Dynamic Load Balancing for Raytraced Volume Rendering on Distributed Memory Machines[*]

Sanjay Goil and Sanjay Ranka
School of CIS ond NPAC
Syracuse University, Syracuse, NY, 13244-4100
`sgoil,ranka@top.cis.syr.edu`

Abstract

We present a technique for adaptive load balancing for ray traced volume rendering on distributed memory machines using hierarchical representation of volume data. Our approach partitions the image onto processors while preserving scanline coherence. Volume data is assumed replicated on each processor since our focus in this paper is to characterize computation and communication requirements for performing dynamic load balancing. We show that communication overheads are negligible to perform dynamic load balancing while rendering a sequence of frames on a distributed memory implementation. By exploiting image and frame coherence while distributing the image space, load balancing can be achieved at a relatively low cost.

1 Introduction

Ray traced volume rendering methods are very computationally intensive which make them slow for interactive rendering. The goal of achieving interactive volume rendering rates has led to several optimizations in use of data structures for reducing computation which include hierarchical spatial enumeration of volume data, early ray termination and adaptive ray firing techniques. To further improve on these optimizations parallel algorithms have been used to accelerate the process. Most such implementations distribute image and object data statically over processors on a one time basis. Each processor generates an image from the data assigned to it and in a final phase combines the partial image into the final result. To achieve an efficient implementation an equitable distribution of work load over processors is necessary. This is not always possible to achieve, especially for dynamically changing structure of computation. Rays traverse through volume data compositing voxel data into color and opacity of that pixel. These applications are usually efficiently represented and manipulated by using sparse data structures such as graphs, trees, and lists in sequential algorithms to reduce problem size as well as gain asymptotic performance. The communication networks and software available on coarse-grained machines make local accesses at least an order of magnitude faster then nonlocal accesses. This is further accentuated by high latency costs of communication software on distributed-memory machines. Effective parallelization of these applications on coarse-grained MIMD machines requires careful attention for the following two reasons: Firstly, the amount of work done by the parallel algorithm should be within a small constant factor of the amount of work done by the sequential algorithm. Secondly, for many applications, the data structures used have inherent locality of access and/or change incrementally. Exploitation of this information is necessary for efficient use of the various levels of memory hierarchy present in these architectures (register, caches, local accesses, nonlocal accesses, etc.). This requires fast methods for partitioning, repartitioning, replication, and migration of data. In this paper we show that dynamic load balancing can be done at a relatively low cost but owing to image and object coherence partitioning the image space once works well in practical situations. Volume rendering and related work is discussed in Section 2. We discuss aspects of data coherence that are needed for data partitioning in Section 3. The algorithm used for dynamic image space partitioning is presented in Section 4. Section 5 presents the performance results.

[*]This work was supported in part by NASA under subcontract #1057L0013-94 issued by the LANL, NSF under ASC-9213821 and AFMC and ARPA under contract #F19628-94-C-0057. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

2 Volume Rendering

Volume rendering is a technique for visualizing sampled scalar or vector fields of three spatial dimensions without fitting geometric primitives to the data. Since all voxels (volume elements) participate in the generation of each image, rendering time grows linearly with the size of the data set. The principal advantages of these techniques over others are their superior image quality and the ability to generate images without explicitly defining surface geometry. Figure 1 gives an overview of the volume rendering algorithm.

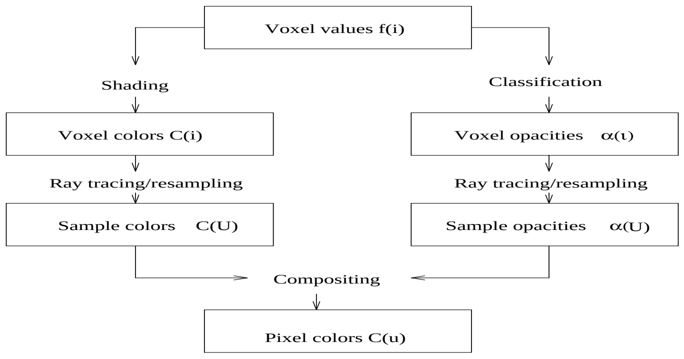

Figure 1: Overview of volume-rendering algorithm

Many datasets contain coherent regions of empty voxels. A voxel is defined as empty if its opacity is zero. Methods for encoding coherence in volume data include octree hierarchical spatial enumeration, polygonal representation of bounding surfaces and octree representation of bounding surfaces. For a dataset measuring N voxels on a side where $N = 2^M + 1$ for some integer M, the hierarchical spatial enumeration can be represented by a pyramid of $M+1$ binary volumes. Adaptive termination is implemented by stopping each ray when its opacity reaches a user-selected threshold level. We use the shear-warped algorithm described by Lacroute [4]. The algorithm given as

1. Transform the volume data to sheared object space by translating and resampling each slice according to S.
2. Composite the resampled slices together in front to back order.
3. Transform the intermediate image to image space by warping it according to M_{warp}.

It is formalized and written as the factorization of the view transformation matrix M_{view} as follows:

$$M_{view} = P \cdot S \cdot M_{warp}$$

where P is the permutation matrix which transposes the coordinate system in order to make the $z-$axis the principal viewing axis. S transforms the volume into sheared object space and M_{warp} transforms sheared object coordinates into image coordinates.

Table 1 summarizes the various approaches that have been reported in the literature for volume rendering on parallel machines.

3 Data Partitioning

Parallelization strategies for volume rendering have two goals. Each processor needs to be assigned equal load and any mapping of data to processors needs to maintain locality. The former helps to reduce processor idle time and the latter helps in keeping overheads of communication low. These are referred to as *load balancing* and maintaining *data locality*, two often conflicting goals. We discuss each of these to motivate the approach we have taken to analyze requirements for each of the two goals. Strategies used for data partitioning are classified as follows.

1. **Image Space Partitioning:** The pixels of an image are distributed across processors. Each processor traces rays for the pixels assigned to it. The volume data is replicated on each processor. Portions of the image from each processor are then combined to yield the final images. This method achieves near linear speedup but is not feasible if the object data set is larger than the available memory on each node.
2. **Object Space Partitioning:** The volume data is partitioned and distributed among processors. Each processor traces each ray in the local partition only. Each non-resolved ray is transmitted to the next processor for further tracing. Once each ray has finished the final composited values are collected to form the final image.
3. **Object Dataflow:** A partition of the image is assigned to each processor, which locally traces and resolves each assigned ray. Volume data is partitioned among nodes too. Non-resolved rays will be sent to appropriate processors for tracing and the "owner" of the ray will get the finished result back.
4. **Image/Object Partitioning:** The volume data is partitioned among processors. The image data is also partitioned among processors. Each processor is responsible to trace rays from pixels assigned to it. Pixels may be traced in the local

Approach	Target Architecture	Description
Montani et al. (1992)[6]	nCUBE	Hybrid image partitioning - ray dataflow approach. Processing nodes organized as a set of *clusters*. Image space is partitioned, Volume data is replicated on each cluster. Static load balancing is used for distributing data.
Nieh (1992) [7]	Stanford DASH	Data interleaved among processor memories. Image partitioned into contiguous blocks for assignment to processors. Task Stealing is used for dynamic load balancing.
Schröder & Stoll (1992) [8] Vezina et al. (1992)	CM-2 MP-1	Data parallel SIMD implementation, rays proceed in lock step Also SIMD with volume transposition to localize data access
Ma, Painter et al. (1993) [3]	CM-5	static input data partitioning into subvolumes using a k-D tree Processing nodes perform local raytracing of their subvolume concurrently.
Karia (1994)[2]	Fujitsu AP1000	Data is decomposed into subvolumes and rendered locally on each processor Scattered decomposition is used for load balancing.

Table 1: Different approaches on parallel volume rendering

volume data that is in the processors memory or it might fetch data that it needs from other processors.

In this paper we have taken the image partitioning approach to highlight the requirements for dynamic load balancing. The only form of dynamic load balancing reported in the literature is performed by *task stealing* on a shared memory machine. We report results for a dynamic load balancing scheme on the CM-5, a distributed memory machine. Communication costs are typically higher than computation costs on most real machines. To keep communication costs down, various forms of data locality need to be exploited. There are three kinds of coherence in images

1. **Image Coherence:** Image coherence is the property that adjacent pixels of an image are illuminated in a similar way. Portions of the image are similar in nearby areas, and this fact can be exploited when allocating pixels to processors. Nearby pixels go to same processors. This coherence exists in two dimensions. Exploiting this property for load balancing is not as straight forward. In an irregular image, where some portions are bright and some are dark, the work done in compositing a ray can vary a lot.
2. **Object Coherence:** Adjacent rays will traverse similar portions of the object. In hierarchical volume representations, ray intersections with the octree can be optimized for adjacent rays. This helps in reducing communication costs for essential volume data on processors for ray tracing by allowing reuse of offprocessor data. The data can be incrementally modified as the previous data can usually be reused.
3. **Frame Coherence:** Data accesses in consecutive frames in a multiframe sequence are quite similar. Figure 2 shows frame coherence for the *brain* dataset. This fact is used in the dynamic load balancing strategy that we propose in this paper.

Data partitioning for the image space needs to provide image coherence. By proceeding scanline by scanline within a slice, scanline coherence is exploited. To preserve this, image partitioning is done by dividing the scanlines to processors, keeping the load balanced. A record of the load on each processor is kept which is used to partition scanlines in the next frame. The first frame is load-balanced by looking at the load of scanlines of each slice on the processor. This strategy works well with a replicated object. For distributed volume, data partitioning the image and the volume are complimentary. Allocation of rays to processors needs to be guided by the volume data that is assigned to a processor. Two dimensional locality needs to be exploited for maximum reuse of data on processors to keep the communication costs low.

4 Dynamic Image Partitioning

Complex images can be non-uniform and hence can require varying amounts of compositing work in different regions of the image. Parallel implementations have to deal with this and try to distribute portions of the image to processors so as to allocate nearly equal work to each processor. Moreover, for a number of frames in an animation sequence, a change in the viewing angle might lead to changing workloads. What was a equitable distribution might no longer be so.

Scanlines are allocated to processors for tracing through volume data. We assume that volume data and its min-max octree is replicated in local memories of each of the processors. A measure of work done by each scanline is taken. This is calculated by factoring

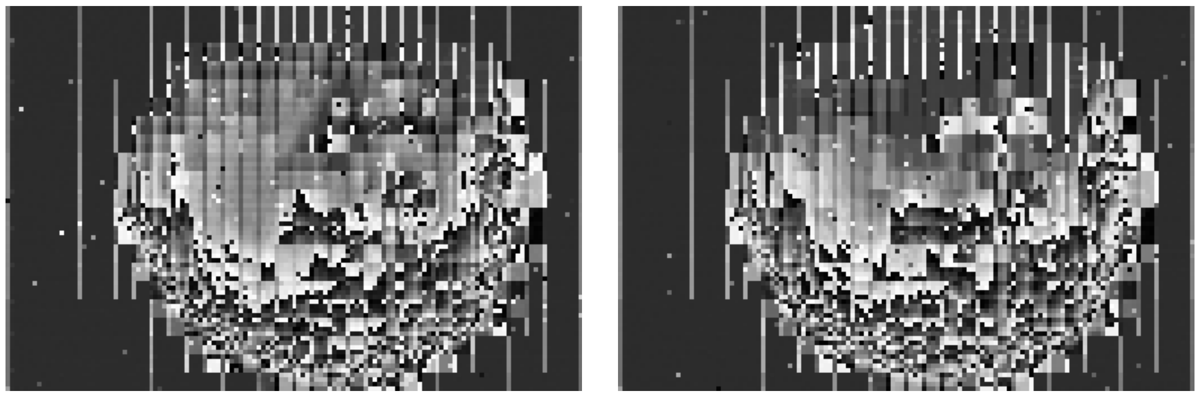
Figure 2: Frame 1 and Frame 2 (At 5^o rotation from FRAME 1): Each pixel shows the computation work of a scanline. Scanlines for a slice progress from left to right on the horizontal axis. Slices of a frame are top to bottom on the vertical axis.

in the complexity of the tree traversal performed for each portion of the scanline to estimate the data for the compositing process. Work assignment is done in terms of number of scanlines allocated to each processor. Dynamic load balancing is performed at intervals when the idle time of any processor crosses a certain threshold value. Figure 3 describes our algorithm.

5 Results

We perform our experiments on two datasets "brainsmall", a 128 X 128 X 83 voxel set from the MRI scan of a human head and "headsmall", a 128 X 128 X 113 voxel data set of the CT scan of a human head. Due to limited memory on each processor of our CM-5 we are not able to run bigger data sets. Although, these results carry more meaning on larger volumes, the idea here is to present the concepts for providing a framework for dynamic load balancing. The target architecture in this paper is the Connection machine (CM-5), a distributed memory multiprocessor.

Load Balancing within a frame Table 4 and Table 5 compare the rendering performance between the three strategies we have compared in this paper. In the case of a static image partitioning the scanlines are allocated at the start of the rendering process. Since there is no estimate of work available yet, equal number of scanlines are allocated to each processor in scanline order. To perform load balancing on a frame when no previous workload information is available, as in this case, a technique for balancing load across slices of the frame can be used. Table 2 shows the cost of repartitioning scanlines within a frame. This requires not only a recalculation of scanline allocation to processors but also the movement of partial opacities calculated thusfar between processors. A scanline may move from one processor to another within a frame and the color, opacity and other parameters of each pixel in the scanline must be moved along with it. If the amount of scanline movement is large the overhead of such a repartitioning will not benefit us. Each pixel is represented by 12 bytes of data, which makes it $12 \times Imagewidth$ bytes for a scanline (12×128) in our case. Further, due to object coherence, scanline reallocation occurs only at processor boundaries for nearby slices. Repartitioning, hence need not be performed for every slice as the load on processors does not create imbalance as to warrant reallocation.

A *sender-directed* communication protocol is used in which each processor can figure out the scanlines it needs to send to a destination processor. Similarly, at the receiving end, the processor is able to calculate the amount of data and the image offset it needs to receive data at. Owing to *slice coherence* a repartitioning scheme every 20 slices for the sample data sets performs well for load balancing. Since the sample data sets in this paper are small the cost of repartitioning during a frame are high to offset any gain in load balance. Hence we use a static allocation scheme for the first frame to report our results on dynamic load balancing at each frame. This performs considerably better than the pure static partitioning scheme since it relies on processor load information acquired during the rendering of the first frame. This might not work though if data access patterns differ substantially from one frame to another. The load estimate of the first frame might not hold true for later frames if discrete view angles are considered.

```
First_Frame = True
For each Frame
    if(First_Frame) {
        For each slice in volume
            Composite slice and collect load data
            Global Synchronization for finding Min and Max Load for slice
            Calculate Idle time of processors
            if(Idle Time > threshold1) {
                Repartition scanlines among processors
                Transfer Opacity and color gathered for pixels in scanlines to new processor.
                Accumulate load of each scanline.
            }
        First_Frame = False
    }
    Global Synchronization to gather each Min and Max for frame
    Calculate Idle time of processors
    if(Idle Time > threshold2)
        Repartition scanlines among processors
```

Figure 3: Dynamic load balancing by repartitioning scanlines on processors

Image	Processors				
	2	4	8	16	32
brainsmall	17.15	13.85	10.11	10.61	6.63
headsmall	22.10	17.87	13.31	9.25	10.61

Table 2: Average cost of intraframe partitioning: partitioning between slices in a frame including partial opacity updates (Time in milliseconds)

No. of Scanlines	Processors				
	2	4	8	16	32
128	4.03	4.04	4.03	4.05	4.05

Table 3: Average cost of interframe partitioning: Calculating new allocation of scanlines for each frame (Time in milliseconds)

Load Balancing between frames Once the rendering work for an image is known for a frame, it can be used to determine a distribution of scanlines for subsequent frames. We use the previous frame's profile to estimate scanline distribution for a new frame. Figures 2 highlights frame coherence present in images, and hence motivate us to use workload information from one frame to another. Results for this are also presented in Table 4 and Table 5. The cost of repartitioning scanlines at each frame boundary is shown in Table 3. A reduction is performed on the workload array of each processor of the previous frame. Since each processor contains the work done by the scanlines allocated to it, the result of this operation is to provide a global view of the work profile on each processor. A prefix scan on this array, performed locally, is used to calculate a processors new set of scanlines by allocating itself close to average work.

Figure 4 plots the speedup we achieve using our dynamic load balancing strategy. The graph ignores the cost of partitioning the first frame since it is a one time cost and will not be a significant factor while rendering over many frames in an animation sequence.

Processors		Frames				
		1*	2	3	4	5
2	I	977	989	977	967	965
	II	940	817	820	802	804
	III	940	804	811	796	801
4	I	500	501	494	488	489
	II	500	404	404	400	401
	III	500	387	385	384	384
8	I	344	338	332	332	328
	II	344	232	231	229	227
	III	344	232	216	230	225
16	I	216	215	212	211	210
	II	216	126	126	125	125
	III	216	125	111	111	114
32	I	137	137	138	137	136
	II	137	92	94	93	93
	III	137	90	100	82	105

Table 4: Rendering time for 5 consecutive frames rendered at 5^o rotation (clockwise) of **brainsmall** for (I) Static partitioning (II) Partition-once for first frame (III) Dynamic load balancing at every frame (Time in milliseconds). (*) Static partitioning is used for the first frame in each case

6 Conclusions

We have presented an approach for dynamic load balancing by exploiting image and frame coherence in volume rendering. A replicated object has been considered since the main objective of these experiments is to show that the dynamic load balancing on distributed memory machines is possible by incurring a reasonable overhead in terms of communication cost. Our experiments on intraframe partitioning resulted in an added overhead due to movement of partial opacity of pixels when a scanline is reallocated to another processor. However with a faster communication medium this cost can be brought down to make dynamic intraframe partitioning feasible. A static assignment for the first frame and dynamic load balancing between subsequent frames works considerably well as shown by our results. We are currently pursuing a more practical and scalable approach of extending these techniques for a distributed volume data.

Processors		Frames				
		1*	2	3	4	5
2	I	2730	2708	2757	2791	2810
	II	2730	2155	2165	2163	2167
	III	2730	2118	2158	2126	2134
4	I	1437	1468	1513	1529	1528
	II	1437	1057	1099	1117	1111
	III	1437	1053	1016	1070	1099
8	I	802	801	834	852	846
	II	802	586	553	548	550
	III	802	576	546	534	546
16	I	396	398	411	427	423
	II	396	311	322	290	326
	III	396	299	310	285	308
32	I	218	219	219	221	220
	II	218	206	185	195	194
	III	218	200	187	192	196

Table 5: Rendering time for 5 consecutive frames rendered at 5^o rotation (clockwise) of **headsmall** for (I) Static partitioning (II) Partition-once for first frame (III) Dynamic load balancing at every frame (Time in milliseconds). (*) Static partitioning is used for the first frame in each case

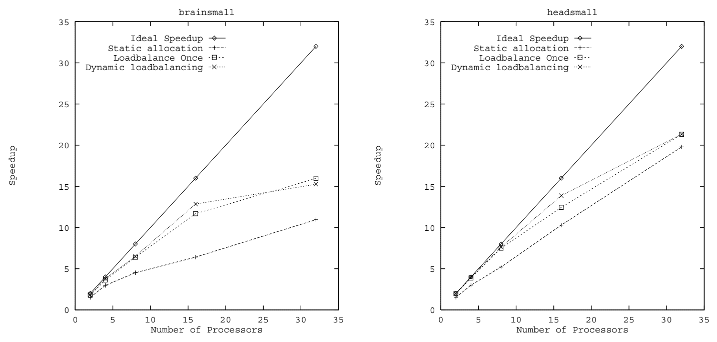

Figure 4: Speedup achieved for the three strategies discussed in this paper for the datasets brainsmall and headsmall

References

[1] Goil, S., Primitives for problems using hierarchical algorithms on distributed memory machines, *The First International Workshop in Parallel Processing*, Bangalore, India, December 1994.

[2] Karia, R., Load balancing of Parallel Volume Rendering with Scattered Decomposition, *Technical Report, Dept. of Computer Science*, Australian National University, Canberra, Australia.

[3] Ma, K., Painter. J. S., Hansen, C. D., Krogh, M. F., A Distributed Parallel Algorithm for Ray Traced Volume Rendering, *Parallel Rendering Symposium*, October 1993.

[4] Lacroute, P. and Levoy, M., Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation, *SIGGRAPH'94*, August 1994, Florida.

[5] Levoy, M., Efficient Ray Tracing of Volume Data, *ACM Transactions on Graphics*, Vol. 9, No. 3, pp 245-261, July 1990.

[6] Montani, C., Perego, R., Scopigno, R., Parallel Volume Visualization on a Hypercube Architecture, *Proceedings of the Boston Workshop on Volume Visualization*, Boston, October 1992.

[7] Nieh, J., Levoy, M., Volume Rendering on Scalable Shared-Memory MIMD Architectures *Proceedings of the Boston Workshop on Volume Visualization*, Boston, October 1992.

[8] Schröder, P. and Stoll, G., Data parallel volume rendering as line drawing, *Proceedings of the Boston Workshop on Volume Visualization*, pp 25-32, Boston, October 1992.