Title
Real-Time Monitoring of Large Scientific Simulations

Authors
Pascucci, Valerio
Laney, Daniel E.
Frank, Randall J.
et al.

Publication Date
2003

Peer reviewed

Real-Time Monitoring of Large Scientific Simulations

V. Pascucci[1,2] pascucci@llnl.gov
D. E. Laney[1] dlaney@llnl.gov
R. J. Frank[1] rjfrank@llnl.gov
G. Scorzelli[3] scr.giorgio@cogesic.it
L. Linsen[4] llinsen@ucdavis.edu
B. Hamann[4] hamann@cs.ucdavis.edu
F. Gygi[1] gygi1@llnl.gov

ABSTRACT

We present a distributed framework that enables real-time streaming and visualization of data generated by large remote simulations. We address issues arising from distributed client-server environments and guarantee good parallel load balancing. We apply progressive computing techniques and parallel, hierarchical data streaming techniques to reduce the "distance" between the simulation hardware and the systems where the actual visualization and analysis occur. We present a simple and efficient load balancing method that scales to arbitrary simulation sizes and does not introduce additional communication cost. We demonstrate the performance of our system with a molecular dynamics code and show its ability to allow monitoring of all the time-steps of a large simulation with negligible time overhead.

1. INTRODUCTION

It is often the case in large high performance computing environments that the external systems, supporting applications such as visualization, are located remotely from the computational host. These systems are "remote" in terms of bandwidth and latency vs. dataset sizes. As the sizes of the datasets increase, there is a corresponding increase in the ratio between the compute power available to the scientific simulation (executed on a large supercomputer) and the compute power available to the external systems. Data transfer and conversion become a major cause of delay, and the external systems become increasingly "remote". This remoteness adversely impacts the steerability of a simulation.

Computational steering involves both inspecting and interpreting the state of an ongoing simulation and modifying simulation parameters. In this paper we focus on the real time assessment of the simulation results, which entails the visualization of data. Simulations that enable the adjustment of parameters at run time can use this technology to achieve a full-fledged steering system. A large number of tools and libraries for linking and steering computational simulations have been developed over the years [7, 10]. These systems have matured into the modern problem solving environments [9, 14, 1], which encompass the simulation building blocks as well. In general, these systems can become scalability bottlenecks when mapped onto large parallel and distributed memory architectures.

We propose a real-time streaming system based on a multi-tiered client-server architecture that enables the real-time monitoring of large scale simulations on distributed memory architectures. Each compute node utilizes a simple library call to stream data to one or more layers of data servers. On the client side, these servers are queried to obtain real-time multiresolution visualization of the data. The streaming system has the following properties:

Efficient. The system computes a fast permutation of the data, resulting in a subsampling hierarchy. This permutation is performed in-place with no expensive construction operations.

Cache Oblivious. The data layout follows the access patterns of typical level of detail visualization algorithms and its locality is independent of the various system cache sizes (CPU, disk, etc.). Therefore a unique data layout facilitates high performance of the system for different cache hierarchies.

Scalable. The system is built on the concept of stream-processors that can be dynamically realized to optimally utilize available computational and network resources.

Progressive. Progressive streams allow data streaming to be dynamically tailored to available network resources and desired visual fidelity.

The system described here addresses some fundamental issues arising from "remote" client-server architectures and parallel load balancing. It does so using progressive computing techniques [15] and parallel, hierarchical data streaming techniques [16]. We present practical results obtained by connecting our system with JEEP, a code that computes first-principles molecular dynamics based on the plane wave method [5]. We demonstrate our ability to perform real-time monitoring of all the time steps of a large scientific simulation at full resolution with negligible overhead.

2. RELATED WORK

In recent years there has been an increasing focus on data transport schemes and mechanisms that optimize information transformation in an attempt to decrease the "remoteness" of external systems. In the case of time-varying volumes, novel volume data encoding ensures timely data transport to graphics hardware [12]. Multi-tier caching schemes, which transform and store data for remote visualization, have been developed that utilize lower-bandwidth graphics primitives [3]. External memory techniques [2] allow efficient access to data that does not fit in main memory [13, 16]. Hierarchical representations for data transport have been proposed to reduce bandwidth requirements [6, 11, 8, 18].

A common requirement of such schemes has been the construction of hierarchical data structures before performing any visualization. In a real-time streaming infrastructure, this expensive computation stage cannot be neglected as off-line independent preprocessing. The present work obviates the need for such data structures by reordering the data into a subsampling hierarchy.

SAC '03, March 1-2, 2003, Melbourne, Florida.

[1] Center for Applied Scientific Computing (CASC), LLNL
[2] Corresponding Author: 925-423-9422
[3] Third University of Rome
[4] University of California at Davis
This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

3. DISTRIBUTED ARCHITECTURE

Three different types of components are used to construct the tiered client-server streaming architecture: Data Sources, Data Servers and Data Clients. Data Sources act as the data producers in the system. Typically, Data Sources are nodes of a running simulation, but the system supports alternate Data Sources, such as remote storage or real-time experimental data streams. Coupling to a simulation is achieved through a single function call inserted in the simulation code. This function is typically called after the computation of each time step. At each call, we convert the input dataset into a progressive hierarchical data stream and transmit the stream to a set of Data Servers.

The Data Servers constitute the central data processing component in the system. Each Data Server component takes a data stream as input and acts as a filtering and buffering agent. A Data Server outputs a data stream either to another Data Server or to a Data Client. Data Servers that perform pure data transformations may do so directly on the stream, without caching. Data Servers are also capable of caching data and responding to spatially bounded queries. We focus on a specific Data Server, which acts as a pure data storage component. Typically, several data servers are run in parallel with the data partitioned amongst them to take advantage of multiple processors and parallel disk I/O subsystems.

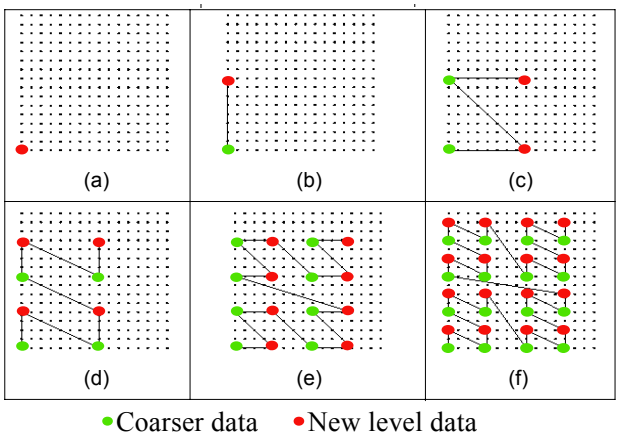

Figure 1: Sequence of levels of resolution of the hierarchical version of the Z-order space-filling curve. In each image fine black points in the background represent the fine resolution data. The dark gray points represent the new samples introduced at the current level of resolution. The light gray points mark the samples introduced in the previous levels of resolution.

The Data Clients act as data stream sinks. The Data Client presented here is a visualization tool that renders textured cut planes and performs progressive volume rendering to a local display. A Data Client makes spatially bounded data queries to its upstream Data Servers in response to interactive user requests. The Data Client receives the data streams corresponding to the samples satisfying the queries and builds a progressive data representation. The progressive nature of the stream representation allows it to be rendered at any time, even asynchronously, with respect to the receipt of the incoming stream.

Key to the scalability of the architecture is that all the system components are realized as distributed software entities with a fully interconnected communication infrastructure between each component. A major challenge in this type of architecture is to ensure load balancing throughout the entire system while maintaining scalability of the global communication. The format of the data stream itself addresses this problem. Implicit in the data ordering of the stream is the parallel distribution logic, which allows for asynchronous load balancing without global communication.

4. CACHE-OBLIVIOUS STREAMING

Consider a rectilinear grid G of size $2^g x 2^g x 2^g$. We turn G into a hierarchy by a reordering of its vertices based on the recursive definition of the 3D Z-order space-filling curve (see [17]). In particular, the nodes at resolution *l* are the nodes of the grid that are interpolated by the Z-order space-filling curve of resolution *l* but that are not interpolated by the Z-order curve at resolution *l*-1. Figure 1 shows this construction for the 2D case. At each level the space filling curve is alternately refined along the x and y axes. Figure 1(a) shows level 0, which is just a (dark gray) vertex. Figure 1(b) shows level 1, which is the new vertex marked in dark gray (the vertex at the previous level is light gray). At each level the number of new vertices introduced is equal to the total number of vertices in all the previous levels of resolution. This data layout is called "cache oblivious" since the locality of the mapping is independent of the block size used to partition the 1D array.

The conversion from the standard *(i,j,k)* index of a row major array order is performed in three simple steps as shown in [16]. First, the bits of the binary representation of *i*, *j* and *k* are interleaved to form the standard Z-order index I_G'. This transformation is denoted $I_G'=Z(i,j,k)$. Next, a sequence of right shifts is used to transform I_G' into the final hierarchical index I_G. This transformation is denoted $I_G=S(2^{3g}+IG')$, where *3g* is the total number of bits used by *(i,j,k)*. Overall, we have

$$I_G = S(2^{3g} + Z(i,j,k)). \quad (1)$$

Unfortunately, while the mapping from *(i,j,k)* to I_G is very efficient, the inverse mapping from I_G to *(i,j,k)* is slow and cannot be used directly in the inner loops of a computation. Consequently, the preprocessing that reorders the data can introduce significant delays before one can access the data efficiently. In the next section, we show how to overcome this problem and allow direct streaming of the data while hiding the preprocessing stage in the data transmission stage.

4.1 Streaming with embedded reordering

One main advantage of the approach proposed in [16] is that the data preprocessing is reduced to computing a permutation of the storage order. While this fact provides orders of magnitude of improvement with respect to previous methods, it would introduce a significant delay in the data servers since they would need to store a temporary copy of the data and then perform a redistribution of the data based on the new index.

The entire dataset *G* is partitioned into equal bricks *B1…Bm*. Each brick is a grid of $2^b x 2^b x 2^b$ vertices and is associated with one Data Source. Our strategy in reordering the data is to create direct connections between each Data Source and every Data

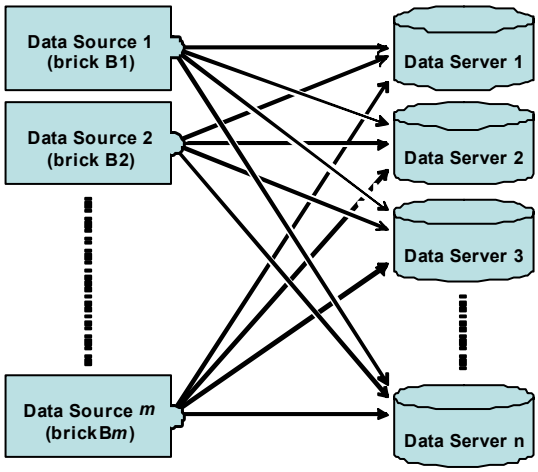

Figure 2: Direct socket connections used to stream data from each Data Source to every Data Server.

Server as shown in Figure 2. Conceptually, each Data Source loops through all the data in the order imposed by the index I_G and determines for each vertex (i) which data server must receive its value and (ii) the *(i,j,k)* index of the vertex to be sent (if stored locally).

In general, the vertex of hierarchical index I_G is stored in the block I_G/d, where d is the size of the data blocks in the Data Servers. Moreover, we assume that the Data Server storing such a block has the index

$$\frac{I_G}{d} \bmod n, \qquad (2)$$

where n is the number of Data Servers available. We discuss in the following section how to select a value of n that yields good load balancing in the data visualization stage. Two fundamental efficiency issues arise from a direct implementation of this scheme:

(i) Performing a loop based on the index I_G on each Data Source induces large delays since it requires enumerating all the vertices of G instead of only those of the local brick of data B.
(ii) The computation of the inverse mapping $I_G \rightarrow (i,j,k)$ is slow and its evaluation for each vertex, even if only those in B, induces additional delays.
To overcome these problems we take advantage of two fundamental properties of the index I_G, derived from the recursive structure of the Z-order curve:

Property 1. The traversal order of the vertices in a brick B, induced by the index I_G, and computed with respect to the global grid G, is the same traversal order induced by the hierarchical index I_B, computed locally with respect to B.

Property 2. The traversal of the elements at level l of the hierarchical index I_G is equivalent to a regular (non hierarchical) Z-order traversal with indices *(i,j,k)* having l-1 bits total.

Property 1 is explained by the fact that any Z-order curve built on G and restricted to B is equivalent to a Z-order built directly on B. Property 2 is illustrated in Figure 1, where the light gray vertices of level l are identical to the translation (along x and y alternatively) of the regular Z-order curve containing the dark gray vertices (those of levels up to l-1).

Property 1 allows each Data Source to loop through the data using the local I_B index. This means that we need to compute the mapping $I_B \rightarrow I_G$. This mapping can be computed efficiently using the indices (i_b,j_b,k_b) of the brick B within G. Note that i_b,j_b, and k_b have g-b bits each. To perform this transformation we take into account the level of resolution l of the vertex with respect to the index I_B. In particular, a vertex has level l if its index I_B satisfies the following inequality:

$$2^{l-1} \le I_B < 2^l.$$

We assume, by convention, that 2^{-1}=0. The first level l=0 has only the vertex with I_B=0. The global index for this vertex is analogous to expression (1):

$$I_G = S(2^{3(g-b)} + Z(i_b, j_b, k_b))$$

The second level l=1 has only one vertex with I_B=1. The global index for this vertex is

$$I_G = 2^{3(g-b)} + Z(i_b, j_b, k_b)$$

For the following levels, the mapping is

$$I_G = (2^{3(g-b)} + Z(i_b, j_b, k_b))2^{l-1} + (I_B - 2^{l-1})$$

Note that all the additions are between addresses that do not share common bits set to 1. Therefore they can be implemented as bitwise-`OR` operators. Similarly, the subtraction simply resets to 0 a bit that was set to 1.

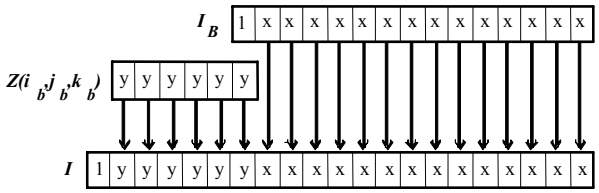

Figure 3: Mapping from the local index I_B to the global index I_G. The local brick has array index (i_b,j_b,k_b).

The diagram in Figure 3 shows the bitwise representation of this mapping from I_B to I_G. The fast evaluation of this mapping allows us to loop only through the data in B instead of the entire grid. The problem remains that the mapping from $I_B \rightarrow (i_b,j_b,k_b)$ is a slow operation. We optimize the loop based on I_B using property 2. At level l we use three auxiliary variables *(x,y,z)* that we increment, via a lookup table, following the regular Z-order for a grid of total size 2^{l-1}. The auxiliary variables *(x,y,z)* allow building the index (i_b,j_b,k_b) using these three rules, where h=l-1:

$h \bmod 3$	i_b	j_b	k_b
0	$x2^h + 2^{h-1}$	$y2^h$	$z2^h$
1	$z2^{h-1}$	$x2^h + 2^{h-1}$	$y2^h$
2	$y2^{h-1}$	$z2^{h-1}$	$x2^h + 2^{h-1}$

Again, these are simple `shift` operations with an eventual `or` operator to set an additional bit.

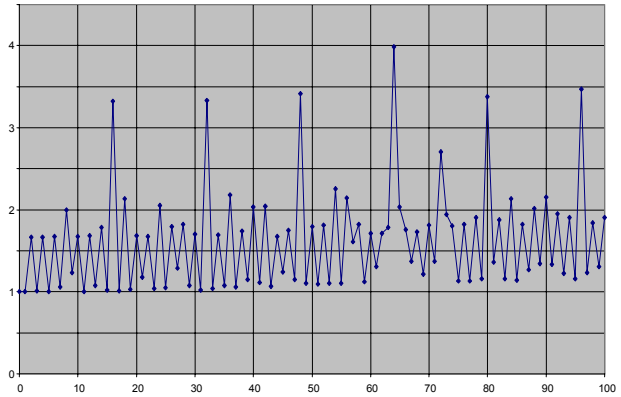

Figure 4. Load Balancing attained for repeated spatial queries with a different number of Data Servers. Each point in the chart plots the ration of the maximum load attained by any Data Server (that is the Data Server that finishes last) divided by the ideal load of the perfectly balanced execution. Smaller ratios mean better load balancing, with a minimum ratio of one.

In our implementation, we attained one order of magnitude speedup with the use of these expressions. On a brick of size 512^3 a loop through the data in I_B order with explicit computation of the inverse mapping $I_B \rightarrow (i_b,j_b,k_b)$ requires 67.7 seconds while the same loop with computation of (i_b,j_b,k_b) without explicit inverse mapping requires 5.5 seconds. Since the computation of one time step of the simulation typically takes a few minutes, we are able to maintain the time overhead at a level that does not appreciably affect the performance of the simulation

4.2 Data Server load balancing

We take a load balancing approach based on static data distribution [4, 19] where there is no data replication and no contention on a centralized agent that distributes tasks. The major challenge is to find a deterministic way to distribute data in a way that guarantees good load balancing for any request within a given set of queries. In [4] there is a study on how to load balance the iso-contouring query for any valid iso-value. Random data distribution [19] has the advantage of not being specialized for any particular query. For direct computation of iso-surfaces this approach works properly but for our type of spatial queries this approach would require large tables that track the actual locations of data elements.

As discussed in the previous section, we use expression (2) to determine the index of the Data Server that stores a given block of data. This technique allows us to distribute the data evenly across the Data Servers and achieves load balancing in the streaming from the Data Sources to the Data Servers. Unfortunately, this method does not guarantee load balancing in the real-time data access, when the Data Clients perform spatial queries (for example for visualization purposes). We have tested experimentally a large set of spatial queries to determine the load balancing achieved in practice.

Figure 4 summarizes the results of our experiments. As expected, the number of Data Servers has a major impact on the load balancing of the system. Even numbers of Data Servers seem to yield poor load balancing, with peaks denoting unbalanced distributions for multiples of 8 (in 2D this would be multiples of 4). This result seems to be due to the nature of the hierarchical indexing scheme that is based on sequences of shift operations, which are equivalent to multiplications by powers of 2. Interestingly, this bias remains even if we can deal with parallelepiped datasets of any aspect ratio using partially empty blocks whose overhead is removed by compression or their complete removal in the case of empty blocks. Symmetrically, most of the odd numbers of Data Servers yield a good load balancing. Most of the prime numbers yield a nearly ideal load balancing, as we would expect, but there is an exception to this rule. In practice, we select a target range of Data Servers using the chart in Figure 8 to select the candidate that would provide the best balancing.

5. EXPERIMENTAL RESULTS

Table 1: System performance for the real-time streaming of the electron density distribution from JEEP Data Servers.

Number of Data Sources	1	8	64
Number of Data Servers	1	3	3
Total Domain Size	128^3	256^3	512^3
Equivalent simulation time/time-step	270s	736s	4224s
Send Time	5.31s	7.03s	38.6s

We tested the performance of our system by creating a direct stream of data from a running simulation to a desktop workstation that visualizes one time step of the data while the simulation is computing the next one. Table 1 shows the performance results obtained for three experiments conducted under realistic conditions.

The Data Sources for our experiments are the compute nodes of JEEP [5], an *ab initio* molecular dynamics simulation code based on the plane wave method. To demonstrate the scalability of the system we have increased the resolution of the simulation by data replication. In particular, we report timings of the compute time of the simulation (equivalent simulation time) assuming linear scalability of the simulation code. Much slower running times would be attained if we where to increase the actual number of atoms being simulated. Streaming the entire electron density field after each time-step took the time Send Time of Table 1.

Three experiments were conducted with 1, 8, and 64 MPI processes respectively. JEEP was executed on an IBM SP system with four 332 MHz PowerPC 604e processors per node. The Data Servers were started on an SGI workstation with 4 194 MHz R10000 processors and 1280MB of memory. A rendering Data Client was instantiated on a desktop workstation to display the data received.

The Data Sources transmitted three bytes (RGB) for each sample of the electron density, resulting in 384MB per time step for the 512^3 volume. The total sustained streaming throughput was roughly 10MB/sec on a network connection on which we could measure a maximum ideal bandwidth of 20MB/sec with TCP sockets. This indicates that the transfer times are negligible with respect to the simulation times. Moreover, our reordering process is not the bottleneck in the data streaming process and higher speed networks would yield better streaming performance. However, for improved performance on standard networks, some type of data compression between the Data Sources and Data Servers will probably be required.

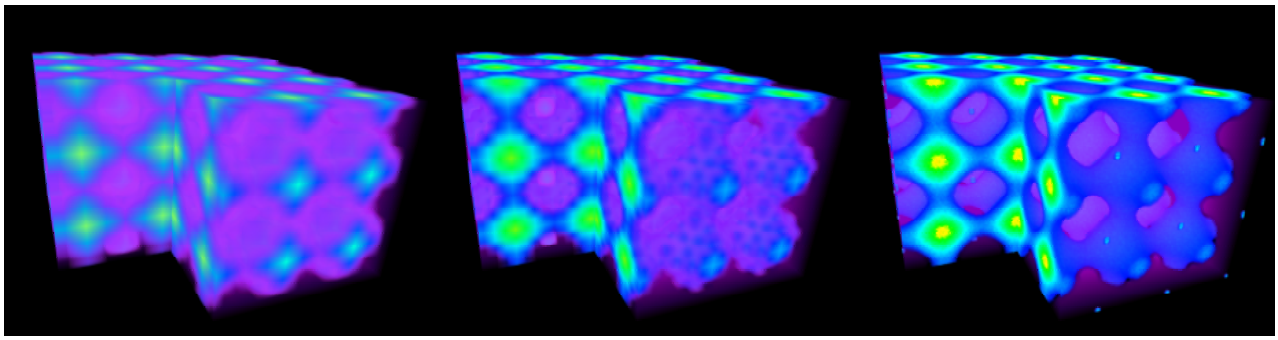

Figure 5 Progressive refinement (left to right) of the volume rendering of the electron density distribution.

Figure 5 illustrates the progressive rendering of the 512^3 data set. The total elapsed time to stream the data from the Data Server to the slicing client and render the image was 0.9 seconds for a 128^3 version and 1.6 seconds for a 256^3 version. Only 4.7 seconds were required to obtain the full 512^3 resolution data.

The progressive reordering of the data at the Data Clients provides additional flexibility in streaming strategy. Interrupting the data stream before the entire data set is transferred results in an approximated representation of the data that can still be used for rendering. In contrast, truncating the data stream in a classical single resolution data stream produces a nearly useless incomplete data set. Therefore, our hierarchical approach provides the simulation code with the added capability to determine the maximum transfer time considered acceptable, and request the Data Sources to truncate the data stream when unacceptable delays may be introduced.

6. CONCLUSIONS

We have presented a distributed framework for real-time streaming and visualization of large datasets generated by scientific simulations. We used progressive rendering algorithms and parallel/hierarchical data streaming techniques to reduce the latency between the simulation and the external software components like visualization and analysis tools. Our simple load balancing system enables scalability to arbitrary simulation sizes without introducing additional communication cost. We have applied our approach to a first-principles molecular dynamics code and showed that our system allows a user to monitor all the time steps of a large simulation with negligible time overhead.

7. REFERENCES

[1] G. Allen, W. Benger, T. Goodale, H.-C. Hege, G. Lanfermann, A. Merzky, T. Radke, E. Seidel and J. Shalf, "The Cactus Code:A Problem Solving Environment for the Grid", Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC9), Pittsburg, 2000.

[2] Abello, J., and Vitter, J.S., (eds.). External Memory Algorithms and Visualization. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society Press, Providence, RI, 1999.

[3] W. Bethel, "Visualization Dot Com", IEEE Computer Graphics and Applications, May/June 2000.

[4] C. Bajaj, V. Pascucci, D.Thompson, X.Y. Zhang Parallel "Accelerated Isocontouring for Out-Of-Core Visualization", In Proceedings of IEEE Parallel Visualization and Graphics Symposium, October 24-29,1999 San Francisco, CA, pp. 97 – 104

[5] J.L.Fattebert, and F. Gygi, "Density Functional Theory for Efficient Ab Initio Molecular Dynamics Simulations in Solution," J. Comput. Chem., in press (2002). Also available as Lawrence Livermore National Laboratory technical report UCRL-JC-143326, April 2001.

[6] R. Grosso, T. Ertl, J. Aschoff "Efficient Data Structures for Volume Rendering of Wavelet-Compressed Data", WSCG '96 - The Fourth International Conference in Central Europe on Computer Graphics and Visualization.

[7] B. Haimes, "pV3: A Distributed System for Large-Scale Unsteady CFD Visualization." AIAA Paper 94-0321, Reno NV, Jan. 1994.

[8] I. Ihm, S. Park, "Wavelet-based 3D compression scheme for very large volume data", In Proceedings of Graphics Interface '98, pages 107-116, Vancouver, June 1998.

[9] C. Johnson, S. Parker, and D. Weinstein "Large-scale Computational Science Applications using the SCIRun Problem Solving Environment", Supercomputer 2000.

[10] J.A. Kohl and P.M. Papadopulos. "A library for visualization and steering of distributed simulations using PVM and AVS." In V. Van Dongen, editor, Proceedings of the High Performance Computing Symposium, Montreal, Canada, pages 243–254, 1995.

[11] E.C. LaMar, B. Hamann, K. Joy, "Multiresolution Techniques for Interactive Texture-Based Volume Visualization" Proceeding of Visualization 1999.

[12] E. Lum, K.L. Ma, J. Clyne, J., "Texture Hardware Assisted Rendering of Time-Varying Volume Data," Proceedings of IEEE Visualization 2001 Conference, October 21-26, 2001.

[13] P. Lindstrom, and C.T. Silva A Memory Insensitive Technique for Large Model Simplification. IEEE Visualization 2001 Proceedings, pp. 121-126, 550, October 2001.

[14] M. Miller, C.D. Hansen, S.G. Parker, and C.R. Johnson. "Simulation Steering with SCIRun in a distributed memory environment." Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC-7), July, 1998.

[15] V.Pascucci and C.Bajaj, "Time Critical Adaptive Refinement and Smoothing" Proceedings of the ACM/IEEE Volume Visualization and Graphics Symposium 2000, Salt lake City, Utah, pg 33- 42.

[16] V. Pascucci and R.J.Frank "Global Static Indexing for Real-time Exploration of Very Large Regular Grids". In proceeding of 14th Annual Supercomputing conference, November 10-16, 2001, Denver, Co. On-line proceedings http://www.sc2001.org/techpaper.shtml

[17] Sagan, H., Space-Filling Curves. Springer-Verlag, New York, NY, 1994.

[18] M. Weiler, R. Westermann, C. Hansen, K. Zimmerman, T. Ertl, "Level-Of-Detail Volume Rendering via 3D Textures" IEEE Symposium on Volume Visualization '00.

[19] X. Zhang, C. Bajaj, and V. Ramachandran. "Parallel and Out-of-core View-dependent Isocontour Visualization Using Random Data Distribution". To appear in Joint Eurographics-IEEE TCVG Symposium on Visualization 2002