Order of Pixel Traversal and Parallel Volume Ray-tracing on the Distributed Shared Volume Buffer

Hansong Zhang, Shenquan Liu *
CAD Lab., Institute of Computing Technology
Academia Sinica, P. O. Box 2704, Beijing 100080, P. R. China
zhs@osf.cnc.ac.cn

Abstract. The distributed shared volume buffer (DSVB) is a software package we developed to facilitate general, parallel volume ray-tracing on networked workstations. It is internally implemented with message-passing and adopts the cache-coherent shared memory model. Thus the cache efficiency of volume data access is of utter importance to the performance of a DSVB-based ray-tracer. For a given data set, the data access behavior of a volume ray-tracer depends mostly on the way in which pixels of the image are traversed. This paper addresses the cache coherence problem and compares three kinds of pixel traversal order: one-way, two-way and along a space filling curve. Experiments show that traversing pixels along a space filling curve (e.g. a Hilbert curve) greatly enhances cache efficiency especially when size of the cache is small compared to that of the volume data, and in the meantime greatly simplifies task distribution and management.

Keywords: Pixel Traversal, Parallel Volume Ray-tracing

1 Introduction

Raytracing volume data can produce impressive images that help a lot in understanding the data [10, 11]. The cost of ray-tracing, however, is quite expensive both in time and in memory space. A high-quality image may take as much as hours to render. Volume data sets are typically large and get easily too large to hold in the main memory of an average workstation. To solve these problems, data distributed and parallel volume ray-tracing algorithm have been developed for modern parallel computers. A network of workstations, which is logically equal to an MIMD, distributed memory, message-passing based multicomputer, provides a cost-effective and powerful environment for parallel volume rendeing.

The great potential of multicomputers and networked workstations must be fully tapped by proper parallel applications. There are plenty of existing serial

*This work was supported by National Natural Science Foundation of China.

volume ray-tracing programs. Unfortunately, however, these serial ray-tracers cannot be easily ported to a distributed-memory parallel environment when the data are distributed among the processing nodes. The direct access of volume data in a continuous, flat address space, which is implicitly assumed in any serial ray-tracer, is not readily available in such an environment. Replicating the whole data in every node's local memory will elliminate the problem, but due to the limited size of the local memory this is not a desireable approach. So the major question for data distributed parallel ray-tracers is: what should I do if the data needed is not in the local memory? One solution is to distribute task in such a way that computation on a node need only local data [1, 3, 4, 5]. Or computation itself can be sent to the node where the requrired data reside [7]. All these approaches lead to highly specific algorithms that bear little resemblence to a serial algorithm. Besides, in general volume ray-tracing (in contrast to simple ray-casting), light behaviors like reflection, refraction and shadowing are taken into account. The ray through any pixel may access any data. In this case it is impossible to construct tasks that access only local data. Porting a serial ray-tracer to a shared memory multicomputer (like DASH)[2, 9], on the other hand, is direct and simply. Aside from task distribution, and the parallel ray-tracer is much the same as a serial one.

The distributed shared volume buffer (DSVB) is a software package we developed to provide a virtual shared memory of volume data for networked workstations. The volume data is physically distributed among machines (nodes) that participate in ray-tracing, but through the DSVB interface, the ray-tracer on each node can regard all data as local. DSVB incorporates the cache-coherent shared memory model [2]. Authors of [6] implemented a virtual memory based on software cache on Fujitu AF100, though in their case the nessesity of global data access arises from dynamic load balancing, but not from global illumination. Section 2 gives a breif description of DSVB.

A software cache is a buffer in the local memory for recently used data that are not local. If the requested non-local data are found in the cache, then a remote data fetching is spared. The efficiency of cache is crucial to the performance of DSVB, and to applications based on it. This is particularly true when the parallel environment is a group of workstations connected by moderate-performance network (e.g. 10Mb/s Ethernet), where communication cost is high. Besides the design of the cache itself, the way how volume data is accessed greatly affects cache efficiency. In a volume ray-tracer, the data access pattern is mostly decided by the order in which pixels are ray-traced. Section 3 and section 5 discuss the pixel traversal order problem and compare three kinds of order.

2 The Distributed Shared Volume Buffer

The structure of our cache-coherent shared volume buffer is shown in figure 1. The buffer is divided into "pages"; page is the unit of network transfer of volume data. The volume data does not lay linearly in the virtual buffer. Instead, it is stored in terms of blocks, with one block residing in one page. Each block contains a $K \times K \times K$ sub-set of data, where K is adjustable and is used to decide the size of a page. The original volume data is padded with zeros to make its X, Y, and Z dimensions multiples of K. Dividing the data into 3D blocks takes into account the coherence of data access by a ray traveling through the volume. In linear division of volume data most pages contain 2-D piece of data, among which very few can be used by a ray.

Applications based on DSVB directly request data values, and don't have to know about the internal organization of data. The most commonly used interface to DSVB is

```
VolVal(P, V)
```

where P is a position vector in 3D space and V is an array of 8 integers. On return, the function stores in V the eight values on the corners of the volume cell that surrounds P.

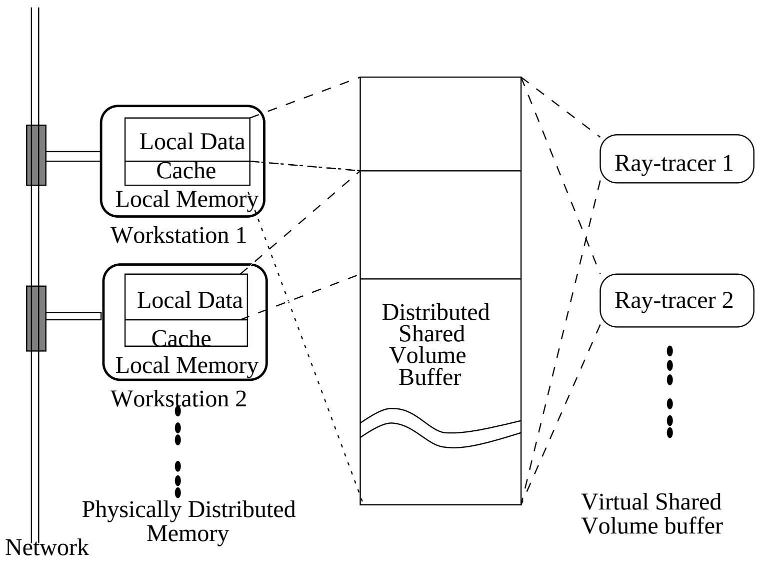

Figure 1: The distributed shared volume buffer

We call each of the networked workstations that take part in ray-tracing a node. When DSVB is initialized, every node gets a group of adjacent blocks of volume data. The data are distributed evenly and without special strategy. During computation, when on a node (say node A) the requested data is not in the local memory, DSVB looks for it in the cache. If it is in the cache (this is called a cache hit), then DSVB will return the cached value. If not, the the data have to be fetched from a remote node, and a situation similiar to a page fault in an operating system occurs. DSVB first finds out where the data reside as local (simple calculation, since the data blocks are distributed evenly), and send a page request to the node (say node B). Node B then send back the page requested by A. Node A, on getting the page, tries to allocate an item in the cache for it. If the cache no longer has free items, an LRU (Least Recently Used) algorithm is used to discard the content of an old item to make space. After the page is cached, DSVB return the requested data values, thus ending the data retrieving. Since a remote data request reqires a round trip on the network, it consumes considerable time. So cache efficiency (marked by the ratio of cache hits to total non-local data accesses, or amount of data tranferred because of cache misses) is critical for DSVB's performance.

3 Order of Pixel Traversal

Most serial ray-tracers pay little attention to the order of pixel traversal. They traverse the pixels on the image plane according to a "natural" order, e.g. row by row. For a DSVB based parallel volume ray-tracer, however, the order deserves a closer look, because it has great impact on the ray-tracer's data access coherence. High coherence means that recently accessed data, which are in the cache, are likely to be used again, thus reducing the times of remote data fetching through the network.

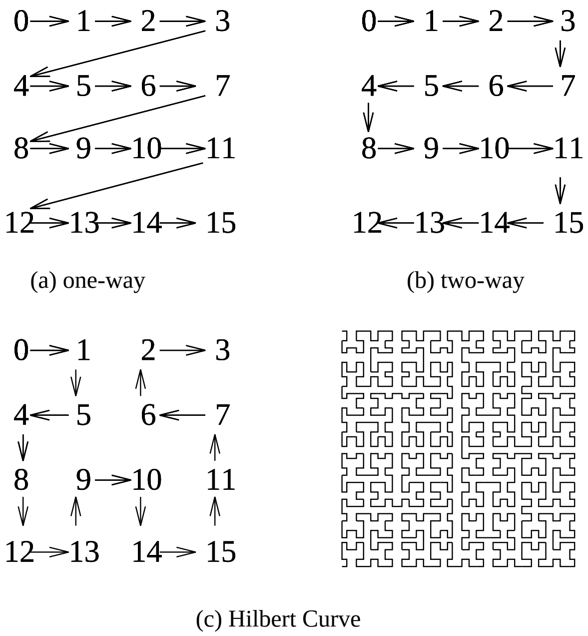

Figure 2: Pixel traversal order

As a matter of fact, rays through pixels that are close to each other access similiar data, or in other words, have high access coherence. So, the coherence of m pixels $\{P_1, ..., P_m\}$ can be measured by the weighted average distance between these pixels:

$$C = \sum p_{ij}|P_iP_j|, 1 \leq i < j \leq m \tag{1}$$

where $|\ P_iP_j\ |$ is the distance between P_i and P_j, $p_{i,j} \geq 0$, $\sum p_{ij} = 1$, $1 \leq i < j \leq m$. The coherence of a pixel traversal path for $n(n \geq m)$ pixels can be calculated by sliding an measure interval of m pixels along the path, and calculate C for each group of adjacent m pixels. This lead to a coherence function $C(s, m), s \in \{1, ..., n-m+1\}$, with s indicating the starting point of the measure interval. A good traversal path's $C(s, m)$ should have low flunctuation and small average.

Here we examine three kinds of pixel traversal order (see figure 2). The first is the "natural" order which goes through the pixels row by row, and we term it *one-way* in order to distinguish it form the second type of order. The second type also travels row by row, but takes inverse directions on adjacent rows. This we call the *two-way* order. The third way of traversal is to follow a space filling curve.

Let $I = [0, 1]$, a space filling curve is a continuous function $c(t)$ with domain I and range $I \times I$. When the mapping between I and $I \times I$ is 1-1, then the curve is simple. A simple space filling curve establishes a one to one correspondence between a point on interval $[0, 1]$ and a 2-D point in square $[0, 1] \times [0, 1]$. So, when t changes from 0 to 1, a simple space filling curve $c(t)$ visits uniquely every point in $I \times I$.

Here we consider only approximations of a simple space filling curve defined above. Often we call such an approximation itself a space filling curve. Let $I_n = \{\frac{1}{n}, \frac{1}{n-1}, ..1, n \in N\}$, an approxiamtion curve (a polyline) defines a 1-1 maps between $I_{n \times n}$ and $I_n \times I_n$. Let $N_n = \{1, \ldots, n, n \in N\}$, then after simple transformation the polyline maps $N_{n \times n}$ to $N_n \times N_n$. An image's domain is in fact the set of coodinates denoted by $N_n \times N_n$. So by traveling along the polyline approximation of a space filling curve, we traverse the pixels in an ordered and unique way. Intuitively, such a curve gives good coherence for any cluster of adjacent pixels on its path by the standard of (1) because of its self-entwined structure.

Visiting pixels along a space filling curve has been used in image processing [12], where typical operations occur on subregions of an image. By using a space filling curve, a 2-D subregion can be mapped to an 1-D interval. This reduction of dimensionality lead to simplification of image processing methods. [13] used space filling curves in digital halftoning of images.

4 Task Allocation

Our ray-tracer uses distributed task pools and task stealing paradigm for task allocation and dynamic load balancing. On start-up, every node obtains approximately the same number of pixels, which comprises the local task pool. If the local task pool of node K is exhausted, it asks node (K+1) *mod* n (n is the number of nodes) for more work. If this request fails, then node (K+2) *mod* n is asked, an so on. This results in a "food chain" among nodes. In this way computing load is balanced.

Obviously, it would lead to very poor data access coherence if we visit the entire image in the one-way or two-way order. For these two kinds of traversal, we first divide the image plane into small sub-blocks (e.g. 16×16 ones) of pixels, and use block as the unit of task allocation. The blocks are evenly distributed to nodes on start-up. The pixels in a block are visited in one-way or two-way order.

For tranversing along a Hilbert curve, task management is dramatically simpler: the division and distribution of an image are reduced to operations on an 1-D interval ($[1, n \times n]$ for an $n \times n$ image); a node's local task pool is as simple as two integers indicating the starting and ending number.

5 Results

We tested the three orders of pixel traversal on two scenes. One scene is a $256 \times 256 \times 109$ head surround by 4 mirrors (see figure 5). Another is a $128 \times 128 \times 197$ data set of animal trunk with reflections in 3 mirrors (figure 6). In the tests

$\frac{\text{cache size}}{\text{data size}}$	Data Transfer (Mbyte)			Rendering Time(sec.)
	one-way	two-way	Hilbert	Hilbert
%5	-	-	1108	6262
%10	281.6	240.1	132.7	226
%15	207.4	175.3	88.1	194
%20	159.4	145.9	73.1	171
%25	131.6	120.7	60.3	154
%30	126.7	113.8	56.9	150
%35	119.3	108.7	50.6	145
%40	113.9	96.5	47.3	145
%50	102.7	85.9	40.4	142
%60	80.5	75.3	36.1	138
%70	71.8	68.2	34.3	137
%80	61.4	60.0	34.3	132
%90	56.1	56.2	34.1	128

Table 1: Test results with the head data

we used 8 SGI Indigo workstations (with 50MHz R4000 CPUs) networked by 100Mb/s Ethernet. Test results are listed in table 1 and table 2, and illustrated in figure 3 and figure 4.

The impact of different orders on cache efficiency is shown by the amount of volume data transfer that is incurred by cache misses. Rendering time (excluding

cache size / data size	Data Transfer (Mbyte)			Rendering Time(sec.)
	one-way	two-way	Hilbert	Hilbert
%5	-	-	-	-
%10	-	-	-	-
%15	541.3	484.0	241.6	298
%20	214.4	185.8	74.4	127
%25	175.9	147.3	55.8	112
%30	139.7	120.0	48.0	107
%35	110.4	80.2	40.4	110
%40	86.9	64.5	35.6	99
%50	57.6	48.6	30.4	99
%60	43.1	39.1	27.9	93
%70	36.1	32.9	25.4	92
%80	30.8	28.5	23.8	88
%90	24.3	24.1	22.6	87

Table 2: Test results with the trunk data

time for data distribution) for Hilbert curve traversal is also given. The tests are performed on ordinary work days when all machines and the network are busy, whose load may vary considerably in the process of our tests. The serial version of our renderer, which directly accesses a volume data array, took 753 and 380 seconds to render the head and the trunk scene, respectively.

In these tests all other parameters are fixed except for the cache size. We use $32 \times 32 \times 32$ blocks, so the size of a page is 32K bytes. This is the amount of data transferred in a single remote data fetching. Under this block size, the head data is padded to $256 \times 256 \times 128$ ($8 \times 8 \times 4$ blocks), and the trunk to $128 \times 128 \times 224$ ($4 \times 4 \times 7$ blocks). Image block size (for one-way and two-way traversal) is 16×16. The image size is 512×512.

Results show that the two-way traversal is a bit more efficient than one-way, and Hilbert curve traversal gives significant performance boosts especially when size of the cache is small compared to the volume data. With Hilbert traversal and small cache size, data traffic drops steeply as the cache becomes larger, showing high cache efficiency. Our tests made use of 8 machines, and the volume data are evenly distributed among them, so when the cache size is larger than $\frac{7}{8} = 87.5\%$ of the data size, each node can accomodate all the volume data. In this case, the maximum amount is $M = (\frac{n-1}{n}D)n = (n-1)D$, where D is the size of the volume data and n the number of nodes. This occurs when all nodes accessed all pages. For the head data, $M \approx 56$Megabytes. From the test results we can see that the Hilbert order result in lower data transfer than M when size of the cache if greater than 30% that of the head data. This implies that the pixels traced by a node are well “clustered”, so that quite some non-local data are never accessed and thus never fetched. For the trunk tests, however, the

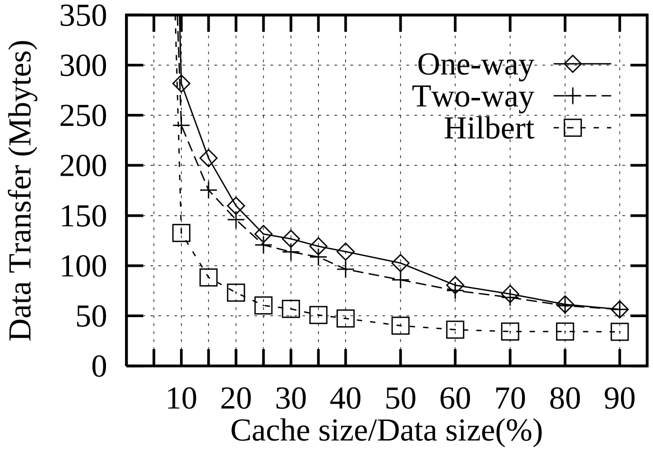

Figure 3: Amount of data transferred in tests with the head data.

data traffic at the high end of cache size, is not much less than M ($\approx$ 24.5Mb). This is because the setting of the trunk scene incurs more complicated reflection phenomenon than the head scene. In other words, the data are accessed more globally because of the reflections. Note that though the trunk data is smaller than the head, the data traffic in its tests is no less, sometimes even greater than tests with the head.

When the cache is too small, DSVB's performance drops dramatically because of *trashing*. Trashing means that the cache has to discard pages that are soon to be used again, thus giving rise to repeated remote data fetching. The missing data from table 1 and table 2 (marked by "-") indicate that trashing occured, and we have to abandon the tests either because the system manager and other users complained about network jams or we ourselves could not wait any longer.

6 Conclusion

In this paper we introduce the distributed shared volume buffer (DSVB) we developed to facilitate general parallel volume rendering. We discuss the relationship between the order in which a ray-tracer visits pixels and the cache efficiency of DSVB. Three types of traversal are tested: one-way, two-way and space filling curve. Experiments show that traversing along a space filling curve greatly enhances cache efficiency, and thus the performance of the ray-tracer.

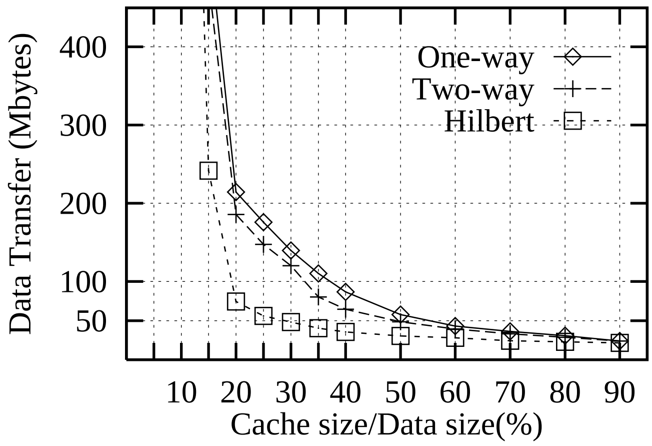

Figure 4: Amount of data transferred in tests with the trunk data.

References

[1] K.-L. Ma, J. S. Painter, C. D. Hansen and M. F. Krogh, Parallel Volume Rendering Using Binary-Swap Compositing, IEEE Computer Graphics and Applications, July 1994, pp. 59-68

[2] J. P. Singh, A. Gupta and M. Levoy, Parallel Visualization Algorithms: Performance and Architectural Implications, IEEE Computer, July 1994, pp. 59-55

[3] U. Neumann, Parallel Volume-Rendering Algorithm Performance on Mesh-Connected Multicomputers, Proc. Parallel Rendering Symp., ACM, New York, 1993, pp. 97-104

[4] W. M. Hsu, Segmented Ray Casting for Data Parallel Volume Rendering, Proc. Parallel Rendering Symp., ACM, New York, 1993, pp. 7-14

[5] E. Camahort and Chakravarty, Integrating Volume Data Analysis and Rendering on Distributed Memory Architectures, Proc. Parallel Rendering Symp., ACM, New York, 1993, pp. 23-26

[6] B. Corrie and P. Mackerras, Parallel Volume Rendering and Data Coherence, Proc. Parallel Rendering Symp., ACM, New York, 1993, pp23-26

[7] D. Badouel, K. Bouatouch and T. Priol, Distributing Data and Control for Ray Tracing in Parallel, IEEE Computer, July 1994, pp. 59-55

[8] G. Vézina, P. A. Fletcher and P. K. Robertson, Volume Rendering on the MasPar MP-1, 1992 Workshop on Volume Visualization, Boston, October 1992, pp. 3-8

[9] J. Nieh and M. Levoy, Volume Rendering on Scalable Shared-Memory MIMD Architectures, 1992 Workshop on Volume Visualization, Boston, October 1992, pp. 9-16

[10] M. Levoy, Efficient Ray Tracing of Volume Data, ACM Transactions on Graphics, July 1990, pp. 245-261

[11] R. A. Drebin, L. Carpenter and P. Hanrahan, Volume Rendering, Computer Graphics, vol. 22, August 1988, pp. 65-74

[12] R. J. Stevens, R.A.Lehar, F. H. Perston: Manipulation and Preservation of Multidimensional Image Data Using the Peano Scan, IEEE Transaction on Pattern Analysis and Machine Intelligence, May 1983, pp. 520-526

[13] L. Velho, J. Gomes, Digital Halftoning With Space Filling Curves, Computer Graphics, Vol. 25(4), July 1991, pp. 81-90