A Parallel Pipelined Renderer for Time-Varying Volume Data

Tzi-cker Chiueh[1]* Kwan-liu Ma[2]

[1]Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
chiueh@cs.sunysb.edu

[2]Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA 23681-0001
kma@icase.edu

Abstract

This paper presents a strategy for efficiently rendering time-varying volume data on a distributed-memory parallel computer. Visualizing time-varying volume data take both large storage space and long computation time. Instead of employing all processors to render one volume at a time, a pipelined rendering approach partitions processors into groups so that multiple volumes can be rendered concurrently. The overall rendering time is greatly minimized because rendering is overlapped with I/O required to load the volume data sets. Moreover, parallelization overhead may be reduced as a result of partitioning the processors. We modify an existing parallel volume renderer to exploit various levels of rendering parallelism and to study how the partitioning of processors may lead to optimal rendering performance. We find that two factors affecting the overall execution time are resource utilization efficiency and pipeline startup latency. The optimal partitioning configuration is the one that balances these two factors. Tests on Intel Paragon computers show that in general optimal partitionings do exist for a given rendering task and result in 40-50% saving in overall rendering time.

1 Introduction

Time-varying volumetric data sets (TVVD), which may be obtained from numerical simulations or remote sensing instruments, provide scientists insights into the detailed *dynamics* of the phenomenon under study. When appropriately rendered, they form an animation sequence that can illustrates how the underlying structures evolve over time. For visualizing large data sets, parallel processing is often adopted to speed up the expensive volumetric rendering process. Although the subject of rendering a single volumetric data set using a parallel computer has been studied extensively by numerous researchers [14, 13, 11, 19, 7], parallel animation of TVVD, in contrast, received relatively little attention.

Compared to parallel volume rendering of a single data set, rendering TVVD in parallel poses a different set of design tradeoffs. First, because TVVD typically consists of a sequence of data volumes, the I/O overhead to bring the data into the parallel machines, accounts for a significant portion of the end-to-end response time, and can no longer be ignored as is usually done by many researchers in parallel volume rendering. The key technique to address this I/O problem is to *hide* the I/O overhead by overlapping computation with I/O. Secondly, since a TVVD rendering job is actually comprised of multiple rendering tasks, it is important to make efficient utilization of the computation resources so that the overall rendering time is minimized. In particular, one should remember that parallelization almost always incurs certain overhead such as data distribution, result collection, or synchronization. Therefore it is critical to balance between the parallelism and overhead of individual rendering tasks, with the goal of optimizing the overall performance of the entire TVVD rendering job. Thirdly, whereas in single-data-set rendering, the response time is the single most important criterion, in TVVD rendering there are multiple criteria that are potentially of interest to the users. One possibility is the start-up latency, the time until which the first image appears. Another candidate is the overall execution time, the time until which the last image appears. Depending on the requirements of the end users, different design tradeoffs are to be made to optimize different performance criteria.

We argue that parallel volume animation requires re-thinking of the types of parallelism one should ex-

*This research was supported by the National Aeronautics and Space Administration under NASA contract NAS1-19480 while the author was in residence at the Institute for Computer Applications in Science and Engineering, NASA Langley Research Center.

ploit to achieve the optimal performance. In particular, I/O overlap and resource utilization efficiency play a crucial role in the parallelization strategy. We start with a generic parallel volume rendering program [11], modify it to experiment with different approaches to support parallel volume animation for time-varying data sets, and analyze the performance tradeoff among various partitioning strategies. Although the results and analysis are based on implementations on Intel Paragon, we believe that the conclusions should remain valid for other parallel distributed memory architectures.

2 Related Work

Ideally, visualizing time-varying volume data should be done while data are being generated, so that users receive immediate visual feedback on the subject under study and the visualization results are stored rather than the raw data which are much larger. VISUAL3 [5] and SCIRun [15] are among the many software systems that can support runtime tracking of three-dimensional numerical simulations. These systems may be operated in a distributed computing environment. Rowlan [16] and Ma [9] also demonstrate such tracking capability using direct volume rendering on a massively parallel computer. However, runtime tracking is not always possible and desirable for certain applications. For example, one may want to explore the data set from different perspectives; or, the amount of computation power required for real-time rendering or a special visualization technique may be not readily available. As a result, postprocessing of pre-calculated data is still being widely used by many scientific researchers.

Several techniques have been developed for visualizing time-varying data as a postprocess. Lane [8] develops a particle tracer for three-dimensional time-dependent flow data. Max and Becker [12] apply textures for visualizing both steady and unsteady flow field. Silver and Wang [20] presents a volume based feature tracking algorithm to help visualize and analyze large time-varying data sets.

What more related to our work is the ray-cast rendering strategy introduced by Shen and Johnson [18] which they call *differential volume rendering*. By exploiting the data coherency between consecutive time steps, they are able to reduce not only the rendering time but also the storage space by 90% for their two test data sets. Differential volume rendering is potentially parallelizable and a caching technique [10] may be integrated into the renderer to avoid recalculations for visualizing irregular data. Goel and Mukherjee [4] also develop an approach similar to Shen and Johnson's and achieve comparable saving.

On the other hand, following the success of MPI, MPI-IO represents another collective effort to propose a standard makes developing a truly portable parallel program possible. The current status of this effort can be found in [1]. Even with the presence of parallel I/O, we cannot guarantee that I/O time becomes less dominant, especially when processor technology is advancing in a faster pace than I/O technology. In fact, the strategy we develop in this research can be used in conjunction with parallel I/O to achieve maximum performance.

There has also been previous research discussing the I/O characteristics of visualization applications on parallel parallel computers [17]. However, most of these projects are related to generic parallel I/O issues. Chiueh [3] presented a memory access algorithm that allows conflict-free access to an interleaved memory system that stores volumetric data sets. The same algorithm is directly applicable in the context of parallel disk arrays. The work described here, in contrast, focuses mostly on resource utilization and parallelism to optimize the overall process of visualizing time-varying volume data on parallel distributed-memory architectures. We also want to to investigate the feasibility of building a volumetric data management system [6, 2] that is easier to use on the one hand, and is capable of efficiently interfacing with parallel rendering engines on the other.

3 Parallelization Approaches

The basic structure of a generic parallel volume rendering program [11] forms a three-step pipeline: *3D data distribution*, in which the volumetric data set is decomposed into subvolumes and distributed to the processor nodes, *subvolume rendering*, in which each processor node renders the assigned subvolume into a 2D subimage, and *image compositing*, in which the set of 2D subimages from the previous step are composited according to the view angle to arrive at the final 2D projected image. When the degree of parallelism is small to modest, i.e., under 16 nodes, the major portion of the computational overhead is attributed to *subvolume rendering*. However, when the degree of parallelism is high or when the data set itself is large (say 1024^3), *3D data distribution* becomes a significant performance factor.

Given a generic parallel volume renderer and a P-processor machine, there are three possible approaches to turn it into a parallel volume animator for TVVD sets. The first approach simply runs the parallel volume renderer on the sequence of data sets one after another. At any point in time, the entire P-processor machine is dedicated to rendering a particular volume [1]. Therefore, only the parallelism associated with rendering a single data volume, i.e., *intra-volume* parallelism, has been exploited. The second approach takes the exact opposite approach by rendering P data volumes simultaneously, each on one processor. This approach thus only exploits *inter-volume* parallelism. As the optimal systems performance can only be achieved by carefully balancing two performance factors: resource utilization efficiency and the parallelism exploitation overhead, both *intra-volume* and

[1] Here we assume the pipeline effect is ignored.

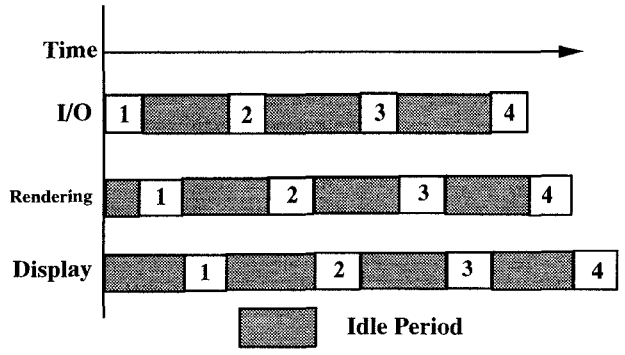

Figure 1: *The utilization of the system components under the Intra-Volume approach. The numbers denote the data volume number.*

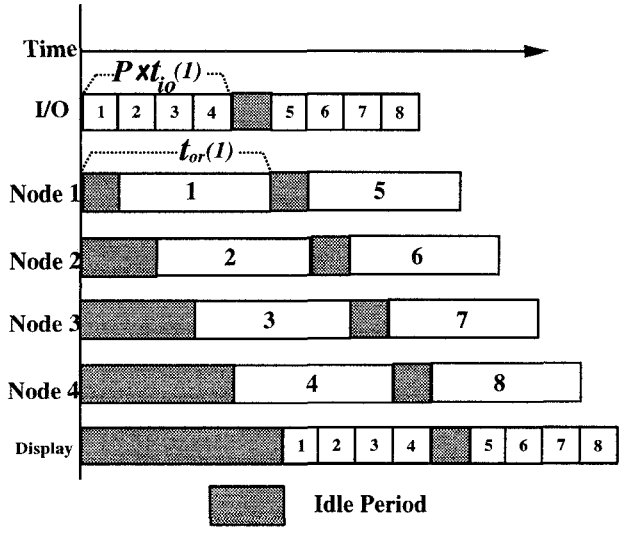

Figure 2: *The utilization of the system components under the Inter-Volume approach when* $t_{or}(1) > P \times t_{io}(1)$. *The number in each box denotes the data volume number. The number of processors, P, is assumed to be 4.*

inter-volume parallelism should be exploited. The third approach thus takes the hybrid approach, in which P processor nodes are partitioned into L groups ($1 < L < P$), each of which renders one data volume at a time. We will show later on that the third approach indeed performs the best among the three. However, the optimal choice of L depends on the type and scale of parallel machines as well the size of data sets. Detailed characterizations of the optimal partitioning strategy are described in Section 5.

4 Performance Analysis

4.1 Metrics

Parallel volume animation of TVVD sets involves rendering multiple data volumes in a single task. There are three potential performance metrics: *start-up latency*, the time until which the rendered image of the first volume appears; *overall execution time*, the time until which the rendered image of the last volume appears; and *inter-frame delay*, the average time between the appearance of consecutive rendered images. In conventional volume rendering applications, since there is only one data set involved, *start-up latency* and *overall execution time* are the same, and *inter-frame delay* is irrelevant. However, when volume animation is used interactively, *start-up latency* and *inter-frame delay* play an crucial role in determining the effectiveness of the system. When volume animation is run in the batch mode, then *overall execution time* should be the major concern. Note that different design tradeoffs have to be made for different performance criterion. For example, if *start-up latency* is the criterion of choice, then the first approach discussed in Section 3 probably should be the design of choice. In the rest of the paper, we will use the *overall execution time* as the main criterion and only mention the other two whenever appropriate.

4.2 Performance Models

Before we present the experiment results, it's useful to construct a performance model for each of the approaches described above so that one can have a basic understanding of the experimental results. For the rest of the discussion in this paper, without limiting the applicability of our research results, we assume a completely serial I/O system to focus on other issues.

Assume that there are N data volumes in the TVVD set, there are P processors in the system, and without loss of generality $N = k * P$. Let $t_{or}(p)$ denote the total rendering time for a single data volume using p processors, including file access and data distribution, rendering, compositing, and image delivery, $t_{io}(p)$ the time to distribute a data set from the disk to the p processors in the beginning of rendering a data volume, and $T(L)$ the overall execution time for rendering N data volumes when P processors are decomposed into L groups, each of which consists of $\frac{P}{L}$ processors. Note that we assume a completely serial I/O system in this study.

For the *intra-volume* approach, the overall execution time is

$$T(1) = N \times t_{or}(P) \quad (1)$$

Because P processors are collectively used to render one data volume at a time, the rendering task for the j-th volume won't start until that for the $(j-1)$-th volume ends. The timing diagram for this approach is shown in Figure 1. For the *inter-volume* approach, the overall execution time is

$$\begin{aligned} T(P) &= k \times max\{t_{or}(1), P \times t_{io}(1)\} + \\ &\quad min\{t_{or}(1) - t_{io}(1), (P-1) \times t_{io}(1)\} \quad (2) \end{aligned}$$

Because each data volume is rendered only by a single processor, there are at most P concurrent rendering tasks on the system. If $P * t_{io}(1) > t_{or}(1)$, then the system is IO-bound. That is, the rendering task for the $(P+j)$-th volume cannot start immediately

after the j-th volume is done. The second term in Equation (2) accounts for the fact that the completion time for the N-th volume is later than that for the $(N-P+1)$-th volume either by $(P-1) * t_{io}(1)$ when $t_{or}(1) < P * t_{io}(1)$, or by $t_{or}(1) - t_{io}(1)$ when $t_{or}(1) > P * t_{io}(1)$. The timing diagram for the inter-volume approach assuming $t_{or}(1) > P \times t_{io}(1)$ is shown in Figure 2. For the *hybrid* approach, assume that P processors are divided into L groups, each of which now contains $P_g = \frac{P}{L}$ processors, then the overall execution time is

$$T(L) = P_g \times k \times max\{t_{or}(P_g), L \times t_{io}(P_g)\} + max\{t_{or}(P_g) - t_{io}(P_g), (L-1) \times t_{io}(P_g)\} \quad (3)$$

As can be seen, the performance formula for the *inter-volume* approach is essentially an instance of that of the *hybrid* approach when $L = P$. Note that whether the rendering task is IO-bound or CPU-bound depends on the size of the data set as well as the number of processors in the system.

5 Test Results

5.1 Experiment Setup

An existing parallel volume rendering software [11] is modified in such a way that it can exploit different levels of intra-volume and inter-volume parallelism by varying the configuration parameter L, the number of processors dedicated to a single volume given that the total number of processors is fixed. Our tests are run on the 72-node Intel Paragon computer operated at the NASA Langley Research Center as well as the 512-node Intel Paragon computer at the California Institute of Technology. The data set is obtained from a time-dependent turbulent simulation and its size is $128 \times 128 \times 128$.

The general structure of the program is shown in Figure 3. Given P processor nodes, there are L virtual rendering nodes, each of which consists of $\frac{P}{L}$ physical processor nodes. In addition, there is a host node performs disk I/O access and volumetric data distribution. The same host node also collects the 2D subimages from each node to form the resultant image and sends it to the end user over the network. Because multiple data volumes are being rendered simultaneously, appropriate flow control is needed to maintain appropriate synchronization between the host node and the virtual render nodes. These are indicated in Figure 3 as gray lines going in both direction. Without proper synchronization, subimages from different rendering runs may become intermixed. For the rest of the discussion, the term "number of processors" refers to the numberof physical processor nodes involved in rendering only, i.e., excluding the I/O and display nodes. Also, the number of data volumes rendered in each run is made equal to the number of physical processors. We make this assumption to ensure that the pipeline start-up overhead will be appropriately accounted for in the performance evaluation.

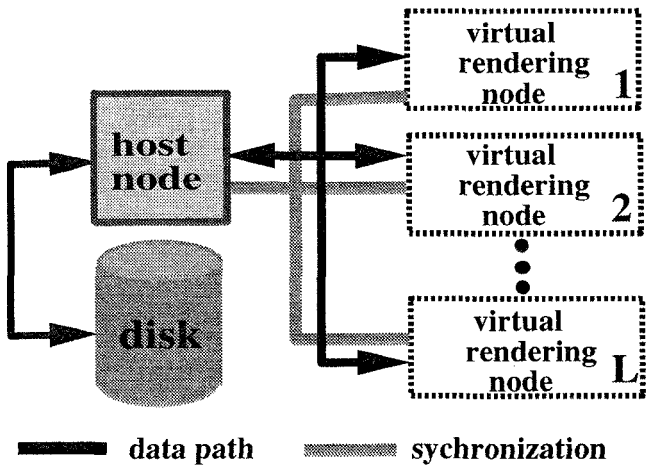

Figure 3: *Software architecture of the implemented parallel volume animator. P computation processors are partitioned into L virtual rendering nodes, each of which is responsible for rendering a single data volume loaded from disk through a host node.*

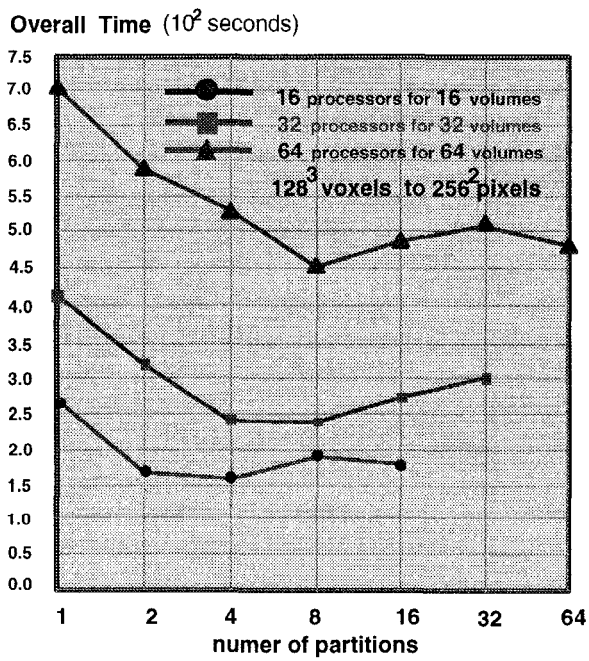

Figure 4: *The overall execution time versus the number of partitions for three different processor sizes.*

5.2 Results and Analysis

The measured overall execution time correlates quite well with the prediction from the model. Presumably this is because the performance model in Section 4 is stated in terms of delays associated with high-level primitives. Our conjecture that the optimal performance can only be achieved by effectively exploiting both intra-volume and inter-volume parallelism is confirmed by Figure 4, which illustrates the relationship between the overall execution time and the number of processor partitions (L), and is on a log_2 scale along the X axis. With 16 processors, the optimal number of partitions for rendering 16 data volumes is 2 or 4; with 32 processors, the optimal number for rendering 32 volumes becomes 4 or 8; with 64 processors, the optimal number is 8. We want to re-emphasize the overall execution time shown consists of three phases: *data distribution*,which includes both disk I/O and data distribution; *rendering*, which includes rendering and compositing; and *image display*, which includes collecting subimages and transferring the final image over the network.

Intuitively, when $L = 1$, each data volume is rendered one after another, without any overlap between different phases from consecutive runs. As a result, the utilization of various system components, as shown in Figure 1, is inherently suboptimal. For example, the utilization of the rendering nodes is

$$\frac{t_{rendering}}{t_{data_distribution} + t_{rendering} + t_{display}}$$

On the other hand, when $L = P$, it takes at least P runs for the entire pipeline to operate in full swing, as shown in Figure 2, where P is assumed to be 4. Unfortunately, since we assume there are a total of P data volumes in the sequence, the pipeline never has a chance to achieve its optimal throughput. Consequently, the overall execution time is the worst among all possible configurations for a fixed number of processors. It should be noted, however, that when the number of data volumes in the time-varying data set is much larger than the number of physical processors so that the start-up overhead can be effectively amortized, the inter-volume approach should achieve the best overall execution time because it incurs the least parallelism overhead. Our test results in fact show such trend in both 16- and 64-processor cases. In practice, this assumption is not necessarily always true, Especially, when the data set size exceeds the node memory, the inter-volume approach is simply not feasible. The optimal partitioning presumably minimizes the start-up overhead while maximizing the utilization efficiency of the rendering nodes.

As we mentioned earlier, there are multiple performance criteria for parallel volume animation of TVVD sets. Figure 5 shows the behavior of the three criteria described earlier versus the degree of partitioning, and the tradeoff among them. The number of processors in this case is fixed at 32. The start-up latency is monotonically increasing with the number of partitions because the number of processors dedicated to a single

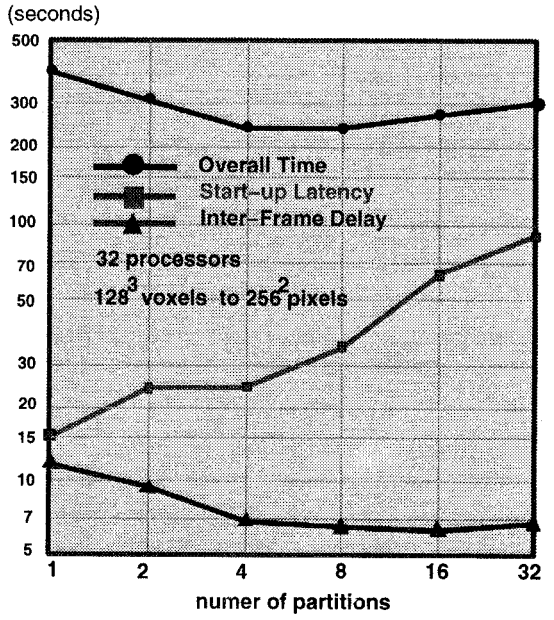

Figure 5: *The overall execution time, start-up latency, and average inter-frame delay versus the number of partitions, when P=32*

data volume is decreasing. The average inter-frame delay is computed by subtracting the start-up latency from the overall execution time and dividing the result by the number of data volumes rendered. Because of the dominance of the overall execution time, the inter-frame delay exhibits a somewhat similar curve as that associated with overall execution time. The computed inter-frame delay is almost identical to the average of the inter-frame delays from actual measurements. Note that the computed average inter-frame delay doesn't necessarily correspond to the *apparent* inter-frame delay that users experienced. In general, the rendered frames come in a burst, stop for a while, and repeat again. The fact that there is a stop period is symptomatic of an imbalance between the data distribution and rendering phases. It is interesting to observe that the smoothest rendering, i.e., the one with the shortest stop period between bursts, indeed occurs under the configuration that has the smallest overall execution time, because it is the most balanced among system components. For $P = 32$, $L_{optimal} = 4$ or 8.

Table 1 displays a more detailed look of the rendering cost, in which we show the time to generate a single frame by using up to 32 processors. The initialization time is mostly for computing the voxel gradient values for lighting calculations. This initialization must be done for each volume. Both initialization and the ray-cast resampling time increase dramatically as fewer processors are used to render a volume. The compositing time includes both calculation and communication time also decreases when more processors are used except for the one-processor case in which no compositing calculation is needed after the resampling process. The total rendering time illustrates that the

tasks	32 nodes	16 nodes	8 nodes	4 nodes	2 nodes	1 node
initialize renderer	0.269	0.654	1.593	3.36	7.02	12.96
ray-cast resample	2.8	5.5	9.5	19	37	64
composite partial images	1.068	1.43	2.32	3.747	5.96	0.00
total time	4.137	7.584	13.413	26.107	49.98	76.96

Table 1: *A breakdown of the rendering time for generating a single frame when using up to 32 processors.*

increasing parallelization penalty we get when using more processors in a partition. Hence, with the same number of processors, rendering multiple volumes concurrently reduces the aggregated parallelization overhead and gives us better overall throughput.

Note that the C++ implementation of the renderer preclude us from using the native compiler which results in at least 30% performance degradation. While our renderer may be optimized to obtain better rendering rates, the use of a highly optimized renderer in our study would show more significantly the relative performance degradation due to I/O delay.

6 Conclusions

Rendering time-varying volumetric data sets poses a different problem than rendering single-volume data set. We start with a naive approach by repeating the execution of a generic parallel volume renderer on the time-varying sequence of 3D data sets, and find that during the beginning and the end of the rendering process for a single data set, the nodes are mostly idle, thus wasting resources unnecessarily. To address this problem, we try to pipeline the rendering tasks for consecutive data sets in the sequence, essentially exploiting inter-volume as well as intra-volume parallelism. Given a fixed number of processor nodes and I/O bandwidth, the research question is what the optimal balance is between inter-volume and intra-volume parallelism exploitation. We have implemented a prototype volume renderer that embodies the idea of pipelined rendering for time-varying data sets. We are able to attain the most effective system utilization bounded only by the data distribution overhead. We also identify three possible performance criteria for evaluating TVVD data sets, and show that different partitioning strategies are needed to optimize for different criteria.

Our results show that there indeed exists an optimal partitioning for a given data set and a parallel computer configuration. But the optimum depends on such factors as the machine size, the length of TVVD sequence, and the ratio between computation and communication/IO overheads, which in turn is affected by the hardware characteristics and the coherence property of the data set itself. Thus, if these hardware system specific parameters are available, an optimal partitioning may be determined automatically.

This study also helps us identify the design issues to construct a volumetric data management system that can interface with parallel rendering engines efficiently. In this work, we find that a dedicated I/O manager plays an important role in improving the overall performance of TVVD rendering. It seems thus logical to include such an I/O manager in the envisioned volumetric data management system. However, there remains the work of developing a sufficiently flexible interface for the I/O manager that can smoothly intergrate with a wide variety of parallel renderers. As part of the volumetric database project, currently we are also working on volumetric data compression algorithms that are shown to be "friendly" to volume renderers, i.e., algorithms that can effectively exploit the coherency properties of volume rendering computation.

As we mentioned, this approach can be used in conjunction with parallel I/O facilities to achieve even better rendering rates. Furthermore, with a good parallel I/O system, the renderer can also read ahead by keeping multiple buffers at each rendering node: one for the current frame being rendered and one for the next frame being read ahead. The read aheads would then have to use asynchronous read request which return after the read is queued but before it completes.

The current implementation of the renderer may be optimized in two ways. First, it takes a slice-by-slice broadcasting approach to distribute the volume data set to the processor nodes, which then pick up the assigned portions of the slices. A more efficient approach is to store 3D subvolumes on the disk, and distribute 3D subvolumes to appropriate nodes directly. One advantage of this approach is the reduction of intermediate packing/unpacking overhead. Ultimately a database system specifically designed for efficient access to volumetric data will be the most desirable solution.

Second, currently all processors involved in a rendering run have to be either implicitly or explicitly synchronized. As a result, additional synchronization overhead is inevitable. An alternative approach is to take a dataflow, functionally-specialized model in which each processor node receives data packets, performs a fixed function, and sends them to the next processor node in the logical pipeline. Each piece of data travels across the system with a tag to identify the associated volume. With this architecture, there is no need to synchronize the processors in a lock-step fashion, thus reducing the synchronization delay. It's

up to the final pipeline stage to pull the subimages together and form the final image. All other nodes are in an autonomous loop and operate completely independently of one another. Because throughput is more important than latency for parallel volume animation, this model seems to be a better fit than the current implementation.

Acknowledgement

We want to thank the Center for Advanced Computing Research at the California Institute of Technology for the computer time on their Intel Paragon computers. Thanks also go to Jamie Painter for many useful suggestions. This first author is also supported by an NSF Career Award MIP-9502067 and a contract 95F138600000 from Community Management Staff's Massive Digital Data System Program.

References

[1] A Parallel File I/O Interface for MPI. http://lovelace.nasa.nasa.gov/MPI-IO.

[2] J. Boyle, S.G. Eick, M. Hemmje, D.A. Keim, J.P. Lee, and E. Sumner. Database Issues for Data Visualization: Interaction, User Interfaces, and Presentation. In *Proceedings of the IEEE 1993 Database Issues for Data Visualization Workshop*, pages 25–34. Springer-Verlag, 1994.

[3] T.-C. Chiueh. A Novel Memory Access Mechanism for Arbitrary-View-Projection Volume Rendering. In *Proceedings of Supercomputing '93 Conference*, 1993.

[4] V. Goel and A. Mukherjee. Volumetric Ray Casting of Time Varying Data Sets. In *Proceedings of the ICASE/LaRc Symposium on Visualizing Time-Varying Data*, pages 89–106, 1996. NASA Conference Publication 3321.

[5] Robert Haimes. Unsteady Visualization of Grand Challenge Size CFD Problems: Traditional Post-Processing vs. Co-Processing. In *Proceedings of the ICASE/LaRc Symposium on Visualizing Time-Varying Data*, pages 63–75, 1996. NASA Conference Publication 3321.

[6] P. Kochevar. Database Management for Data Visualization. In *Proceedings of the IEEE 1993 Database Issues for Data Visualization Workshop*, pages 109–117. Springer-Verlag, 1994.

[7] Philippe Lacroute. Real-Time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp Factorization. In *Proceedings of Parallel Rendering Symposium*, pages 15–22, 1995.

[8] D. Lane. UFAT- A Particle Tracer for Time-Dependent Flow Fields. In *Proceedings of the Visualization '94 Conference*, pages 257–264, 1994.

[9] Kwan-Liu Ma. Runtime Volume Visualization for Parallel CFD. In *Proceedings of Parallel CFD '95 Conference*, 1995. California Institure of Technology, Pasadena, CA, June 25-28.

[10] Kwan-Liu Ma, M.F. Cohen, and J.S. Painter. Volume Seeds: A Volume Exploration Technique. *The Journal of Visualization and Computer Animation*, 2:135–140, 1991.

[11] Kwan-Liu Ma, Jamie S Painter, C.D. Hansen, and M.F. Krogh. A Data Distributed Parallel Algorithm for Ray-Traced Volume Rendering. In *Proceedings of Parallel Rendering Symposium*, 1993. San Jose, October 25-26.

[12] N. Max and B. Becker. Flow Visualization using Moving Textures. In *Proceedings of the ICASE/LaRc Symposium on Visualizing Time-Varying Data*, pages 77–88, 1996. NASA Conference Publication 3321.

[13] Ulrich Neumann. Parallel Volume-Rendering Algorithm Performance on Mesh-Connected Multicomputers. In *Proceedings of Parallel Rendering Symposium*, pages 97–104, 1993. San Jose, October 25-26.

[14] Jason Nieh and Marc Levoy. Volume Rendering on Scalable Shared-Memory MIMD Architectures. In *1992 Workshop on Volume Visualization*, pages 17–24, 1992. Boston, October 19-20.

[15] S. G. Parker and C. R. Johnson. SCIRun: A Scientific Programming Environment for Computational Steering. In *On-line Proceedings of the 1995 Supercomputing Conference*, 1995. http://scxy.tc.cornell.edu/sc95/proceedings/.

[16] J.S. Rowlan, E. Lent, N. Gokhale, and S. Bradshaw. A Distributed, Parallel, Interactive Volume Rendering Package. In *Proceedings of the Visualization '94 Conference*, pages 21–30, 1994.

[17] K.E. Seamons and M. Winslett. An Efficient Abstract Interface for Multidimensional Array I/O. In *Proceedings of Supercomputing '94*, pages 650–659, November 1994.

[18] Han-Wei Shen and C.R. Johnson. Differential Volume Rendering: A Fast Volume Visualization technique for Flow Animation. In *Proceedings of the Visualization '94 Conference*, pages 180–187, October 1994.

[19] C.T. Silva and A.E. Kaufman. Parallel Performance Measures for Volume Ray Casting. In *Proceedings of Visualization '94 Conference*, pages 196–204, 1994.

[20] D. Silver and X. Wang. Volume Tracking. In *Proceedings of the Visualization '96 Conference*, pages 157–164, 1996.