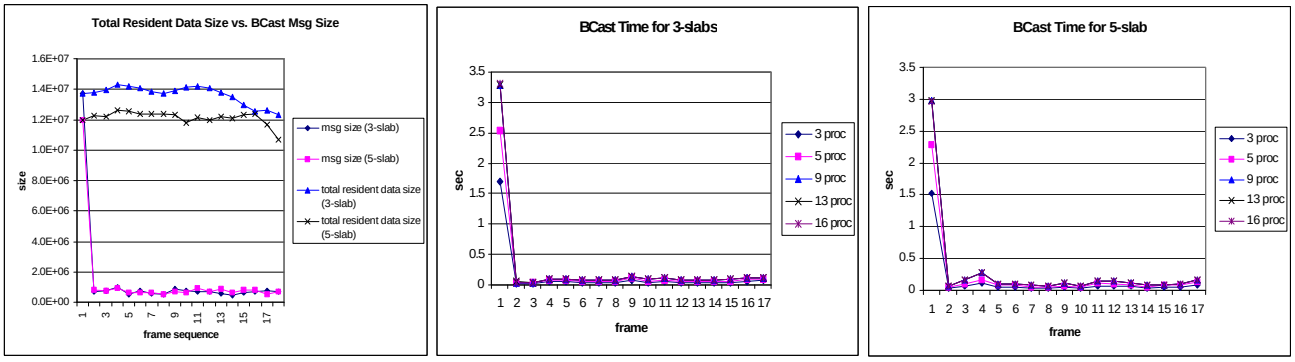
In an effort to try to alleviate this, after the *macrotile* queue is built, we use 64 pixel tiles for initial tasks and partition the *macrotiles* left in the queue to 32 pixel tiles. For this configuration, the total rendering time increased to 22.24 seconds, and the maximum wait time was reduced to 0.21 seconds from the strictly 64-tile configuration. This improvement in load-balancing or wait time does not overcome the extra time incurred in the rendering. Therefore we did not include this extension as a solution. The degree of granularity that the *macrotiles* can reach depends on the size of the splat footprints. Using smaller kernels or larger data sets will reduce the wait times, but increasing the image size effectively increases the projected splat size and increases the overhead.

5.2 Data Distribution

Figure 4 illustrates the savings achieved with our data distribution and culling scheme. Fig. 4a plots the message and data sizes using 3 and 5 slabs across many frames. The upper two curves in Figure 4a show the amount of data identified as possibly needed for each frame. Since occlusion occurs after slab boundaries, the 3-slab scheme has more resident data for the current view. The total amount of data needed for each view ranges from around 12 to 14 MBytes. This represents about 15% to 18% of the total volume data. The lower curves in Figure 4a represent the amount of data that needs to be actually updated for each new view. This amount is on average only about 6% of the resident data, or about 400Kbytes to 600Kbytes. Thus, the data updated via the broadcast is only about 1% of the total sparse splatting representation. This low amount of data communication and storage requirements allow us to easily replicate this data on every rendering node. There is an initial communications penalty, as the first view requires broadcasting all non-occluded data needed for that frame. Fig. 4b, and c show the actual time spent in the Bcast operation using 3-slabs and 5-slabs respectively. As expected, the initial view needs a rather large amount of time for the broadcast, but the broadcast time reduces substantially for subsequent views. The efficiency of the log(n) broadcast time can be seen in these figures, as the curves do not deviate much as the number of processors is increased. Using 5 slabs involves 2 more broadcast messages than with a 3 slabs. Due to the constant start up time involved in a broadcast (global barrier), a little more time is spent using 5 slabs than using 3 slabs on broadcast communications.

5.3 Speedup and Scalability

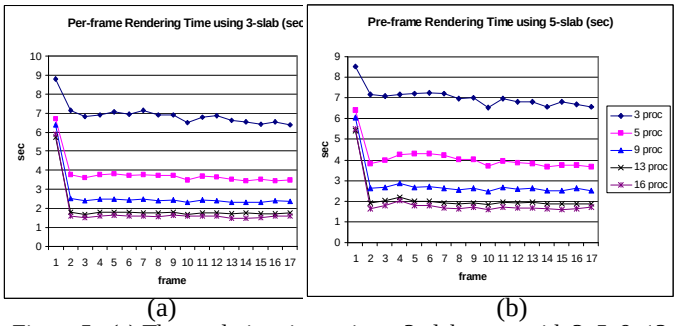
The wall-clock times to render the UNC Head data set, using 3-slab and 5-slab setups, are shown in Fig. 5a, and b, respectively. For each case, the rendering times with 3, 5, 9, 13 and 16 nodes are



(a) (b) (c)

Figure 4: Analysis of data distribution. (a) The upper two curves show the total size of volume data resident on each node for every frame, using 3-slab and 5-slab, respectively. The lower two curves show the total size of the actual messages Bcasted in each frame. (b) The Bcast time using 3-slabs for 3, 5, 9, 13, 16 procs, respectively. (c) The Bcast time using 5-slabs for 3,5,9,13 and 16 processors, respectively.

shown. The actual timings suggest that the 3-slab setup is slightly faster due to the two additional slab barriers in the 5-slab setup. The slab configuration is needed to cull down the amount of data broadcast and stored on each rendering node. Reducing the number of slabs/barriers would actually reduce the wait time. For evenly divided slabs, our experiments showed that more than 3 slabs are needed for a reasonably low storage requirement on each rendering node. Future research will examine slab partitioning schemes that partition the space non-uniformly to allow for wide slabs whenever possible.

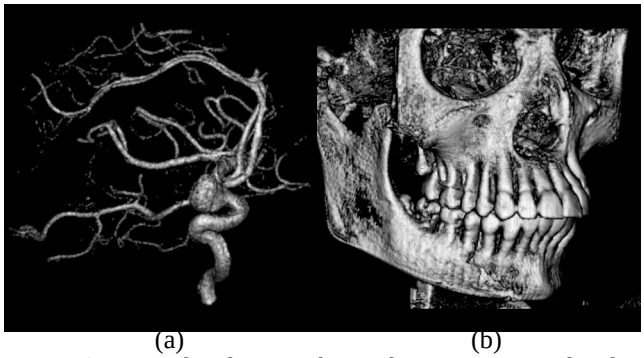


(a) (b)

Figure 5: (a) The rendering time using a 3-slab setup with 3, 5, 9, 13 and 16 nodes, respectively. (b) The rendering time using a 5-slab setup with 3, 5, 9, 13 and 16 nodes.

Our algorithm is designed to render data sets that have exploitable occlusion. The UNC Head data set is one typical such example. Our analysis in the previous sections shows a picture of how data sets of this category would perform. On the other hand, one might be interested in the performance of our algorithm when applied to data sets that do not exactly fall into this category.

We present the speed-up results of two other data sets, a blood vessel data set (Fig. 6a) and a human skull data set (Fig. 6b). The resolution of both is $256 \times 256 \times 256$ voxels.



(a) (b)

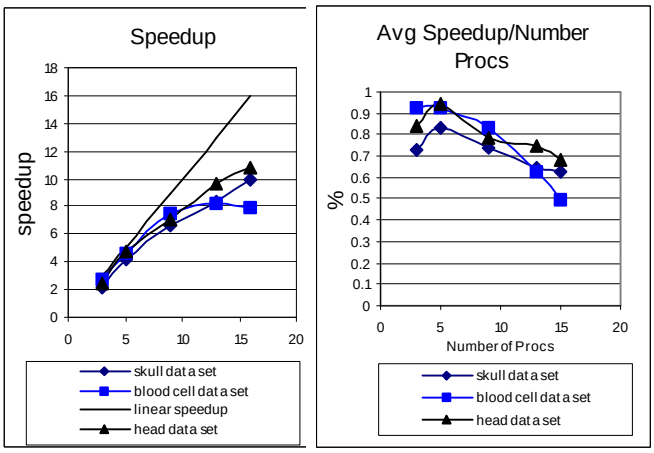
Figure 6: Two other data sets that we have experimented with. (a) Blood vessels extracted from CT scan of human brain. (b) A skull captured by MRI.

The blood vessel data set is branched in nature. There is not much exploitable occlusion. The skull data set has some portions of it showing strong occlusion, but not all portions of the data set have this feature. We hope to demonstrate the applicability of our algorithm in the three different scenarios represented by the three data sets, UNC Head, Skull and Blood Vessel.

During an 18 frame sequence using 5 degree rotations per frame, we record the actual rendering times for the 3-slab setup, with 3, 5, 9, 13 and 16 processors. We compute the average speed up, shown in Fig. 7a, for each frame except the initial one. Results for all three data sets have been collected.

In case of the UNC Head, when using 3 nodes, where only 2 nodes are used for rendering, we achieve a speed-up factor of 2.5. For 5 nodes, 4 rendering, we achieve a speed-up of 4.4. Here, the master node is doing some work, but it is only scarcely utilized. The high speed-up is primarily due to better caching performance. The PII PCs that we have on our cluster only have 128KBytes level 2 cache, 16KBytes on-chip cache. But our software based approach implements a full frame buffer in main memory. The frame buffer implements 32-bit depth color at 512 by 512 image resolution. Caching performance is much improved when a rendering node strictly manages a considerably smaller macrotile of the frame buffer.

Although the master node is only scarcely used, and should be considered a separate system dedicated to large-scale data management, we still include it in our study of CPU utilization. In Fig. 7b, we take the speed-up numbers and divide them by the total number of CPU's used. For the case of 3-CPU, the utilization is lower, due to the primarily idle Master node. For our implementation, the more processors used, the smaller the size of the macro-tiles becomes. As discussed, the overhead of the overlapping bricks increases as the macrotile size decreases, while the average utilization decreases to about 67% with 16 nodes.



(a) (b)

Figure 7: (a) The average speed up using a 3-slab setup with 3, 5, 9, 13 and 16 CPUs, during the 18 frame-sequence. (b) The ratio of the average speed up over the number of processors used, using a 3-slab setup.

For the Skull data set, although the occlusion is less prevalent than that of the UNC Head, similar results have been obtained. The compromise in performance is tractable. Using 16 nodes, the average utilization is still around 63%. However, with the Blood Vessel data set, the scalability is questionable. The system scales reasonably well below 10 nodes, beyond which scalability is poor. The reason is that for such data sets that show little occlusion characteristics, occlusion culling contributes little in speed up. The barriers introduced between slabs hurt performance significantly, when the number of nodes to meet at the barrier is large. Using less than 10 nodes, the macrotiles are relatively large, and the workload distributes evenly. When more nodes are being utilized, smaller macrotiles are used, load balancing becomes problematic. For such data sets, actually in sequential mode, occlusion culling does not speed up rendering much either. Parallel rendering targeted at such data sets would achieve more scalable speedup results with a more

straightforward parallelization that does not incorporate occlusion culling.

6. Conclusions & Future Work

This paper presents an innovative approach to parallelizing IASB splatting with occlusion culling, designed for efficient parallel rendering of data sets (transfer functions) that show heavy occlusion. The volume data is organized as object space bricks. The tight bounding box of the sparse voxel list within each brick facilitates fast and simple schemes for both brick level data culling and workload estimation. During rendering, the concept of image-aligned slab is leveraged to cull down the amount of data communication and data storage on the rendering nodes. Coupled with view coherence, the image-aligned slab representation reduces the data communication needed to two orders of magnitude lower than the storage of the raw volume itself. Broadcast operations are used such that one optimized collective communication operation distributes data to all parties in need. The duplicated non-occluded data storage on all rendering nodes supports a flexible load balancing scheme, which utilizes a partial screen-space quadtree.

There are two possible underlying hardware systems, a non-symmetric one or a symmetric one. We currently use a single master node for data communication, task management and scheduling. On a non-symmetric type of system, where a powerful master node has fast access to the disk storage, our prototype system would be able to accommodate large data sets. But on a symmetric system, such as SGI Origin 2000 and most NOW clusters, the current implementation of ours is not scalable to large data sets. The data management task should be distributed to each node on the cluster, with each node possessing a portion of the volume, and replacing the *Bcast* operation for data update operations with *All-Gather*. In either case, the rendering algorithm is independent from the data serving approach and stays the same.

For data sets that have a reasonable amount of occlusion to be exploited, our system shows good scalability. With a 16-node Pentium II 300MHz cluster, 63% to 67% CPU utilization is achieved. But our algorithm is not designed for data sets that exhibit little occlusion characteristics. Such data sets are more efficiently rendered with a more straightforward parallel algorithm of the IASB splatting without occlusion culling.

7. Acknowledgment

This project is funded by the DOE ASCI program and a NSF Career Award received by Dr. Roger Crawfis. The NOW cluster that we used are maintained by Dr. Dhabaleswar Panda and Dr. Ponnuswamy Sadayappan's group at the Ohio State University. We deeply appreciate the fruitful discussions that we have had with Dr. Han-wei Shen. The anonymous reviewers' comments must also be acknowledged for their valuable suggestions. The blood vessel and the human skull data sets are provided to us by Mr. Michael Meissner at University of Tuebingen, Germany.

REFERENCES

[1] Amin, M., Grama, A., singh, V., "Fast Volume Rendering Using an Efficient, Scalable Parallel Formulation of the Shear-Warp Algorithm", Proc. Parallel Rendering Symposium, Atlanta, GA, 1995, pp. 7-14.

[2] Anderson, T., Culler, D., and Patterson, D., "A Case For Networks Of Workstations (NOW)", *IEEE Micro*, 1995, pp. 54-64

[3] Becker, D. J., Sterling, T., Savarese, D., Dorband, J. E., Ranawak, U. A., and Packer, C. V., "BEOWULF: A PARALLEL WORKSTATION FOR SCIENTIFIC COMPUTATION", *Proc. International Conf. on Parallel Processing*, 1995.

[4] Challinger, J., "Scalable Parallel Volume Raycasting for Nonrectilinear Computational Grids", Proc. Parallel Rendering Symposium, San Jose, CA, 1993, pp. 81-88.

[5] Corrie, B., Mackerras, P., "Parallel Volume Rendering and Data Coherence", Proc. Parallel Rendering Symposium, San Jose, CA, 1993, pp. 23-26.

[6] Elvins, T., "Volume Rendering on a Distributed Memory Parallel Computer", Proc. Parallel Rendering Symposium, San Jose, CA, 1993, pp. 93-98.

[7] Huang, J., Mueller, K., Shareef, N., Crawfis, R., FastSplats: Optimized Splatting on Rectilinear Grids, Proc. IEEE Conference on Visualization, Salt Lake City, Utah, 2000.

[8] Hsu, W., "Segmented Ray Casting for Data Parallel Volume Rendering", Proc. Parallel Rendering Symposium, San Jose, CA, 1993, pp. 93-98.

[9] Lacroute, P., Levoy, M., "Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation", Proc. SIGGRAPH'94, pp. 451 - 458.

[10] Lacroute, P., "Analysis of a Parallel Volume Rendering System Based on the Shear-Warp Factorization", IEEE Trans. of Visualization and Computer Graphics, September, 96, Vol. 2, No. 3, pp. 218 - 231.

[11] Li, P., Whitman, S., Mendoza, R., Tsiao, J., "Prefix -- A Parallel Splattting Volume Rendering System for Distributed Visualization", Proc. Parallel Rendering Symposium, Phoenix, AZ, 1997.

[12] Ma, K., "Parallel Volume Ray-Casting for Unstructured-Grid Data on Distributed- Memory Architectures", Proc. Parallel Rendering Symposium, Atlanta, GA, 1995, pp. 23-30.

[13] Ma, K., Crockett, T., "A Scalable Parallel Cell-Projection Volume Rendering Algorithm for Three-Dimensional Unstructured Data", Proc. Parallel Rendering Symposium, Phoenix, AZ, 1997.

[14] Machiraju, R., and Yagel, R., "Efficient Feed-Forward Volume Rendering Techniques For Vector And Parallel Processors," *SUPERCOMPUTING '93*, 1993, pp. 699-708.

[15] Montani, C., Perego, R., Scopigno, R., "Parallel Volume Visualization on a Hypercube Architecture", Proc. Volume Visualization Symposium, Boston, MA, 1992, pp. 9 - 16.

[16] Mueller K. and Crawfis, R., "Eliminating Popping Artifacts in Sheet Buffer-based Splatting," *Proc. IEEE Vis.'98*, Research Triangle Park, NC, pp. 239-245, 1998.

[17] Mueller, K., Moeller, T., Swan, J.E., Crawfis, R., Shareef, N., Yagel, R., "Splatting Errors and Anti-aliasing," *IEEE TVCG*, vol. 4., no. 2, pp. 178-191, 1998.

[18] Muller, K., Shareef, N., Huang, J., Crawfis, R., "High-Quality Splatting on Rectilinear Grids with Efficient Culling of Occluded Voxels," *IEEE TVCG*, June, 1999.

[19] Neumann, U., Communication Costs for Parallel Volume-Rendering Algorithms, IEEE Computer Graphics and Applications, Vol. 14, No. 4, July 1994, pages 49-58

[20] Nieh, J., Levoy, M., "Volume Rendering on Scalable Shared-Memory MIMD Architectures", Proc. Volume Visualization Symposium, Boston, MA, 1992, pp. 17-24.

[21] Parker, S., Shirley, P., Livnat, Y., Hansen, C., Sloan, P., "Interactive Ray Tracing for Isosurface Rendering", Proc. IEEE Visualization'98, Durham, NC.,1998.

[22] Sano, K., Kitajima, H., Kobayashi, H., Nakamura, T., "Parallel Processing of the Shear-Warp Factorization with the Binary-Swap Method on a Distributed-Memory Multiprocessor System", Proc. Parallel Rendering Symposium, Phoenix, AZ, 1997.

[23] Westover, L.A., "Interactive Volume Rendering," *Proc. of Volume Vis. Workshop*, Chapel Hill, N.C., pp. 9-16, 1989.

[24] Westover, L.A., "SPLATTING: A Parallel, Feed-Forward Volume Rendering Algorithm," *Ph.D. Dissertation Department of Computer Science, The University of North Carolina at Chapel Hill*, 1991.

[25] Whitman, S., "A Task Adaptive Parallel Graphics Renderer", Proc. Parallel Rendering Symposium, San Jose, CA, 1993, pp. 27-34.