An Incremental Alignment Algorithm for Parallel Volume Rendering

Daniel Cohen-Or[1,2] and Shachar Fleishman[2]

[1] School of Mathematical Sciences
Tel-Aviv University, Ramat-Aviv 69978, Israel

[2] Department of Mathematics and Computer Science
Ben Gurion University, Beer Sheva 84015, Israel

Abstract
This paper introduces a data distribution scheme and an alignment algorithm for parallel volume rendering. The algorithm performs a single wrap-around shear transformation which requires only a regular inter-processor communication pattern. The alignment can be implemented incrementally consisting of short distance shifts, thus significantly reducing the communication overhead. The alignment process is a non-destructive transformation, consisting of a single non-scaling shear operation. This is a unique feature which provides the basis for the incremental algorithm.

Key Words: Volume Rendering, Parallel Algorithms, Parallel Rendering, Distributed Memory, Ray Casting.

1 Introduction

Volume rendering is a technique for visualizing 3D samples which are assumed to be partially transparent materials. Volume rendering applications require extensive computational resources to deal with the typical large sizes of the volumetric datasets. In order to achieve interactive visualization, parallel computation is inevitable. Several special-purpose architectures have been proposed [1,2], but most recent efforts have been directed towards implementing volume rendering on general-purpose parallel machines. These include distributed memory architectures [3,4,5,6] and shared memory architectures [7,8].

The scalability of a parallel algorithm in both number of processing units and data size is a major concern when evaluating parallel algorithms. One important factor that influences the performance of a parallel algorithm is the computation to communication ratio [9]. The larger that ratio, the more efficient is the algorithm and therefore scalable. A second consideration is that global communication operations, such as shifts, are much more efficient than point-to-point communications, because they avoid conflicts on the communication channels. When examining the communication overhead of a parallel algorithm, the volume of communication is not the only consideration since every communication operation includes some "start up" phase which yields an overhead that might be equivalent to the transfer of several thousands bytes of information. Therefore, reducing the number of communication steps might produce a significant speedup. Another factor for the efficiency of a parallel algorithm is the balanced distribution of work among the processing units.

The algorithm we are presenting in this paper accounts for these considerations. It requires a network topology of a ring, where each node have two communication channels to its left and right adjacent nodes, a topology which can be mapped on virtually every network topology that exists in today's parallel machines. The only communication pattern required by the algorithm is a global shift of data around the ring of the processing units. Moreover, the algorithm strives to shorten the length of the shift operations by an incremental technique. The algorithm includes a special solution for rotating the raster data across the 45 degree barrier.

2 Ray Casting on Parallel Architectures

One of the major techniques in volume rendering is ray casting. The rays sample the volume along the ray passage, evaluating opacity accumulation [10]. There are two prominent approaches to implementing ray casting on a parallel distributed memory scheme. The first approach is to follow the rays through the volume, moving the rays among the processing units, while the volume data remains in place. The second approach first transforms the data to be axis-aligned with the ray's direction. In a second phase, after the data has been localized, rays are sampled with no inter-processor communication. The first approach imposes the difficult task of maintaining a conflict-free access to the distributed data, since rays accesses arbitrary data. However, when casting parallel discrete rays using a 26-connected form, adjacent rays have the same pattern [11], hereafter referred to as the *template* property. This property permits simultaneous sampling by parallel rays while using regular inter-processor communications [5]. However, such methods are not symmetric and works well for angles in the range of $-45°$ and $45°$; otherwise the data has to be flipped (see [12]). Another method uses a memory skewing scheme that supports the extraction of an arbitrary ray in one parallel access [2,13].

The second approach has been implemented by applying a sequence of three one-dimensional shear transformations which are a decomposition of a 3D rotation [14]. The rotated data is aligned with the viewing direction so that the data accessed by a single ray requires either no communication or regular communication. Usually, the shear transformations include scale and re-sampling which modifies the dataset [15,6]. Thus, repeated transformations cause a non-reversible destruction of the original dataset. A *non-destructive* transformation is, of course, a very attractive feature. A non-destructive transformation is such that filtering is embedded in the rendering phase and no scaling is performed on the alignment phase, and there is no lose of data information. The shear transformation is a net shift of rows, therefor the dataset is is not modified and only re-mapped among the processing units and within their local memories.

The three-pass shear method is attractive for parallel implementation since a single, one-dimensional shear requires regular inter-processor communication or just an in-memory re-mapping [4,12]. However, a minor change of the viewing direction requires a complete rotation from the initial state. No incremental rotation algorithm is known. In our work we suggest moving the volume data in such a way that a processing unit holds data that is perpendicular to the viewing direction, using a single wrap-around shear in an operation we call *ray alignment*, described in Section 4(recently, a similar approach has been proposed [16]). Ray alignment is non-destructive and offers incremental rotations (Section 5). To reduce aliasing, we have developed a padding technique which allows an on-line re-sampling of the data during the ray sampling phase (Section 6).

It should be emphasised that this paper does not propose a volume rendering technique but a parallel memory organization scheme that provides a simplified mechanism for localizing the data for rendering in a way that reduces communication overhead. The image quality is independent of the proposed mechanism.

3 Terms and Definitions

We assume a MIMD architecture consisting of p symmetric *processing units* each of which consists of a CPU and a local memory. The processing units are interconnected in a ring topology or in some other topology which can be mapped to a ring.

The *volume data* is a discrete three dimensional grid of values. A volume v of resolution N is the following set: $v = \{(x, y, z) | 0 \leq x, y, z < N\}$

We define the *initial state* to be the memory partitioning of the volume data v among the processing units. In the initial state, processing unit i contains the following set of voxels $\{(x, y, z) \,| x = i, (x, y, z) \in v\}$. That is, the volume data is distributed along the major axis.

A *shift* operation is a global communication routine in which all the processing units send their data to their adjacent neighbor on one side and receive data from the neighbor on the other side. We use the term *long shift* to distance k to denote a sequence of k consecutive shifts.

4 Alignment

First let's examine the *alignment* process on a two dimensional grid. Suppose we have a square grid of $N \times N$ pixels and that every processing unit contains one column of the pixels grid. We would like to scan the entire grid using an eight-connected line algorithm (such as Bresenham's line algorithm) from an arbitrary angle. The 2D lines are analogs of the rays that traverses the volume in the 3D case.

Let us consider the case when the viewing angle is $0°$. In this simple case which is the *initial state*, data is distributed among the processing units in such a way that every processing unit scans the column stored in its local memory. Now we examine the general case of an arbitrary viewing angle in the range of $-45°$ and $45°$. If we were to keep the current distribution of data among the processing units then we would have to communicate, in a regular form, information while scanning. Instead we propose an alternative which is executed in two phases. In the first phase which we call *alignment*, we perform a "virtual" scanning in which data is moved among the processing units in such a way that after the alignment, every processing unit holds in its local memory the data required for scanning the diagonal line. In the second phase, we scan the data in local memory achieving the same effect as the $0°$ case.

At different viewing angles the grid needs to be scanned by a different number of lines. The number of lines varies between N at $0°$ and $2N$ at $45°$, although the amount of data remains the same. Observe that for every line that ends at the right side of the grid there is a corresponding line that begins at the left side (wrapped around). Using this property of eight-connected lines we allow every processing unit to scan one or two rays depending on whether a ray exits at the top side or right side of the pixel grid. Since the processing units are interconnected in a ring-like topology, the wrap-around alignment is easily achieved by partitioning the memory in a straightforward column-wise fashion. The distribution of work among the processing units is such that each of them makes exactly N steps, providing an inherent load balancing (Figure 1). Figure 2 is a pseudo-code of the alignment process.

The alignment consists only of shifting rows with no filter or other destructive operations. The shift function is a one to one mapping and is thus reversible. The fact that the alignment is non destructive is a key point of the incremental alignment, as will be discussed next.

5 Incremental alignment

In a typical interactive volume rendering application, the viewing direction incrementally changes by small angles for example by sampling the movement of the mouse. An incremental transition from one angle α to the next $\alpha + \xi$ is desirable. Using the fact that the alignment process is non-destructive and the template property, the data is already aligned to α therefore only a minimal number of shifts are required to correct the rows in order to match the $\alpha + \xi$ direction (see Figure 4). The pseudo code shown in Figure 3 is a modification of the pseudo code shown in Figure 2 that supports incremental alignment.

Communication is also reduced by the fact that in short distance shifts, only part of the rows need to be shifted. See Figure 4 the shifts that need to be done on each row are depicted in (d). Note that many rows need no shift at all, while others need only a short shift either to the left or to the right.

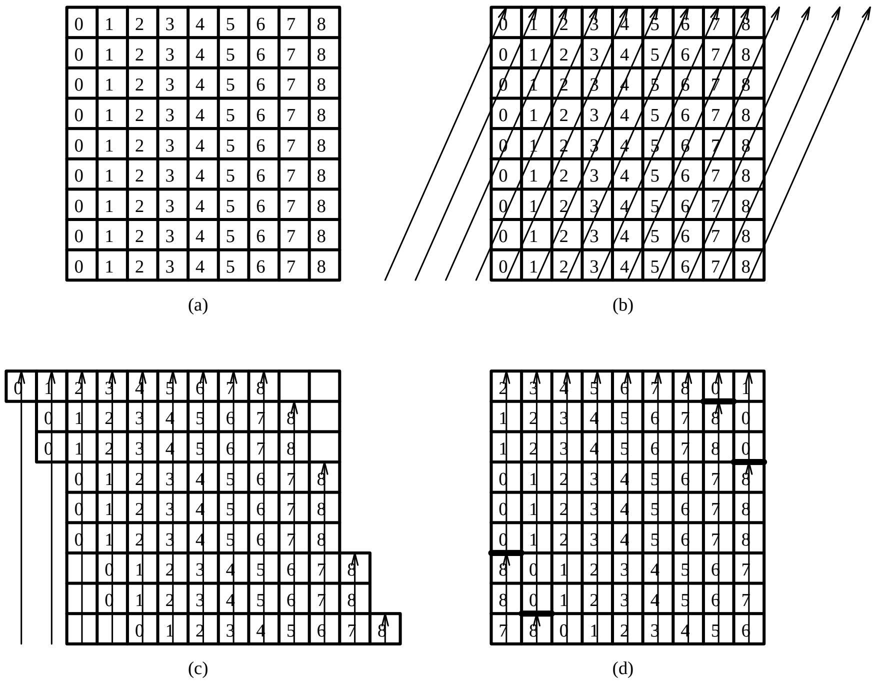

Figure 1: *The alignment process of an $N \times N$ grid. The numbers denote the slice number. The initial state (a). The rays path at an arbitrary viewing angle (b). The grid aligned with the rays (c). The aligned data after wrapping (d), observe that some processing units contains one ray, while others contain two rays. In any case, the number of voxels that are processed by each processing unit is exactly N.*

```
align(int $A[N]$, Angle $\theta$) {
    Pattern P; /* an array of integers where $x = p[y]$
    int i;
    create_pattern($P, \theta$);
    for ($i = 0; i < N$; $i++$)
        $A[i]$ = shift($A[i]$, $P[i]$);
}
```

Figure 2: *Pseudo-code that aligns a two-dimensional grid. Every row is shifted according to the pattern of an eight-connected line (the array P).*

```
incremental_align(int $A[N]$,Angle $\alpha_t$,Angle $\alpha_{t+1}$) {
    Pattern $P_1$,$P_2$;
    int i;
    create_pattern($P_1$,$\alpha_t$);
    create_pattern($P_2$,$\alpha_{t+1}$);
    for ($i = 0; i < N$; $i++$)
        $A[i]$ = shift($A[i]$, $P_2[i] - P_1[i]$);
}
```

Figure 3: *Pseudo-code that incrementally aligns a two-dimensional grid. Every row is shifted to adjust the previous alignment to the new angle.*

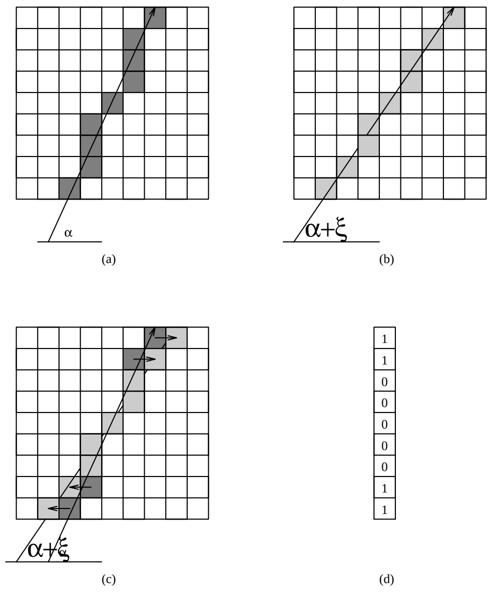

Figure 4: *(a) shows the set of pixels a ray at angle α traverses. This set of pixels is held locally to a processing unit memory. (b) shows the set of pixels a ray at angle $\alpha+\xi$ traverses, this set of pixels need to be aligned to a local memory. (c) shows that only a few rows need to be moved among processing units when changing from angle α to $\alpha+\xi$.*

When the increments are of small angles the set of shifts are short distance shifts. On most topologies short shifts are executed faster and are proportional to the distance. This, in addition to the fact that small changes of the viewing direction do not require every row to be shifted, provides a very fast mechanism for ray alignment (see Table 1).

Table 1: *Average number of regular shifts to adjacent neighbors for 128^3 and 256^3 volume when the volume is rotated from $0°$ to $45°$ at different steps of angles.*

Δ_{angle} (degrees)	Average number of shifts to distance of 1/step		Long distance shifts			
			Average number of shifts/step		Maximum distance of a shifts	
N	128	256	128	256	128	256
1	89	356	75	203	3	5
2	172	684	99	222	5	8
5	409	1648	106	222	11	21
10	678	2739	98	201	16	33

The incremental algorithm presented so far is limited to angles in the range of $-45°$ and $45°$. Clearly, extending the range of rotations is necessary for viewing from an arbitrary angle. One method of moving from the $45° - \epsilon$ angle to the $45° + \epsilon$ angle is to take the original data with "slices" parallel to the Z-Y plane, perform a $90°$ flip and then align to $45° + \epsilon$ [12]. Adapting such a method violates the incremental property of our algorithm. Instead we can change the viewing angle from $45° - \epsilon$ to $45° + \epsilon$ (and similarly at $135°$, $225°$ and $315°$ angles) in a cheap function that does not conflict with

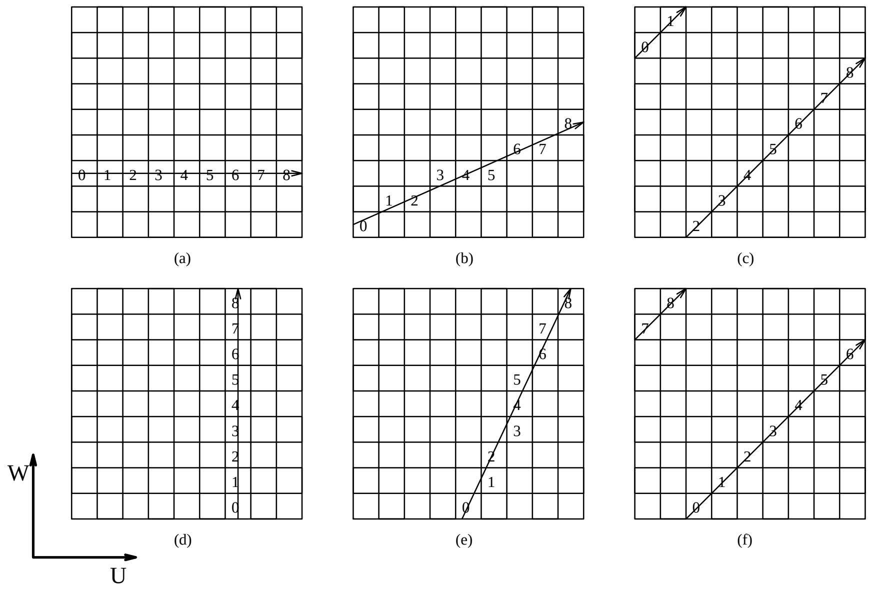

Figure 5: *Figures (a), (b) and (c) shows a series of rotations from* $90°$ *to* $45°$*, where z is the major axis. Figures (d), (e) and (f) shows a series of rotations from* $0°$ *to* $45°$*, where x is the major axis. The numbers denote the pixel's address within a processing unit local memory. One can see that the difference between Figure e and Figure f is the internal order of the data within a processing unit.*

the speed of the incremental alignment for any other angle.

To explain the transition from $45° - \epsilon$ to $45° + \epsilon$ let us look at two occurrences, in the first the x-axis is the major axis, we rotate the grid counterclockwise towards $45°$. In the other, the major axis is the z-axis, again we rotate the grid toward $45°$ but clockwise this time. When both reach the angle of $45°$ each scene has its own partitioning of the volume-data. By carefully examining this partitioning we observe that every processing unit holds exactly the same set of pixels, although in a different order. This difference is caused by the choice of major axis (see Figure 5). Therefore, all we need is a switch of the major axis and to continue aligning beyond $45°$.

Observe Figure 5, the row in (a) is rotated counterclockwise through (b) until (c) where it is rotated to $45°$. Now, observe the column in (d) which is rotated clockwise through (e) to $45°$ in (f). The difference between (c) and (f) is merely the internal order of the data within the local processing units memory. Changing the internal order is done locally either by a reordering of the data or by using look up tables.

6 Alignment and Volume Rendering

Alignment is a non-destructive mechanism that maintains the property that all the data a processing unit needs for rendering from an arbitrary direction resides in its local memory. The extension of the 2D alignment to three dimensions treats the volume as a set of two dimensional grids piled one on top of the other (Figure 6). Each processing unit holds a vertical slice of the volume the counter-part of the 2D row. The volume alignment is thus a sequence of the two dimensional alignments. However, since the alignments of each volume layers are of the same form it is possible to replace the outer loop which runs over all the layers by a shift of the entire column instead of a single volume element, saving an order of magnitude in communication steps while shifting the same amount of data.

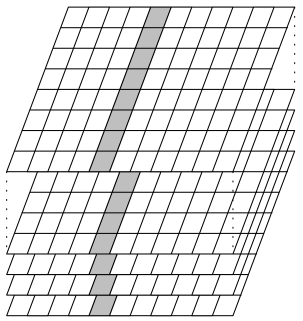

Figure 6: *Piling two-dimensional grid, forms a volume. The dark voxels are the voxels held by one of the processing elements at the initial state.*

The above data partitioning permits rotations around the X-axis by casting diagonal rays into the local slice without a need for alignment since the vertical slice is invariant under a horizontal rotation. This of course avoids the communication time when rotating only around the X-axis. A single axis rotation is a very powerful interactive tool for observing a 3D object. An arbitrary rotation is thus composed of two rotations of which only one requires alignment. A rotation around the z-axis can be avoided or delayed until after rendering by rotating the projected image. It should be noted that a Z-axis rotation is uncommon in practice, a fact that suggested partitioning the volume data non symmetrically to the three axes.

The discussion has so far assumed that the number of processing units is N while the volume dimensions are N^3. The alignment algorithm with minor modifications works when p the number of processing units is less than N. By modifying the *initial state* is such that every processing unit contains a "slab" of the volume. A slab is a set of adjacent slices and the slab thickness is N/p slices. A second alternative is a cyclic partitioning, where processing unit i holds the following set of slices $\{slice_j | 0 \leq j < N \text{ and } (j \text{ mod } p) = i\}$, where p is the number of processing units in the system. The cyclic data partitioning yields better load-balancing due to its randomized nature. However, the slab data partitioning is preferable for two reasons. First, the amount of communication required is smaller than of the cyclic partitioning since small changes of the viewing angle might keep more data within a processing unit. The second reason is the lesser amount of data duplication that is required by the padding technique described below.

The implementation of a volume rendering algorithm should include some form of anti-aliasing. The aliasing is caused by low sampling rates relative to the inherent frequencies of the data. Assume that we sample and render the image by ray casting. One possible solution is super-sampling the volume by accelerating the sampling rate along the ray and by casting more rays from one image pixel. Another solution improves the single sample by use of some 3D filter. A common and simple filter is a trilinear interpolation of the voxel at the sampling point [7]. Trilinear interpolation requires access to the eight corners of the voxels which contains the sample. We wish to support this access without sacrificing the locality of the rendering phase. For this purpose each slab is padded by an adjacent slices that otherwise would have to be brought in from adjacent processing unit(s). (see Figure 7). If the samples occur only at voxels walls a bilinear interpolation of the four corners of the voxel walls will require only a single padding slice. However, if the samples occur arbitrarily along the ray, a trilinear interpolation will require a double pad of two slices. The padded slices overlap between the adjacent slices which means some extra space. This suggests that larger slabs are more attractive as the addition of slices becomes relatively smaller. But on the other hand, our alignment mechanism offers almost linear speed up and a large number of processing units means smaller slabs.

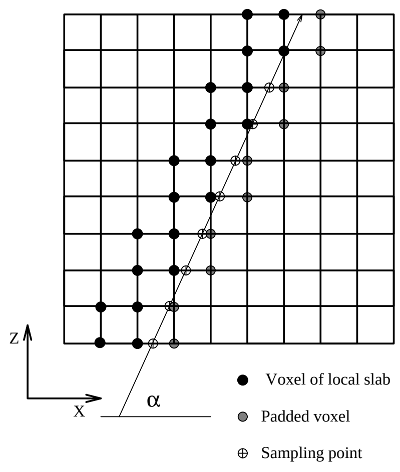

Figure 7: *A ray samples along a slab of two slices (painted in black). The ray grazes the slab. The sample points are interpolated with the value of the pad slice (painted in gray).*

7 Performance Analysis

In order to evaluate a parallel algorithm in general and our alignment algorithm in particular we have to analyze its communication overhead, which defines the speed-up provided by the algorithm. The communication overhead consists of the data transfer time and the latency time associated with each communication operation. However, since we are interested in hardware independent values, we are assuming that the communication throughput and its latency are given, and we account only for the volume of the transferred data and the number of communication operations. An efficient parallel algorithm minimizes on both criteria. In this way we also avoid biased results due to the efficiency of the software implementation.

Plain numbers such as the ones in Table 1 are not of much interest unless they are compared to other algorithms. We have implemented two algorithms that represent the two basic approaches used in parallel volume rendering. The first is a three-pass shear based algorithm [15] which can be implemented on a parallel machine with a ring topology in such a way that only one shear out of the three requires inter-processor communication. A second algorithm is the line-drawing algorithm [5] described in Section 2.

Figures 8 and 9 show the amount of data that is communicated and the number of communication operations required by the three algorithms. The incremental alignment algorithm performs well on both criteria and it shifts small amount of data uniformly over different angles when rotating the volume from $0°$ to $45°$, while the other two algorithms perform significantly better at $0°$ than at $45°$. Note in Table 10 that the number of voxels that are transferred by the alignment algorithm is independent to the viewing angle yielding a stable communication load.

Both the three-pass shear and the alignment algorithm separates the communication phase from the rendering phase, resulting in two significant benefits. One benefit is that no communication is required for rendering if any viewing parameters other than the rotation angles are changed. The other advantage is that when the communication is interleaved with computation, the processing unit must operate in synchronous lock-steps losing many computation cycles. Separating the communication and computation into two different phases reduces the number of the synchronous lock steps.

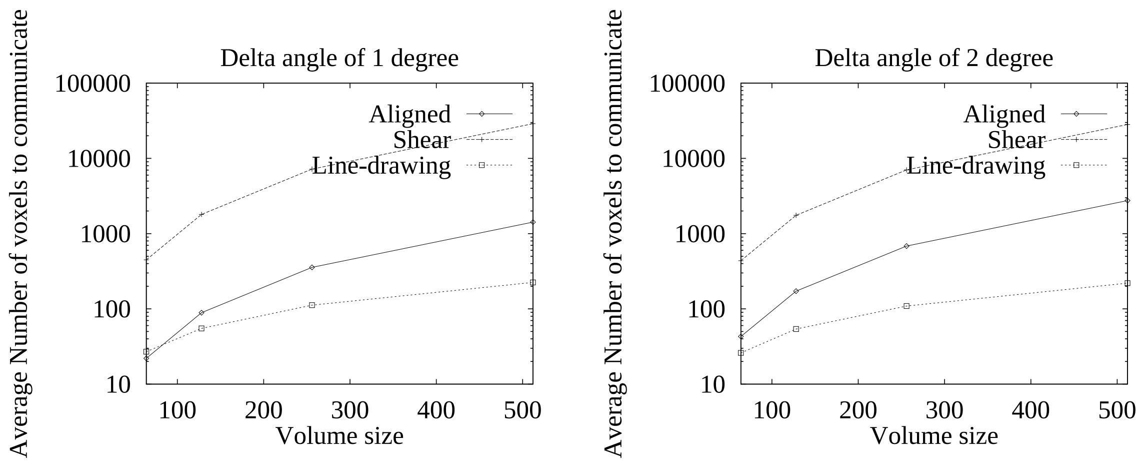

Figure 8: *Average number of voxels that is communicated at a step, when rotating a volume from the* 0° *to* 45°*.*

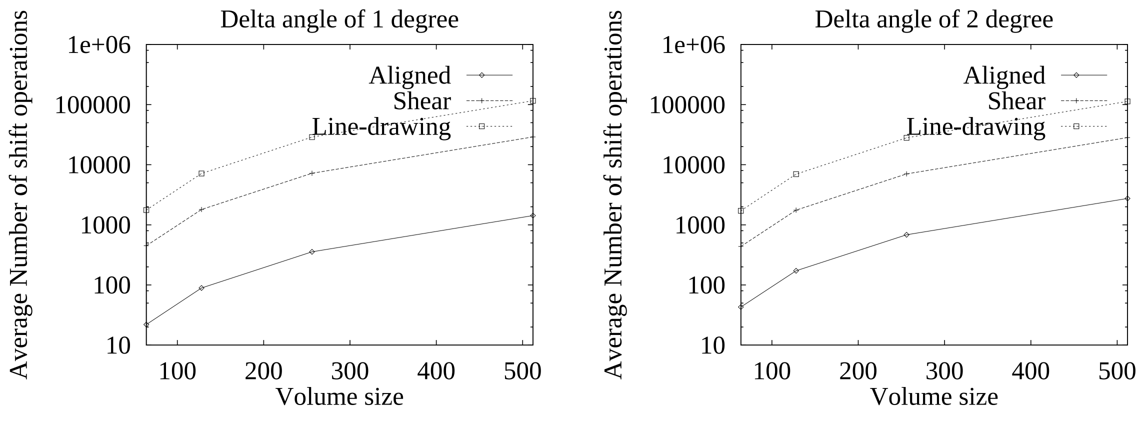

Figure 9: *Average number of shift operations performed at a step, when rotating a volume from the* 0° *to* 45°*.*

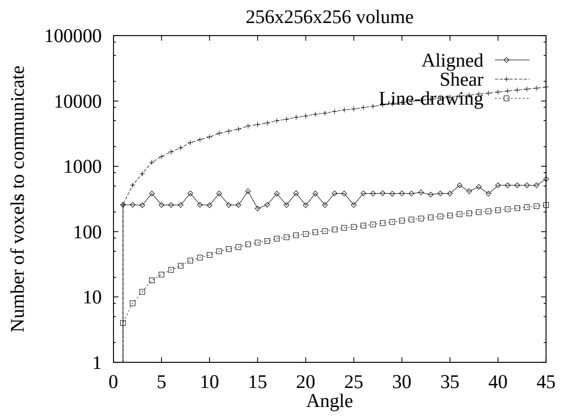

Figure 10: *The number of voxels that are communicated at each viewing angle when rotating the volume from* $0°$ *to* $45°$ *by steps of* $1°$*.*

8 Conclusion

We have presented an incremental alignment algorithm for parallel ray casting. The efficiency of the algorithm stems from its low communication overhead. In our analysis we avoided measuring efficiency in terms of speed-up because this criteria can be biased by the computation time spent in the rendering process or other parameters irrelevant to the alignment itself. If the computation time is increased while the communication time is kept constant, the speed-up factor seems to indicate better results. We have presented results which are independent of the computation time. The communication overhead is measured by the amount of transferred data and the number of communication operations. Yet, the alignment algorithm is well-balanced and requires a regular inter-processor communication pattern, which avoids conflicts or complicated routings.

The alignment process is a non-destructive transformation, consisting of a single non-scaling shear operation. This is a unique feature which provides the basis for the incremental algorithm presented in this paper. Filtering is delayed until the rendering phase. This is in contrast to shear-based algorithms which include scaling and filtering of the original data.

References

1. A. Kaufman, R. Bakalash, D. Cohen, and R. Yagel. Architectures for volume rendering - survey. *IEEE Engineering in Medicine and Biology*, 9(4):18–23, December 1990.

2. H. Pfister and A. Kaufman. Real-time architecture for high-resolution volume visualization. *Proc. 8th Eurographics Workshop on Graphics Hardware*, pages 72–80, 1993.

3. C. Montani, R. P., and R. Scopigno. Parallel volume visualization on a hypercube architecture. In *1992 Workshop on Volume Visualization*, pages 9–16, Boston, MA, October 1992.

4. P. Schröder and J.B. Salem. Fast rotation of volume data on data parallel architectures. In *Visualization'91*, pages 50–57, San Diego, CA, October 1991.

5. P. Schröder and G. Stoll. Data parallel volume rendering as line drawing. In *1992 Workshop on Volume Visualization*, pages 25–31, Boston, MA, October 1992.

6. G. Vezina, P.A. Fletcher, and P.K. Robertson. Volume rendering on the maspar MP-1. In *1992 Workshop on Volume Visualization*, pages 3–8, Boston, MA, October 1992.

7. T. Fruhauf. Volume rendering on muliprocessor architecture with shared memory: A concurrent volume rendering algorithm. *In 3rd Eurographics Workshop on Scientific Visualization*, April 1992.

8. R. Machiraju and R. Yagel. Efficient feed-forward volume rendering techniques for vector and parallel processors. *Proc. SUPERCOMPUTING'93*, pages 699–708, November 1993.

9. Indurkhya B., H. S. Stone, and L. Xi-Cheng. Optimal partitioning of randomly generated distributed programs. *IEEE Transactions on Software Engeneering*, SE-12(3):483–495, March 1986.

10. M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(5):29–37, May 1988.

11. R. Yagel and A. Kaufman. Template-based volume viewing. *Computers and Graphics, Proc. Eurographics'92*, pages 153–157, 1992.

12. R. Yagel. The flipping cube architecture for real time volumetric graphics. *Eurographics'91 Hardware Workshop*, 1991.

13. D. Cohen and A. Kaufman. A 3D skewing and deskewing scheme for conflict-free access to rays in volume rendering. *IEEE transtactions on Computers*, 44(4), April 1995.

14. E. Catmull and A.R. Smith. 3D transformations of images in scanline order. *Computer Graphics*, 14(3):279–285, July 1980.

15. P. Hanrahan. Three-pass affine transforms for volume rendering. *Computer Graphics*, 24(5):71–78, November 1990.

16. A. Law and R. Yagel. Voxelflow: A parallel volume rendering method for scientific visualization. *submitted to the International Conference on Computer Applications in Engineering and Medicine*, October 1994.