Available online at www.sciencedirect.com

SCIENCE DIRECT®

Parallel Computing 31 (2005) 185–204

PARALLEL COMPUTING

www.elsevier.com/locate/parco

A parallel multiresolution volume rendering algorithm for large data visualization ☆

Jinzhu Gao [a,1], Chaoli Wang [b,*], Liya Li [b], Han-Wei Shen [b]

[a] *Oak Ridge National Laboratory, One Bethel Valley Road, Bldg 5600, MS 6016, Oak Ridge, TN 37831, USA*

[b] *The Ohio State University, 395 Dreese Laboratories, 2015 Neil Avenue, Columbus, OH 43210, USA*

Received 16 February 2004; revised 20 December 2004
Available online 14 April 2005

Abstract

We present a new parallel multiresolution volume rendering algorithm for visualizing large data sets. Using the wavelet transform, the raw data is first converted to a multiresolution wavelet tree. To eliminate the data dependency between processors at run-time, and achieve load-balanced rendering, we design a novel algorithm to partition the tree and distribute the data along a hierarchical space-filling curve with error-guided bucketization. Further optimization is achieved by storing reconstructed data at pre-selected tree nodes for each processor based on the available storage resources to reduce the overall wavelet reconstruction cost. At run time, the wavelet tree is first traversed according to the user-specified error tolerance. Data blocks of different resolutions that satisfy the error tolerance are then decompressed and rendered to compose the final image in parallel. Experimental results showed that our algorithm can reduce the run-time communication cost to a minimum and ensure a well-balanced workload among processors when visualizing gigabytes of data with arbitrary error tolerances.

☆ An abbreviated version appears in Eurographics Symposium on Parallel Graphics and Visualization 2004 (EGPGV2004).

* Corresponding author.

E-mail addresses: gao.52@osu.edu (J. Gao), wang.726@osu.edu (C. Wang), li.530@osu.edu (L. Li), hwshen@cse.ohio-state.edu (H.-W. Shen).

[1] Research supported in part by the Mathematics, Information and Computational Sciences Office, Office of Advanced Scientific Computing Research, US Department of Energy, under contract no. DE-AC05-00R22725 with UT-Battelle, LLC.

0167-8191/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2005.02.005

Keywords: Large scale data; Multiresolution; Wavelet transform; Space-filling curve; Load balancing; Volume rendering; Parallel rendering

1. Introduction

An increasing number of scientific applications are now generating high resolution three-dimensional data sets on a regular basis. The sizes of those data sets are often so large that it is almost impossible to perform interactive data analysis using only a single PC or workstation. Take the time-dependent Richtmyer–Meshkov turbulence simulation [18] as an example, at each time step the simulation produced about 7.5 gigabytes of data defined on a $2048 \times 2048 \times 1920$ rectilinear grid. Not surprisingly, data of this scale cannot be handled easily by a single machine with limited computational resources. A viable solution to address this challenge is to utilize a cluster of PCs to distribute the data and perform the computation and rendering in parallel.

As visualization is an iterative and exploratory process, rendering a lower resolution of data sometimes is sufficient for the user to get a quick overview before querying further details in selected regions. Given the physical limitation in the current generation of display devices, it is also not always desirable to render the entire data set at the finest resolution considering that the projection of such a large data set can exceed the highest screen resolution currently available. For this reason, many visualization algorithms now provide the user with the ability to perform multiresolution rendering for interactive and adaptive data navigation.

In this paper, we present a parallel algorithm for multiresolution volume rendering. Although researchers previously have proposed various techniques for multiresolution encoding and rendering of large scale volumes on a single graphics workstation [9,15,27], fewer studies were focused on designing parallel algorithms for such a purpose using PC clusters. Several issues need to be addressed in order to achieve efficiency and scalability when parallel multiresolution volume rendering is performed. One is the issue of designing an effective data distribution scheme that can minimize both space and run-time computation overheads for storing and reconstructing the multiresolution volumes. When hierarchical encoding schemes such as the wavelet transform is used, the multiresolution data are often represented in the form of a hierarchical tree [9]. Obviously, the sheer size of the data prohibits the replication of the entire tree in every processor so it is necessary to partition and distribute the multiresolution hierarchy. Since there is often a great deal of dependency among the parent and child nodes in the data hierarchy, it is critical to design an efficient partitioning and distribution algorithm to minimize such dependency and thus reduce the run-time inter-processor communication cost for data reconstruction. Another issue that needs to be addressed is load balancing. At run-time, when the user specifies arbitrary error tolerance to visualize the volume, different spatial subvolumes of various levels of detail will be reconstructed, which could cause uneven rendering workloads among the processors. It is important to design an effective workload distribution scheme, so that the rendering subtasks can be evenly distributed among the processors for any given error tolerance used to traverse the multiresolution data hierarchy. It is also crucial for the workload distribution algorithm to work hand-in-hand with the data distribution scheme so as to avoid expensive data redistribution at run-time.

In our algorithm, we exploit the wavelet transform to convert the data into a hierarchical multiresolution representation, called a *wavelet tree*. To alleviate the long chains of parent–child node dependencies when reconstructing volumes of different resolutions, the wavelet tree is partitioned into *distribution units* in a way that no data dependency exists between processors. To balance the volume rendering workload without run-time data redistribution, we utilize a scheme based on *hierarchical space-filling curves* and *error-guided bucketization* to distribute the data and the rendering tasks. Moreover, to minimize the run-time wavelet reconstruction cost, we employ an effective greedy algorithm which utilizes the additional disk space allowed at each processor to store the reconstructed data at selective nodes in the distribution units.

The remainder of the paper is organized as follows. First, we review related work in Section 2. From Sections 3–7, we describe our parallel multiresolution volume rendering algorithm, including the construction of the wavelet tree with hierarchical error metric calculation, data distribution with error-guided bucketization, greedy selection of nodes to store reconstructed data, and run-time parallel multiresolution volume rendering. Results on multiresolution rendering and load balancing among different processors are given in Section 8 and the paper is concluded in Section 9 with an outline of future work for our research.

2. Related work

Having the ability to visualize data at different resolutions allows the user to identify features in different scales, and to balance image quality and computation speed. Along this direction, a number of techniques have been introduced to provide hierarchical data representations for volume data. Burt and Adelson [2] proposed the *Laplacian Pyramid* as a compact hierarchical image code. This technique was extended to 3D by Ghavamnia and Yang [8] and applied to volumetric data. Their Laplacian pyramid is constructed using a Gaussian low-pass filter and encoded by uniform quantization. Voxel values are reconstructed at run-time by traversing the pyramid bottom up. To reduce the high reconstruction overhead, they suggested a cache data structure. LaMar et al. [15] proposed an octree-based hierarchy for volume rendering where the octree nodes store volume blocks resampled to a fixed resolution and rendered using 3D texture hardware. A similar technique was introduced by Boada et al. [1]. Their hierarchical texture memory representation and management policy benefits nearly homogeneous regions and regions of lower interest.

Wavelets are used to represent functions hierarchically, and have gained increasing popularity in several areas of computer graphics [25]. Over the past decade, many wavelet transform and compression schemes have been applied to volumetric data. Muraki [19,20] introduced the idea of using the wavelet transform to obtain a unique shape description of an object, where 2D wavelet transform is extended to 3D and applied to eliminate wavelet coefficients of lower importance. The use of a single 3D [12,13] or multiple 2D [21] Haar wavelet transformations for large 3D volume data has been well studied, resulting in high compression ratios with fast random access of data at run-time. More recently, Guthe et al. [9] presented a hierarchical wavelet representation for large volume data sets that supports interactive walkthrough using a single commodity PC. Only the levels of detail necessary for display are extracted and sent to texture hardware for viewing.

Parallel computing has been widely used in large volume visualization. Hansen and Hinker [10] proposed a parallel algorithm on SIMD machines to speed up isosurface extraction. Ellsiepen [5] introduced a parallel implementation for unstructured isosurface extraction with a dynamical block distribution scheme. Crossno and Angel [4] devised an isosurface extraction algorithm using particle systems and its parallel implementation. A parallel isosurface extraction algorithm based on span space subdivisions was described in [24]. To speed up the volume rendering process, Ma et al. [17] proposed a parallel algorithm that distributes data evenly to the available computing resources and produces the final image using binary-swap compositing. Schulze and Lang [23] provided a parallelized version of perspective shear-warp volume rendering algorithm [14]. A scalable volume rendering technique was presented in [16], utilizing lossy compression to render time-varying scalar data sets interactively. To further reduce the rendering time of large-scale data sets, several parallel visualization algorithms [6,7,11] with visibility culling were introduced to render only visible portion of a data set in parallel.

Balancing the workload among the processors is always a key issue in a parallel implementation. In [3], Campbell et al. showed a load-balanced technique using the space-filling curve [22] traversal. In this method, the spatial locality preserved by a space-filling curve was utilized to balance the workload. Gao et al. [7] also showed that, even after visibility culling, the parallel volume rendering algorithm can still achieve well-balanced workload by distributing volume blocks to processors along a space-filling curve.

3. Algorithm overview

Our parallel multiresolution volume rendering algorithm consists of two stages: preprocessing and run-time rendering. In the preprocessing stage, we first construct a hierarchical wavelet tree and then compress the wavelet coefficients using a combination of run-length and Huffman encoding. Coupled with the construction of the wavelet tree, a hierarchical error metric is used to calculate the approximation error for each of the tree nodes, which will be used to control the tradeoff between image quality and rendering speed at run-time. This error metric can be rapidly computed, and also guarantees that the error value of a parent node will be greater than or equal to those of its eight child nodes. The data blocks associated with the wavelet tree nodes are then distributed among different processors along a hierarchical space-filling curve with an error-guided bucketization scheme to ensure load balancing. In the case that additional disk space at each processor can be allocated, certain tree nodes in the distribution units assigned to each processor are selected to store the reconstructed data to improve the run-time data reconstruction time.

At run time, our parallel multiresolution volume rendering algorithm is performed according to a user-specified error tolerance. The wavelet tree is first traversed front to back to identify the nodes with varied resolutions that satisfy the error tolerance. Then, the wavelet-compressed data associated with those nodes are decompressed and the data blocks are reconstructed on the fly. Finally, the processors render the selected data blocks of various levels of detail in parallel. The final image is generated by compositing the partial images rendered at different processors.

In the following, we describe each stage of our algorithm in detail. We first present the multiresolution wavelet tree construction algorithm and the error metric. Then, we discuss our data distribution scheme for the purpose of minimizing the dependency among processors and ensuring run-time load balancing. Finally, we introduce our greedy optimization algorithm that selects nodes from distribution units assigned to each processor to store the reconstructed data. Implementation details about our parallel volume rendering will follow.

4. Wavelet tree construction with hierarchical error metric calculation

Our hierarchical wavelet tree construction algorithm is similar to the methods described in [9,26], where a bottom-up blockwise wavelet transform and compression scheme is used. The algorithm starts with subdividing the original three-dimensional data into a sequence of blocks. We assume each raw volume block has n voxels. Without loss of generality, we also assume $n = 2^i \times 2^j \times 2^k$, where i, j, k are all integers and greater than zero. These raw volume blocks form the leaf nodes of the wavelet tree. After performing a 3D wavelet transform to each block, a low-pass filtered subblock of size $n/8$ and wavelet coefficients of size $7n/8$ are produced. The low-pass filtered subblocks from eight adjacent leaf nodes in the wavelet tree are then collected and grouped into a single block of n voxels, which will become the lower resolution data block represented by the parent node in the wavelet hierarchy. We recursively apply this 3D wavelet transform and subblock grouping process until the root of the tree is reached, where a single block of size n is used to represent the entire volume. As we arrive at the root of the wavelet tree, since the root node has no parent, no 3D wavelet transform is performed. To save space and time for the wavelet tree construction, unnecessary wavelet transform computation could be avoided by checking the uniformity of the data block. If the data block is uniform, we can skip the 3D wavelet transform process and set the low-pass filtered subblock to that uniform value and all its corresponding wavelet coefficients to zero.

To reduce the size of the coefficients stored in the wavelet tree, the wavelet coefficients associated with a tree node resulting from the 3D wavelet transform will be compared against a user-provided threshold and set to zero if they are smaller than the threshold. These wavelet coefficients are then compressed using run-length encoding combined with a fixed Huffman encoder [9]. This bit-level run-length encoding scheme exhibits good compression ratio if many consecutive zero subsequences are present in the wavelet coefficient sequence and is very fast to decompress. The compressed bit stream is saved into an individual file.

At run-time, a data block at a certain resolution is reconstructed as follows: the low-pass filtered subblock of size $n/8$ is first retrieved from its parent node. This may entail a sequence of recursive requests of the low resolution data blocks associated with its ancestor nodes. Reconstructions will be performed in those nodes if necessary. The wavelet coefficients of size $7n/8$ are obtained by decoding the corresponding bit stream. Finally, we group the low-pass filtered subblock and the wavelet coefficients and then apply an inverse 3D wavelet transform to reconstruct the data block.

Coupled with the construction of the wavelet tree, an error value is calculated at every tree node. Our error metric is based on the mean square error (MSE) calculation. As shown in Fig. 1, let us assume that the current wavelet tree node in question is S, the ith child node of S is S_i, $i \in [0,7]$, and the data block of n voxels associated with S that approximates the original subvolume V is B. One way to calculate the error metric is to compute the MSE between the low resolution data block B and the corresponding raw data in subvolume V using the following formula:

$$E = \frac{\left(\sum_{(x,y,z)\in V} (v(x,y,z) - f(x,y,z))^2\right)}{m}$$

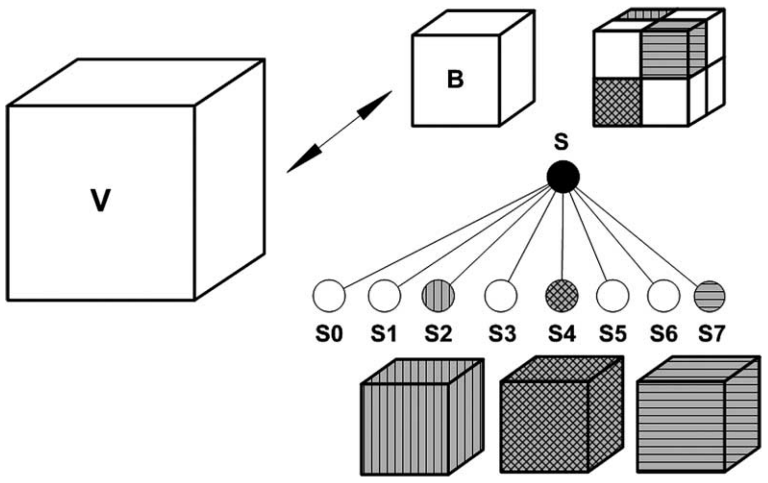

Fig. 1. Calculating the error metric of a wavelet tree node S. B is the low resolution data block associated with S, representing the raw data subvolume V. The three nodes and their associated data blocks (drawn with same pattern) are examples used to illustrate the data relationship of the parent node S and it child nodes S_i, where $0 \leqslant i \leqslant 7$.

where $v(x, y, z)$ is the original scalar data value at location (x, y, z) in V. $f(x, y, z)$ is the interpolated data value at its corresponding position in B. m is the total number of voxels in V. The interpolation function for obtaining the approximated data value can be either nearest neighbors or linear. For any wavelet tree leaf node, we define $E = 0$.

The main drawback of calculating the error metric this way is that when the underlying data set is large, it can be very slow to perform the error computation. The MSE calculation will become progressively more expensive as we traverse toward the wavelet tree root since the size of the volume covered by a tree node will increase proportionally. Furthermore, a large I/O overhead is involved because the computation requires the raw data as well as the approximated data.

To overcome these problems, we propose a much faster way to calculate the error metric which considers the MSE between the data in a parent node and the data in its eight immediate child nodes, taking the maximum error value of the child nodes into account. We compute the error in a bottom-up manner as follows:

$$E = \frac{\sum_{i=0}^{7}\left(\sum_{(x,y,z)\in B}(b_i(x,y,z) - f(x,y,z))^2\right)}{8n} + \max E$$

where $b_i(x, y, z)$ is the data value at location (x, y, z) in the data block associated with S_i. $f(x, y, z)$ is the interpolated data value at its corresponding position in B. $\max E$ is the maximum error of S_i, where $0 \leqslant i \leqslant 7$. Again, the interpolation function for getting the approximated data value can be either nearest neighbors or linear. Essentially, the error E of a parent node S is calculated by adding $\max E$ to the MSE between the eight data blocks associated with the child nodes S_i and their corresponding low-pass filtered data in B. A nice feature of this error metric is that it guarantees that the error value of any parent node is greater than or equal to those of its corresponding eight child nodes. For any wavelet tree leaf node, we define $E = 0$.

5. Hierarchical data distribution with error-guided bucketization

For large scale data sets, the resulting wavelet hierarchy needs to be partitioned and distributed among the processors since it is impractical to replicate the data everywhere. To ensure the scalability of the parallel algorithm, it is important that the partitioning result will minimize the dependency among the processors and ensure a balanced workload. In this section, we describe our data distribution and load balancing algorithm in detail.

One of the primary issues to be addressed when designing the data distribution scheme is to minimize the dependency among the processors. In the wavelet tree structure mentioned above, there exist long chains of parent–child node dependencies—a node needs to recursively request the low-pass filtered subblocks from its ancestor nodes in order to reconstruct its own data. When nodes with such dependencies are assigned to different processors, expensive communications at run-time become inevitable. To eliminate such dependency among processors, we design the following *EVERY-K* storage strategy to arrange the multiresolution data blocks in the wavelet tree. Instead of having the leaf and intermediate nodes store the wavelet coefficients, and only the root node store the low resolution data block, we reconstruct and store low resolution data blocks in advance for nodes at every k levels starting from the root, where $k < h$, and h is the height of the wavelet tree. (In practice, h may not be an exact multiple of k and this can be easily handled.) We call a node that stores the reconstructed data block a *representative node*, while a node that stores only wavelet coefficients an *associated node*. By default, the root of the wavelet tree is a representative node and all the leaf nodes of the tree are associated nodes. Fig. 2 shows an example of such schemes where $k = 2$ and $h = 6$. It is clear that data reconstruction only needs to be performed for the associated nodes by recursively requesting their parent nodes up to the closest representative node, where the low resolution data have already been reconstructed. We define a *distribution unit* as the data at a representative node along with the wavelet coefficients at all its descendent nodes which depend on the representative node. This definition implies that there must be one and only one representative node in a distribution unit and all the nodes in one distribution unit are independent of nodes in any other distribution units. We use the distribution units to form a *partition* of the wavelet tree, and a distribution unit is used as the *minimum* unit to be assigned to a processor. Since there is no data dependency among distribution units during wavelet reconstruction, we are able to eliminate the dependency among processors at run-time.

An optimal data distribution scheme should ensure that all the processors have an equal amount of rendering workload at run-time. However, when multiresolution rendering is performed, different data resolutions, and thus different rendering

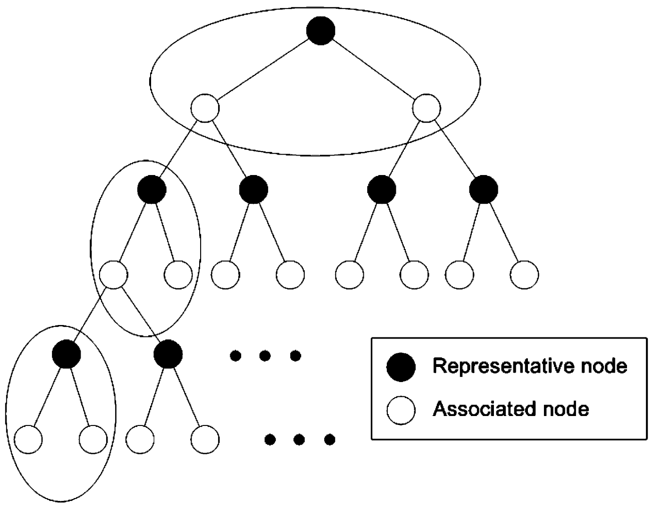

Fig. 2. Only the nodes at every k levels starting from the root (drawn in black) store the data blocks. The ellipsis show partitions of the distribution units. In the figure, $k = 2$ and $h = 6$. A binary tree rather than an octree tree is drawn here for illustration purpose only.

workload, will be chosen to approximate the local regions. This makes the workload distribution task more complicated. In the following, we describe a static load distribution scheme to solve the load balancing problem.

In general, a volumetric data set usually exhibits strong spatial coherence. Given an error tolerance, if a particular data resolution is chosen for a subvolume, it is more likely that a similar resolution will also be used for the neighboring subvolumes. In our multiresolution algorithm, this means if a block at a certain level is selected to be rendered, it is most likely that its neighboring regions will also be rendered from the blocks in the same tree level. Thus, if neighboring data blocks at a similar resolution are evenly distributed to different processors, each processor will receive approximately the same rendering workload in that local neighborhood. Based on this idea, a space-filling curve [22] is utilized in our data distribution scheme to assign the distribution units to different processors. The space-filling curve is used for its ability to preserve spatial locality, i.e., the traversal path along a space-filling curve always visits the adjacent blocks before it leaves the local neighborhood. The hierarchical property of a space-filling curve also makes it suitable to be applied to a hierarchical algorithm. In Fig. 3, we give a simplified 2D example of a wavelet

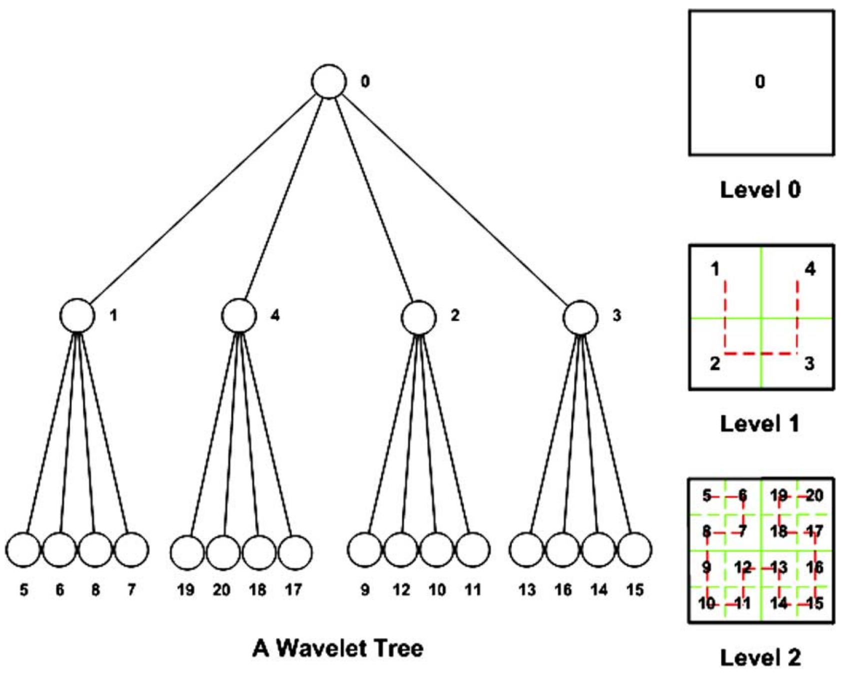

Fig. 3. A simplified 2D example of data distribution along the hierarchical space-filling curve. All the wavelet tree nodes are traversed level by level in a breadth-first search manner. The numbers associated with the tree nodes indicate the traversal order given by the space-filling curve. A popular space-filling curve, the Hilbert curve, is used in this example.

tree and its corresponding space-filling curve at each level. The numbers in the figure show the traversal order along the hierarchal space-filling curve.

To ensure load balancing at run-time under different error tolerances, data blocks with similar error values should be distributed to different processors since our error-guided wavelet tree traversal algorithm usually selects them together for rendering. To achieve this, in addition to the hierarchal space-filling curve traversal, we include an error-guided bucketization mechanism in our data distribution scheme. As illustrated in Fig. 4, our algorithm works as follows: The whole error range [*errmin*, *errmax*] is first partitioned into several error intervals, where *errmin* and *errmax* are the minimum and maximum error values of the representative nodes from all the distribution units. Then, we traverse along a hierarchical space-filling curve, where every distribution unit encountered is sorted, according to the traversal order, into a bucket whose error range covers the error of its representative node. The intervals of the buckets will be adjusted so that each bucket holds similar number of distribution units. Finally, all sorted distribution units in each of the buckets are distributed among processors in a round-robin manner.

Utilizing our hierarchical error-guided data distribution scheme, neighboring distribution units with similar errors will be distributed to different processors. As demonstrated in Section 8, our error-guided hierarchical data distribution scheme can achieve well-balanced workload among processors.

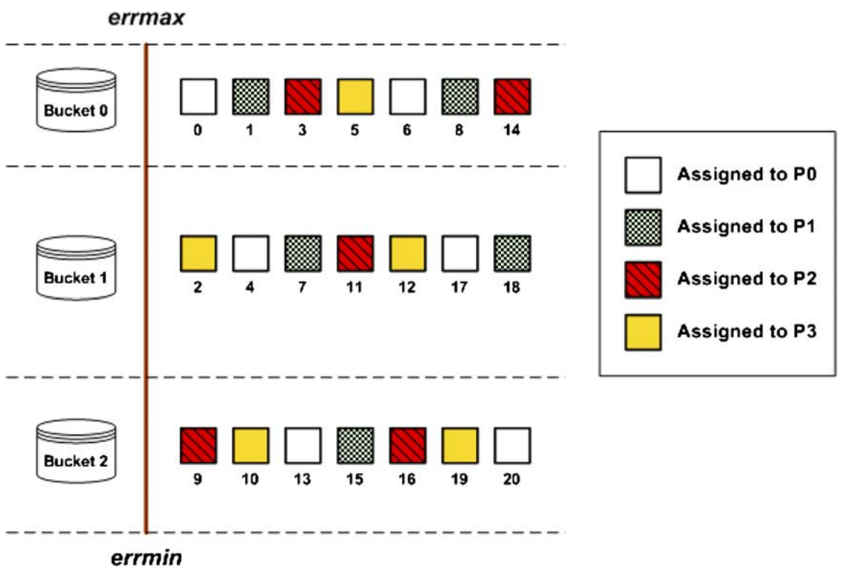

Fig. 4. An example of data distribution along the hierarchical space-filling curve with error-guided bucketization. The numbers associated with the distribution units shown in the figure indicate the traversal order given by the space-filling curve. In this example, a total of 21 distribution units with three different resolutions are distributed among four processors.

6. Reconstructed data storage with greedy optimization

According to the EVERY-K scheme described in Section 5, only data at the representative node of each distribution unit are pre-reconstructed. This means most of the nodes assigned to a processor store compressed wavelet coefficients. Reconstruction of those nodes can be expensive, and hence will slow down the overall run-time rendering performance. In the case when there is additional storage space available in each of the processors, it can be beneficial to reconstruct and store the low resolution data in more tree nodes to trade space for run-time efficiency.

Given that the available storage space can be limited, the selection of nodes from the distribution units for storing the reconstructed data is primarily based on the benefit of those nodes, that is, how much the nodes can contribute to the reduction of the reconstruction time if their wavelet coefficients are decompressed and reconstructed in advance. The amount of reconstructed data that can be stored is subject to the additional storage space that the user is willing to commit in each processor. To determine which tree nodes should have their data pre-reconstructed, we define two functions, *Benefit* and *Cost*. *Benefit* estimates the degree of usefulness when keeping reconstructed data at a node, and *Cost* evaluates the amount of the additional storage space required to keep the reconstructed data rather than the compressed wavelet coefficients. The goal can be then stated as:

$$\begin{aligned}&\text{maximize}: \sum_{\omega\in\Omega} \mathit{Benefit}(\omega)\\ &\text{subject to}: \sum_{\omega\in\Omega} \mathit{Cost}(\omega) \leqslant \Delta\end{aligned}$$

where Ω is the set of nodes selected to keep the reconstructed data. Δ is the additional storage space the user is allowed to use in each processor.

We propose a greedy algorithm to tackle this optimization problem by repeatedly selecting a node i with the highest benefit until there is no more storage space available. Specifically, our greedy algorithm assigns each node a benefit value, and selects the one with the highest benefit at each iteration by traversing the wavelet tree nodes in all distribution units assigned to a processor. Every time when a node is selected to store the reconstructed data, the benefit values for the remaining nodes in its corresponding distribution unit need to be updated. Finding the exact optimal solution for this optimization problem is actually NP-complete, which prompted us to adopt an approximate greedy solution.

We define the benefit $B(i)$ for each node i by considering the visiting likelihood L, the penalty factor P, and the quality of reconstruction R, which can be written as:

$$B(i) = L(i) \times P(i) \times R(i)$$

We now need to define $L(i)$, $P(i)$, and $R(i)$.

$L(i)$: This function approximates the likelihood of a node being visited. A node will be visited more often when (1) it is closer to the root of the distribution unit since the reconstructions of data in its descendant nodes will depend on it; (2) it is not a uniform node, hence contains more information. We characterize $L(i)$ by the error calculated for the node, that is:

$$L(i) = E(i)$$

where $E(i)$ is the error value of node i. The value $L(i)$ is normalized before being used in function $B(i)$ to calculate the benefit.

$P(i)$: For a node i, when a large portion of the nodes in its distribution unit have already been chosen to store the reconstructed data, it is less beneficial to select this node. This is because an uneven distribution of the additional resources among the distribution units will lead to uneven speedups when different error tolerances are used for traversing the multiresolution tree, a cause for load imbalance. To avoid this problem, we prevent this clustering effect by adding a penalty factor into the benefit measure:

$$P(i) = p^{\lambda}$$

where $p \in (0.0, 1.0)$ is the penalty heuristic and λ is the number of nodes already selected to store reconstructed data in node i's distribution unit.

$R(i)$: This is used to estimate the contribution to the reduction of overall reconstruction time by allocating additional disk space to keep the reconstructed data for node i. The following formula is used:

$$R(i) = \frac{T'(i) - T(i)}{S(i) - S'(i)}$$

where $S(i)$ is the storage space that node i will consume to store the reconstructed data if it is selected, and $S'(i)$ is the storage space that node i will need to store wavelet coefficients when it is not selected. $T(i)$ is the time to load the reconstructed data if node i is selected, while $T'(i)$ is the time to load and reconstruct the data if node i is not selected.

Following the notations defined in Table 1, $T(i)$ is computed as:

$$T(i) = T_1(i)$$

And $T'(i)$ is computed as:

$$T'(i) = T_2(i) + T_3(i) + T_4(i)$$

Table 1
Notations used to calculate $T(i)$ and $T'(i)$ in the calculation of $R(i)$ for determining the benefit function, $B(i)$

Time	Description
T_1	Load the file stream of the reconstructed data from disk
T_2	Load the compressed file stream of the wavelet coefficients from disk, and perform decoding to get the coefficients, proportional to the size of the stream
T_3	Retrieve the low-pass filtered subblock from the ancestor nodes recursively
T_4	Compose the low-pass filtered subblock and wavelet coefficients into one block, and perform a 3D inverse wavelet transform to get the reconstructed data
T_5	Extract the low-pass filtered subblock from a data block

where $T_3(i)$ depends on the current selecting status of the parent node j of node i, and is calculated as:

$$T_3 = \begin{cases} T(j) + T_5(i) & j \in \Omega \\ T'(j) + T_5(i) & j \notin \Omega \end{cases}$$

The calculation of the reconstruction time of all the descendant nodes of node i in its distribution unit depends on whether node i is selected or not. After the algorithm selects a node to store reconstructed data, the benefit values of the remaining nodes in the same distribution unit need to be recomputed. The total number of nodes that can be selected is limited by the additional storage space Δ allocated at the processor.

7. Parallel multiresolution volume rendering

Before rendering each frame, the wavelet tree is traversed if the viewing parameter or the error tolerance has been changed. This could be done either by the host processor which in turn broadcasts the traversal result to all the other processors, or by all processors simultaneously (each processor only needs to have a copy of the wavelet tree skeleton with error at each node regardless of whether it actually has been assigned the data block or not), obviating the communication among the processors. Our error-guided tree traversal algorithm allows the user to specify an error tolerance as the stopping criterion so that regions having smaller errors can be rendered at their lower resolutions. The nodes in the wavelet tree are recursively visited in the front-to-back order according to the viewing direction and a series of subvolumes with different resolutions that satisfy the error tolerance are identified. If the data blocks associated with those selected subvolumes have not been reconstructed, we need to perform reconstruction before the actual rendering begins. Our wavelet tree partition and data distribution scheme ensures that the reconstruction dependencies could only exist within a distribution unit. This will reduce the overall reconstruction time since the cost of retrieving the low-pass filtered subblock is bounded by the height of the subtree corresponding to a distribution unit, which is usually a small number, two or three in our experiments.

During the actual rendering, each processor only renders the data blocks preassigned to it during the data distribution stage, so that there is no expensive data redistribution between processors. The screen projection of the entire volume's bounding box is partitioned into smaller tiles with the same size, where the number of the tiles equals the number of processors. Each processor is assigned one tile and is responsible for the composition of the final image for that tile. Each time a processor finishes rendering one data block, the resulting partial image is sent to those processors whose tiles overlap with the block's screen projection. After rendering all the data blocks, the partial images received at each processor are composited together to generate the final image for its assigned tile. Finally, the host processor collects the partial image tiles and creates the final image. Because we perform data distribution along the hierarchical space-filling curve and image space partition for the image composition, each processor renders a similar number of data blocks and composites a similar number of image tiles, which ensures a well-balanced workload among processors.

As we may anticipate reusing most of the reconstructed data blocks for subsequent viewings due to the spatial locality and coherence exploited by the wavelet tree structure, it is desirable to cache the data blocks that have already been reconstructed during the rendering for better performance. The user can predefine a fixed amount of disk space and memory dedicated for the caching purpose. Upon requesting a data block for the rendering, we retrieve its data from the memory, provided the block is cached in the main memory. Otherwise, we need to load the data from the disk if the reconstructed data block is on disk. If it is neither cached in memory nor on disk, then we need to reconstruct the data block and load it into the main memory. When the system runs short of the available storage for caching the reconstructed blocks, our replacement scheme will swap out a data block that has been visited least often.

8. Results

In this section, we present the experimental results of our parallel multiresolution volume rendering algorithm running on a PC cluster that consists of 32 2.4 GHz Pentium 4 processors connected by Dolphin Networks. The test data set was the 7.5 GB $2048 \times 2048 \times 1920$ Richtmyer–Meshkov Instability (RMI) data set from Lawrence Livermore National Laboratory.

The dimensions of the leaf node blocks in our wavelet tree were set to be $128 \times 128 \times 64$, or 1 MB in the total size. This is a tradeoff between the cost of performing the wavelet transform for a single data block, and the rendering and communication overheads for final image generation. Since each voxel value is represented using a single byte, Haar wavelet transform with a lifting scheme was used to construct the data hierarchy for simplicity and efficiency reasons. A lossless compression scheme was used with the threshold set to zero. We considered one voxel overlapping boundaries between neighboring blocks in each dimension when loading data from original brick data files in order to produce correct rendering results. The wavelet tree we constructed has a depth of six with 10,499 non-empty nodes.

For comparison, we chose two schemes to partition the wavelet tree and form the distribution units. The first one is the EVERY-2 scheme *without* greedy optimization, and the second one is the EVERY-2 scheme *with* greedy optimization. The construction of the wavelet tree was performed on a 2.0 GHz Intel Pentium 4 processor with 1 GB main memory. It took about an hour to complete and the compressed data size was around 2.65 GB. The data associated with the wavelet tree nodes were distributed using the hierarchical data distribution scheme with error-guided bucketization described in Section 5. The space-filling curve used in our implementation was the Hilbert curve. For both schemes, we allocated 256 MB *temporary* disk space and 128 MB main memory at each processor for run-time caching of the reconstructed data blocks.

For the EVERY-2 scheme with greedy optimization, in order to achieve better load balancing, we took into account the actual non-empty nodes in the distribution units when distributing the data along the hierarchical space-filling curves. A processor would not receive any more distribution units if the total number of non-empty nodes in the distribution units already assigned to it had exceeded the average limit (the total number of non-empty nodes in a wavelet tree divided by the total number of processors). We performed benchmark tests to obtain the constants (decompression speed, T1, T4, and T5) for the performance model presented in our greedy selection algorithm. The results are shown in Table 2. The average decompression speed for the run-length and fixed Huffman decoding was around 6.542 MB/s, which included disk file I/O. For every node i, the benchmark results were used to calculate the timing of $T_2(i)$ and $T_3(i)$. The penalty heuristic p in the benefit function $B(i)$ was set to 0.8. When testing our greedy selection algorithm, rather than specifying exact *permanent* storage sizes for each processor, the space allocated to store the additional reconstructed data was set to different percentages of the total non-empty nodes in each processor in order to have more intuitive comparisons.

The error-guided hierarchical data distribution allowed our parallel multiresolution volume rendering algorithm to effectively balance the workload. Fig. 5 shows the number of data blocks distributed to each of the 32 processors for the EVERY-2 scheme. Fig. 6 shows the number of data blocks rendered at each of the 32 processors when three different error tolerances were used. For the EVERY-2 scheme with greedy optimization, Fig. 7 shows the number of data blocks reconstructed at each of the 32 processors when three different error tolerances were used. Since the processors reconstructed/rendered approximately an equal number of blocks, it can be seen that good rendering load-balancing was achieved. Fig. 8 shows the number of data blocks reconstructed at each of the 32 processors for the EVERY-2 scheme with greedy optimization under error tolerance of 1000. Three different percentages of non-empty nodes were selected to store the reconstructed data at each processor. Table 3 shows the corresponding timing results using software raycasting and the time spent on each of the stages. As we can observe, when we chose a higher percentage of nodes to store the reconstructed data, the time to perform wavelet reconstruction decreased accordingly. This demonstrates the effectiveness of our greedy optimization algorithm in trading additional storage space for the wavelet reconstruction time. The well-balanced workload implies that our parallel algorithm is highly scalable. For the EVERY-2 scheme with greedy optimization under error tolerance 1000, when 40% of non-empty nodes were selected to store the reconstructed data, it took 840.838 s total time to render the data on a single processor. (We actually

Table 2
The benchmark test results

Speed	T_1	T_4	T_5
6.542 MB/s	0.0286 s	0.3563 s	0.0018 s

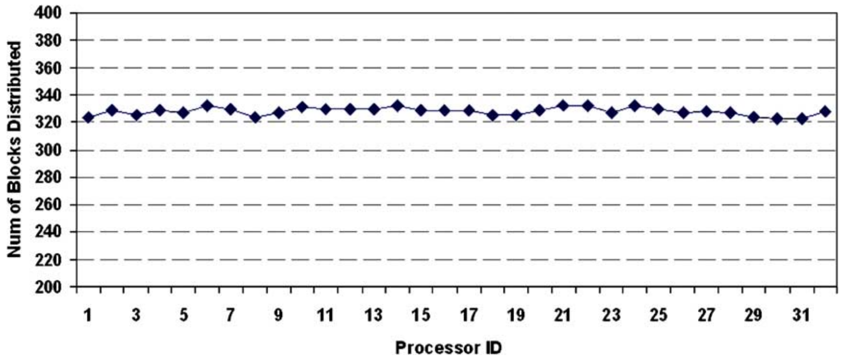

Fig. 5. The number of data blocks distributed to each of the 32 processors for the EVERY-2 scheme.

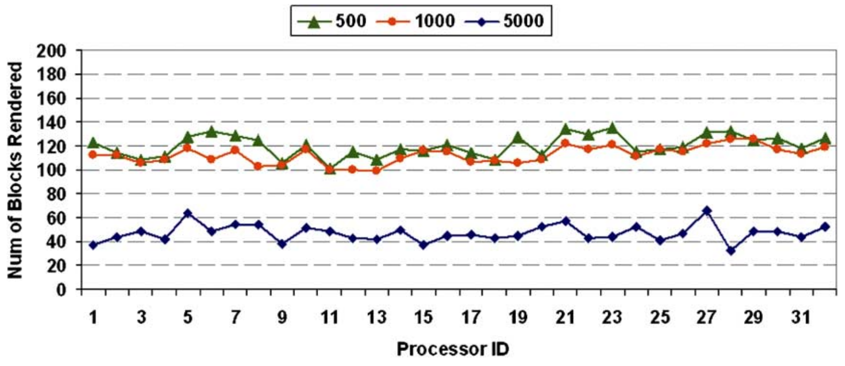

Fig. 6. The number of data blocks rendered at each of the 32 processors for the EVERY-2 scheme under three different error tolerances. A total of 1510, 3602, and 3850 blocks were rendered for error tolerances of 5000, 1000, and 500 respectively.

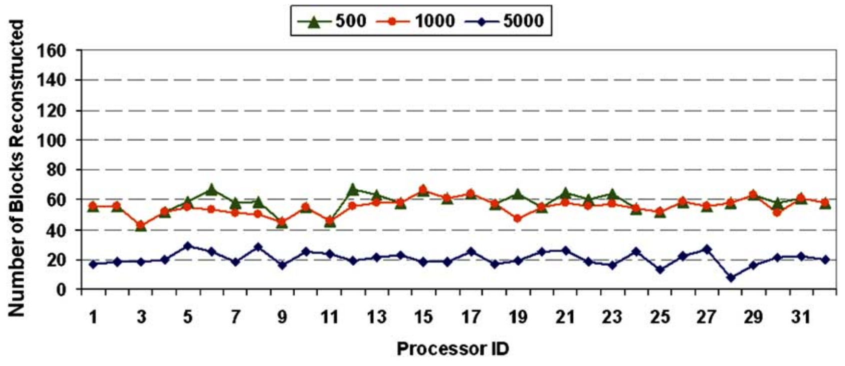

Fig. 7. The number of data blocks reconstructed at each of the 32 processors for the EVERY-2 scheme with greedy optimization under three different error tolerances. In all the three cases, 20% of non-empty nodes were selected to store the reconstructed data at each processor. A total of 657, 1767, and 1859 blocks were reconstructed for error tolerances of 5000, 1000, and 500 respectively.

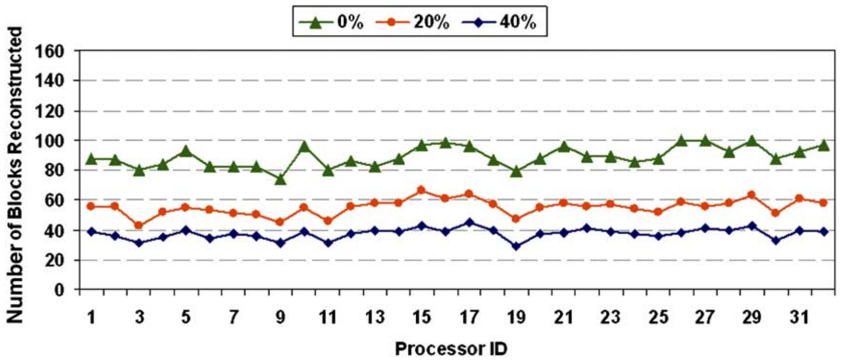

Fig. 8. The number of data blocks reconstructed at each of the 32 processors for the EVERY-2 scheme with greedy optimization under error tolerance of 1000. Three different percentages of non-empty nodes were selected to store the reconstructed data at each processor. A total of 1203, 1767, and 2845 blocks were reconstructed for percentages of 40%, 20%, and 0% respectively.

Table 3
The timing for the EVERY-2 scheme with greedy optimization under error tolerance 1000

Percentage	Total (s)	Reconstruction (s)	Raycasting (s)	Overhead (s)
0%	53.021	31.466	15.603	5.952
20%	35.430	21.567	12.486	1.377
40%	27.701	14.527	11.413	1.761

40%, 20% and 0% of non-empty nodes were selected to store the reconstructed data at each of the 32 processors. The overhead included the initialization, image composition and communication overhead.

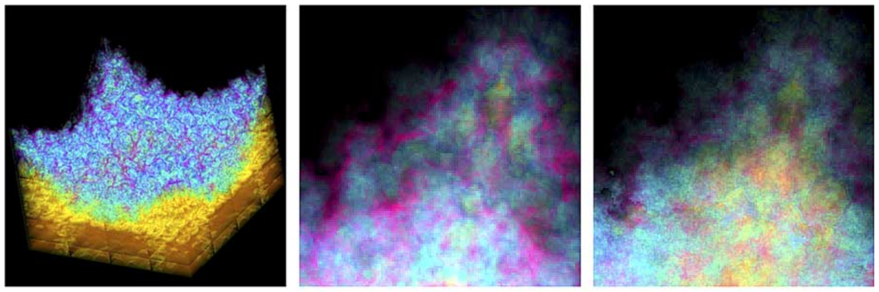

Fig. 9. Multiresolution rendering of the RMI data set. The resolution of the output images is 512 × 512. Image (a) shows a rendering of the data. Images (b) (E = 40,000, 21 blocks rendered) and (c) (E = 30,000, 120 blocks rendered) were zoomed-in views using different error tolerances with the same viewing setting. As can be seen, the lower the error tolerance was, the more delicate details of the data were revealed.

increased the main memory size for caching to 1 GB for the single processor. If we still use 128 MB main memory, the total rendering time was 898.303 s with many more data blocks shuffling between memory and disk.) Our algorithm can achieve approximately 94.86% parallel CPU utilization, or a speedup of 30.36 times for 32 processors.

Fig. 9 shows several results with different levels of detail for the RMI data set rendered using software raycasting. When the error tolerance became higher, data of lower resolutions were used, which resulted in a smaller number of blocks being rendered. It can be seen that, although more delicate details of the data were revealed when reducing the error tolerance, images of reasonable quality can still be obtained at their lower resolutions. The use of wavelet-based compression also allowed us to produce images of good visual quality with much smaller space commitment.

9. Conclusion and future work

We have presented an efficient parallel multiresolution volume rendering algorithm. A multiresolution wavelet tree is used to allow for interactive analysis of large data and flexible run-time tradeoff between image quality and rendering speed. To ensure the algorithm's scalability, we propose a unique tree partitioning and distribution algorithm, and utilize a hierarchical space-filling curve with error-guided bucketization scheme to eliminate the parent–child node wavelet reconstruction dependencies, balance the rendering workload, and minimize the run-time communication overhead. A greedy optimization algorithm is introduced to store reconstructed data at selective nodes for each processor as a further speedup. The experimental results demonstrate the effectiveness and utility of our parallel algorithm. Future work includes utilizing graphics hardware to perform wavelet reconstruction and rendering, and extending our parallel multiresolution volume rendering algorithm to large-scale time-varying data.

Acknowledgments

This work was supported by NSF ITR grant ACI-0325934, DOE Early Career Principal Investigator Award DE-FG02-03ER25572, and NSF Career Award CCF-0346883. The RMI data set was provided by Mark Duchaineau at Lawrence Livermore National Laboratory. Special thanks to Jack Dongarra, Jian Huang and Clay England from The University of Tennessee, and Don Stredney and Dennis Sessanna from Ohio Supercomputer Center for providing the test environment.

References

[1] I. Boada, I. Navazo, R. Scopigno, Multiresolution volume visualization with a texture-based octree, The Visual Computer 17 (3) (2001) 185–197.

[2] P.J. Burt, E.H. Adelson, The Laplacian pyramid as a compact image code, IEEE Transactions on Communications 31 (4) (1983) 532–540.
[3] P.C. Campbell, K.D. Devine, J.E. Flaherty, L.G. Gervasio, J.D. Teresco, Dynamic octree load balancing using space-filling curves, Tech. Rep. CS-03-01, Williams College Department of Computer Science, 2003.
[4] P. Crossno, E. Angel, Isosurface extraction using particle systems, in: Proceedings of IEEE Visualization'97, 1997, pp. 495–498.
[5] P. Ellsiepen, Parallel isosurfacing in large unstructed datasets, in: Visualization in Scientific Computing, 1995, pp. pp. 9–23.
[6] J. Gao, H.W. Shen, Parallel view-dependent isosurface extraction using multi-pass occlusion culling, in: Proceedings of IEEE Symposium in Parallel and Large Data Visualization and Graphics'01, 2001, pp. pp. 67–74.
[7] J. Gao, J. Huang, H.W. Shen, J.A. Kohl, Visibility culling using plenoptic opacity functions for large data visualization, in: Proceedings of IEEE Visualization'03, 2003, pp. 341–348.
[8] M.H. Ghavamnia, X.D. Yang, Direct rendering of Laplacian pyramid compressed volume data, in: Proceedings of IEEE Visualization'95, 1995, pp. 192–199.
[9] S. Guthe, M. Wand, J. Gonser, W. Straßer, Interactive rendering of large volume data sets, in: Proceedings of IEEE Visualization'02, 2002, pp. 53–60.
[10] C. Hansen, P. Hinker, Massively parallel isosurface extraction, in: Proceedings of IEEE Visualization'92, 1992, pp. 189–195.
[11] J. Huang, N. Shareef, R. Crawfis, P. Sadayappan, K. Mueller, A parallel splatting algorithm with occlusion culling, in: Proceedings of Eurographics Workshop on Parallel Graphics and Visualization'00, 2000, pp. 125–132.
[12] I. Ihm, S. Park, Wavelet-based 3D compression scheme for very large volume data, in: Proceedings of Graphics Interface'98, 1998, pp. 107–116.
[13] T.Y. Kim, Y.G. Shin, An efficient wavelet-based compression method for volume rendering, in: Proceedings of Pacific Graphics'99, 1999, pp. 147–157.
[14] P. Lacroute, L. Marc, Fast volume rendering using a shear-warp factorization of the viewing transformation, in: Proceedings of ACM SIGGRAPH'94, 1994, pp. 451–458.
[15] E. LaMar, B. Hamann, K.I. Joy, Multiresolution techniques for interactive texture-based volume visualization, in: Proceedings of IEEE Visualization'99, 1999, pp. 355–362.
[16] E. Lum, K.L. Ma, J. Clyne, A hardware-assisted scalable solution for interactive volume rendering of time-varying data, IEEE Transactions on Visualization and Computer Graphics 8 (3) (2002) 286–301.
[17] K.L. Ma, J.S. Painter, C.D. Hansen, M.F. Krogh, Parallel volume rendering using binary-swap compositing, IEEE Computer Graphics and Applications 14 (4) (1994) 59–68.
[18] A.A. Mirin, R.H. Cohen, B.C. Curtis, W.P. Dannevik, A.M. Dimits, M.A. Duchaineau, D.E. Eliason, D.R. Schikore, S.E. Anderson, D.H. Porter, P.R. Woodward, L.J. Shieh, S.W. White, Very high resolution simulation of compressible turbulence on the IBM-SP system, in: Proceedings of ACM/IEEE Supercomputing Conference'99, 1999.
[19] S. Muraki, Approximation and rendering of volume data using wavelet transforms, in: Proceedings of IEEE Visualization'92, 1992, pp. 21–28.
[20] S. Muraki, Volume data and wavelet transforms, IEEE Computer Graphics and Applications 13 (4) (1993) 50–56.
[21] F.F. Rodler, Wavelet-based 3D compression with fast random access for very large volume data, in: Proceedings of Pacific Graphics'99, 1999, pp. 108–117.
[22] H. Sagan, Space-filling Curves, Springer-Verlag, New York, 1994.
[23] P. Schulze, U. Lang, The parallelized perspective shear-warp algorithm for volume rendering, Parallel Computing 29 (3) (2003) 339–354.
[24] H.W. Shen, C.D. Hansen, Y. Livnat, C.R. Johnson, Isosurfacing in span space with utmost efficiency (ISSUE), in: Proceedings of IEEE Visualization'96, 1996, pp. 287–294.
[25] E.J. Stollnitz, T.D. DeRose, D.H. Salesin, Wavelets for Computer Graphics: Theory and Applications, Morgan Kaufmann, 1996.

[26] C. Wang, H.W. Shen, A framework for rendering large time-varying data using wavelet-based time-space partitioning (WTSP) tree, Tech. Rep. OSU-CISRC-1/04-TR05, Department of Computer and Information Science, The Ohio State University, January 2004.

[27] Y. Zhou, B. Chen, A. Kaufman, Multiresolution tetrahedral framework for visualizing regular volume data, in: Proceedings of IEEE Visualization'97, 1997, pp. 135–142.