COMPUTER GRAPHICS forum

DOI: 10.1111/j.1467-8659.2008.01307.x

COMPUTER GRAPHICS forum
Volume 28 (2009), number 1 pp. 141–153

Out-of-Core and Dynamic Programming for Data Distribution on a Volume Visualization Cluster

S. Frank and A. Kaufman

Department of Computer Science, Stony Brook University, Stony Brook, NY, USA

Abstract

Ray directed volume-rendering algorithms are well suited for parallel implementation in a distributed cluster environment. For distributed ray casting, the scene must be partitioned between nodes for good load balancing, and a strict view-dependent priority order is required for image composition. In this paper, we define the load balanced network distribution (LBND) *problem and map it to the NP-complete precedence constrained job-shop scheduling problem. We introduce a kd-tree solution and a dynamic programming solution. To process a massive data set, either a parallel or an out-of-core approach is required. Parallel preprocessing is performed by render nodes on data, which are allocated using a static data structure. Volumetric data sets often contain a large portion of voxels that will never be rendered, or empty space. Parallel preprocessing fails to take advantage of this. Our* slab-projection slice, *introduced in this paper, tracks empty space across consecutive slices of data to reduce the amount of data distributed and rendered. It is used to facilitate out-of-core bricking and kd-tree partitioning. Load balancing using each of our approaches is compared with traditional methods using several segmented regions of the Visible Korean data set.*

Keywords: Distributed visualised load balancing, partitioning, volume visualization

ACM CCS: I.3.2 [Computer Graphics]: Distributed/network graphics; C.2.4 [Distributed Systems]: Distributed applications

1. Introduction

As systems, memory and data sets continue to grow, so does the difficulty of managing these resources. Numerous applications require rendering of massive data sets. Examples include geophysical data, high-resolution X-ray computed tomography (HRXCT) data and colour photographic data sets. Interactive visualization of seismic data allows geophysicists to plan oil hole drilling. HRXCT is a new imaging approach with a resolution in the tens of microns. The accuracy of HRXCT is similar to the destructive method of sectioning, which is sometimes used to study fossils, teeth and bones. Studies have shown that the high resolution of HRXCT is necessary for the accurate reconstruction of bone to correctly quantify structural parameters [FRK01]. Photographic axial-aligned serial sections of specimens enable interactive examination of data sets, with realistic colouring for medical education and surgical planning applications.

The purpose of this paper is to address the following specific problems encountered when rendering massive volumetric data sets. The full data set may not fit in the memory of a single render node so that data must be distributed, rather than replicated on each render node. To have a scalable system that efficiently utilizes the available resources, a load balancing scheme is needed to ensure that all render nodes perform an equitable amount of work during rendering. In addition, data typically do not fit in main memory; so, all preprocessing must use either out-of-core or distributed methods. Finally, the data size may exceed total rendering hardware memory available across all rendering nodes, requiring multiple rendering passes.

For many volumetric data sets, there is a large portion of empty space or voxels that will never be rendered. We discuss techniques which leverage empty space that is always 'empty', such as air or suspension fluids, rather than empty space which results from a transfer function setting. We have

Submitted January 2008
Revised August 2008
Accepted August 2008

rendered several segmented regions of the Visible Korean Human [PCH*06] on the Stony Brook Visualization Cluster. Segmented data is used to illustrate our algorithms with various empty space distributions; yet it does not preclude the use of transfer functions.

In this paper, we introduce the *slab-projection slice*, an orthographic projection of slice information. Unless otherwise specified, the word slice refers to a raw data slice, which is orthogonal to the z-axis. Each slab-projection slice voxel counts non-empty voxels in the slices represented. Later preprocessing steps avoid re-reading data slices by using information encoded in a series of slab-projection slices during a single pass through the raw data.

We introduce the problem of partitioning volumetric data for distribution across a cluster to achieve good load balancing for parallel ray casting. We call this the *load balanced network distribution (LBND)* problem. We describe a mapping of the LBND problem to the precedence constrained job-shop scheduling problem, which is known to be NP-complete [GJ79]. The goal of the LBND problem is to minimize end-to-end render time in a distributed rendering system within resource and priority order constraints. The input is a volumetric data set, along with render and network cost information. The precedence order is defined as the relative distance of each voxel with respect to a given view direction.

Due to the wide variety and the frequency of changes in visualization system hardware, we are motivated to find a solution to the LBND that can adapt to new hardware as it is introduced. This is achieved by defining an appropriate cost function and constraints to reflect the end-to-end rendering on a cluster. Dynamic programming (DP) optimizes a cost function while meeting a set of constraints. We present two solutions to the LBND problem that are particulary applicable to scenes with large portions of unevenly distributed empty space. The first, the slab-projection slice kd-tree (*skd-tree*), uses non-empty voxel information collected in a series of slab-projection slices. The second, *brick grouping* inputs a directed acyclic graph (DAG) of data bricks and finds an optimal partition with respect to the given cost model using DP. Our *out-of-core bricking* creates the DAG of data bricks, where the DAG represents a view-dependent brick precedence order.

The specific contributions of this paper are

- Definition of the LBND problem;
- Mapping of LBND to job-shop scheduling;
- Slab-projection slice and skd-tree;
- Out-of-core bricking and brick grouping.

This paper is organized as follows. In Section 2, we present related work. Distributed parallel ray casting is described in Section 3. We also define the LBND problem, along with a mapping to job-shop scheduling. An overview of our data management pipeline is given in Section 4. We also introduce our slab-projection slice and skd-tree partitioning. Our brick grouping algorithm and out-of-core bricking are presented in Section 5. Results are presented in Section 6.

2. Related Work

In parallel volume rendering, the scene is typically subdivided using an acceleration structure such as a grid, octree or kd-tree. Lombeyda *et al.* [LMS*01] have shown the arithmetic equivalence of a single ray-casting composite computation and the combined result of a set of (smaller) composite computations. They demonstrate that the alpha compositing operator is associative, which means portions of the scene can be rendered individually and composited together, as long as the overall priority order or relative distance to the viewpoint is respected. In the VG-cluster system [MLM*03], special purpose hardware composites images from eight PCs. These interim images are composited to produce the final image. In the HP MDS Visualization cluster, a series of images are composited for each frame by composite hardware, HP Sepia-2a, located on each render-node. The Sepia-2a [MHS99] composites a local image with one received through DVI acquisition from another node. Compositing is also performed on a GPU [GPKM03].

Volume rendering on graphics cards (GPUs) has become increasingly popular [EKE01; KW03; MWMS07]. Shader programs allow complex problems to be solved on a GPU, as long as the underlying problem can be reformulated to fit the single instruction, multiple data paradigm. However, most complex applications, including texture-mapped volume rendering, exceed the resource capacity of even the most up-to-date GPU hardware. The solution is to partition the problem into multiple passes. Multipass rendering refers to the process of using the same hardware to render an image for each of several different *bricks* of data, where a brick is a portion of the volume which is sized to fit the rendering hardware.

The multipass partitioning problem (MPP) [CNS*02] is an instance of job-shop scheduling [RLV*04]. Several solutions to the MPP [CNS*02; RLV*04; Hei05] have been proposed. Each of these algorithms evaluates the cost function of a proposed GPU pass by generating the code for that pass. Heirich [Hei05] proposed the DP algorithm for MPP (DPMPP). The input to the MPP is a valid shader program in the form of a DAG and costs of each operation and GPU pass. The output is a schedule of DAG operations, partitioned into passes that minimize the total runtime cost, where the schedule observes the precedence relations of the DAG, and no pass exceeds the physical resource constraints of the GPU. Although in theory this algorithm is intractable, Heirich demonstrates the potential to achieve a reasonable run-time with a DP solution to the MPP by restricting the branching factor. Our

DP partitioning is based on the DPMPP solution; however, DPMPP cannot be directly applied to the LBND problem. Although these problems map to the same underlying problem, their size, scope and cost functions differ significantly. For example, the input to LBND includes a DAG of bricks, whereas DPMPP processes a DAG of GPU kernel programs and texture accesses.

Acceleration data structures are used in ray casting for empty space skipping [PSL*98; LMK03; WIK*06] and to divide the scene among processing units [Cha02]. Optimized octrees have been successfully used for multipass rendering task scheduling [Kno06, KWPH06]. Optimized octree-based rendering techniques that skip empty space directly on the GPU can be used for local GPU rendering pass scheduling in conjunction with our methods. However, the goal of the LBND problem is different in that the cost of moving data between nodes and internode image composition is more significant. The smallest resolution cell is sized to fit the target rendering hardware because data bricks are available in local disk memory. In contrast, for network distribution, only a subset of bricks is available to each node, and the partition generally should minimize internode communication. If a static partition is used, the granularity required to force all local images to have consecutive depth priorities is typically orders of magnitude larger than multipass partitioning cells. As a result, the rendering assignments are not evenly allocated among resources if the empty space distribution is not uniform.

Recent research on dynamic partitioning [MMD06, MSE06] has improved load balancing for interactive viewing of large data sets. Müller *et al.* [MSE06] present a parallel visualization system that uses graphics hardware accelerator (GPU) rendering and kd-tree space subdivision, augmented by dynamic re-balancing of data. Marchesin *et al.* [MMD06] leverage interframe coherence to dynamically distribute data, based on the load from the previous frame. In our DP solution, render-node assignments are dynamically adjusted, as the viewpoint changes. Data for each view-dependent assignment for a render node is stored in local disk memory; so, no data is transferred between nodes during rendering.

3. Distributed Volume Rendering Overview

An overview of our distributed ray cast rendering is shown in Figure 1. A volumetric data set in this context consists of a 3D grid of information at sample locations or voxels. The information is either a scalar value, such as density, or a vector, such as colour in a photographic data set. Volumetric ray casting is a technique in which one or more rays emanates for each pixel and each ray accumulates the colour and opacity contribution of a series of voxels along the ray in the volume data. Our visualization cluster is a network of PC nodes, connected with a high-speed back end network for fast sharing of partial images, as well as a front end network for process communication and control.

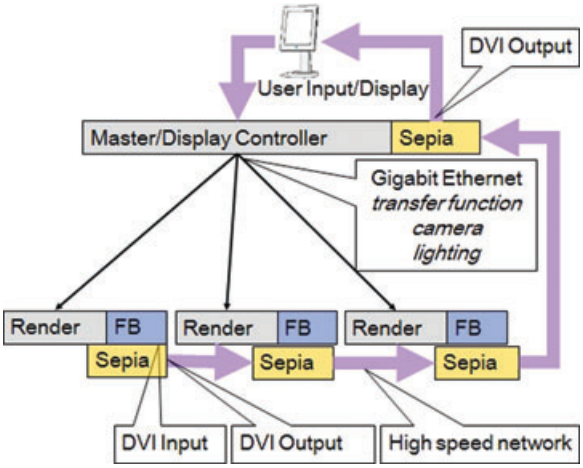


Figure 1: *Distributed ray cast rendering block diagram for a sort-last architecture.*

Interactive rendering is done in parallel, as viewing parameters are parsed and distributed by the master node. Rendering is performed on individual render-nodes by VolumePro1000 hardware or a GPU, and the final image for each frame is a composite of the resulting images. The alpha blending equation used for image compositing requires the relative depth of each image. The visibility ordering is unique for orthogonal projections along one of the octants of a cube centred at the origin; images are composited in the depth order for the current view direction. Both the GPU and VolumePro1000 use parallel, orthographic projection, ray cast rendering; perspective rendering requires more complex image alignment and is not used in our system.

Internode image composition is controlled by the master node but takes place locally on render nodes (either in the GPU or in Sepia-2a compositing hardware). The rendering time on each node for a given image size is roughly proportionate to the size of volume rendered. Other factors include sampling rate and the cache coherency of the volume. Early ray termination is programmed into the rendering hardware. However, taking full advantage of this requires that data be redistributed between nodes as the opacity changes, and the network transfer cost is relatively high in our system; so, we don't move data during rendering.

3.1. Load balanced network distribution problem

For a given scene of volumetric data and a given system configuration, our goal is to distribute data across the system to optimize available rendering resources. The problem input includes a scene with volumetric data, a description of the distributed system configuration, including the number of render-nodes, rendering and local composition costs, network transfer costs and the memory capacity of each render-node.

We use multipass rendering within each render node; the same hardware is used to render an image for each of several different bricks of data, each in a separate pass. Images within the same render node having consecutive depth priorities are composited into a single image, prior to internode composition. The final image must wait for every rendering pass to finish on each render node; so, end-to-end rendering time is a function of the slowest renderer plus internode composition time. For every image requiring internode composition, additional network communication is required.

3.2. Problem mapping

The input to the LBND is a volumetric data set, along with render and network cost information. A volumetric data set consists of a grid of voxels, each defining a scalar or vector field. The precedence constraints between voxels are defined by the relative distance to the viewpoint along the view direction. The output is a render-node assignment, which minimizes the total runtime cost, does not violate any physical resource constraint of the system and observes the precedence order for image composition. The LBND problem is an instance of the job-shop scheduling problem, as demonstrated by the following mapping.

The input to the job shop scheduling problem is a list of jobs, with associated resource requirements, and a DAG of job dependences. The goal of the problem is to schedule each job to a shop, or resource, and minimize the overall job completion time, without violating resource or job dependency constraints. Data voxels are jobs, rendering memory is the limited resource required for each job, and the precedence constraints are given in a DAG which is derived from the relative distance of voxels to the viewpoint along the view direction. The goal is to create a scene partition, or schedule, which minimizes end-to-end rendering time, or job completion time.

Although the underlying structure of these problems are the same, there are several differences between them. For example, the cost calculation used in DPMPP is based on resource usage of compiled shader code; this is measured as part of the optimization decision in DPMPP. Partition costs are directly calculated from the DAG of bricks in our LBND solution. The cost function for end-to-end render time depends on the time of the slowest renderer in LBND, a function of the maximum number of bricks assigned to a render-node. The cost function for DPMPP is a function of the sum of *operations*, which reflects resource use in total across the whole shader program.

4. Out-of-Core Data Management

We propose out-of-core preprocessing to minimize the number of times a piece of data is read from disk memory. A volume is defined as massive when it is one or more orders of magnitude larger than the size of main memory available in a single visualization engine. Massive data require either out-of-core (external memory) or distributed preprocessing. For distributed preprocessing data must be replicated on render nodes and moved again if non-static data distribution is used. Static data distribution doesn't achieve good load-balancing for data sets with large variations in sparsity because partitioning is not guided by the distribution of empty space throughout the data. Another drawback to distributed preprocessing is that empty space is unnecessarily sent to render nodes. Figure 2 illustrates that, even without replication of slices to multiply overlapping render nodes, an order of magnitude is saved by cropping the Visible Korean data set prior to distribution compared with sending full slices.

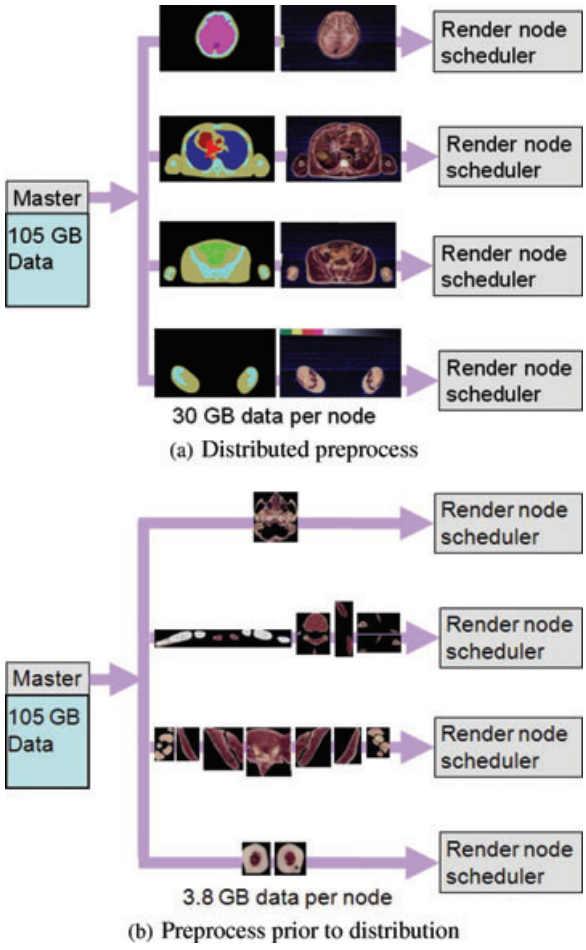


Figure 2: *Distributed preprocessing compared with preprocessing on a single node: (a) Full slices distributed prior to preprocessing; (b) Cropped slices distributed after preprocessing. Data are reduced by nearly an order of magnitude, using empty space removal.*

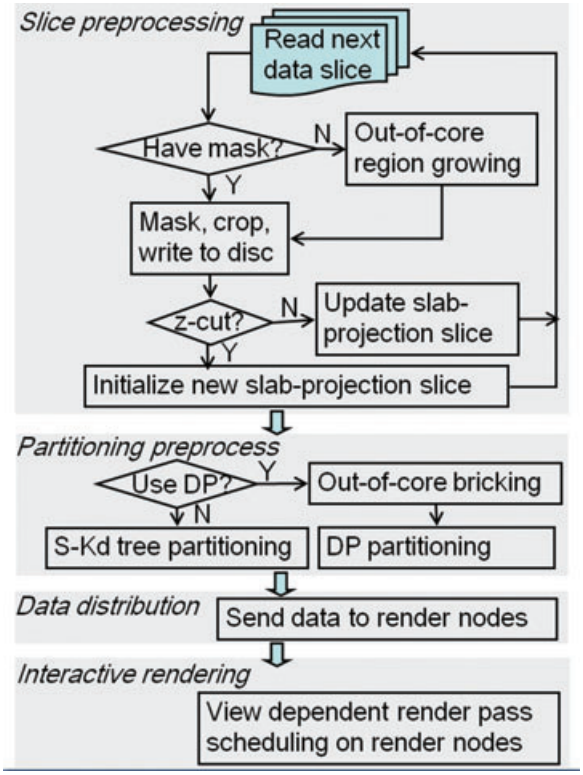


Figure 3: *Distributed rendering data management pipeline.*

4.1. Data management pipeline

Our data management pipeline is shown in Figure 3. The preprocessing consists of the slice preprocessing phase, followed by the data distribution process. Slice preprocessing is used for creating cropped slices to be distributed across the network for rendering. Each raw data slice is read once during this phase. Non-empty regions are segmented using a mask volume, then cropped and written to disk memory. Out-of-core region growing is used if no mask has been provided with the data set. Concurrently, data extent information is consolidated into a series of slab-projection slices (Section 4.2). Render pass scheduling, hardware ray casting and compositing all take place on the render nodes during rendering.

4.2. Slab-projection slice

A series of slab-projection slices are created during the initial preprocessing step of masking and cropping. These are used to avoid any further raw data reads during the data distribution preprocessing. We gather a consecutive series of slices, a slab, into an orthographic projection of slice data information, the slab-projection slice. Slices are masked, cropped and written to disk memory for use during rendering. The number of non-empty voxels within the current slab of data is stored in the corresponding x–y position of the slab-projection slice. This maintains a count of non-empty voxels for the current slab.

A *z-cut* is a cut across the z-axis. If the z-length of the slab is larger than a given slab minimum and either a split or merge is detected, or if the z-length is equal to a given slab maximum, then a z-cut is made. The concepts of a split and a merge are introduced in [FK05]. A split occurs when a slice contains two distinct non-empty regions that both overlap a single region in the current slab. A merge occurs when a single region in the slice overlaps two or more regions in the slab. Whenever a z-cut is made, a new slab-projection slice is initialized using the current slice.

We use a series of slab-projection slices for out-of-core preprocessing. The first step is to gather data extent information from slices. These are used either directly for skd-tree partitioning and brick grouping DP algorithms (see Section 4.3) or to find the input brick DAG for DP partitioning.

Data distribution uses either skd-tree partitioning and brick grouping algorithm or DP partitioning. For DP partitioning, a separate partition is derived for each viewing octant by running DP with a DAG, representing the corresponding visibility ordering. Data bricks are replicated as needed across the rendering system, rather than moving data across the network during rendering. The portion of data assigned to a render node for each unique partition is stored in local disk memory. A kd-tree partition has the advantage of giving a view-independent solution, but has several disadvantages for massive data sets. We must restrict the number of render nodes to a power of two, or either allow some render nodes have more data assigned than others using a non-balanced kd-tree or by assigning more than one leaf node to a render node.

4.3. Slab-projection Kd-tree partitioning

A traditional kd-tree is obtained through recursive splitting of data using empty space information. Each recursion requires one or more scans of every data slice. To avoid these memory accesses during the partitioning process, we introduce the skd-tree partitioning algorithm.

The input to each recursion of the kd-tree partitioning includes a set of volume data slices, current subscene Q, the current depth of the recursion d and a cut plane. The centre of a subscene is the plane that splits the non-empty voxels into nearly equal numbers across the cut axis. At each recursion, the subscene is split with the centre cut plane, and then each side is recursively partitioned. The kd-tree root represents the whole scene, and the depth is initialized to 0. The recursion ends when the number of leafs at the current level is the same as the number of render nodes, B. This requires the number of render nodes to be a power of two. The final depth of the kd-tree is $b_d = \log_2 n$.

We use the number of non-empty voxels as the criteria for finding the centre of a subscene. The set of data slices that overlaps the current subscene is S. For each voxel position along the cut axis there is a data slice, s, that is orthogonal to the current cut axis. The set of these cut crossing slices is Q, and $Q = S$ for the z-cut axis. The number of non-empty voxels in slice i is $i.voxelCount$, where i is either a data slice or a cut crossing slice.

For each recursion of the partitioning algorithm, every data slice, s, that overlaps S, is examined. If it is not in main memory it is read from disk memory. Traditional kd-tree partitioning proceeds recursively as follows:

$\forall s \in S$, read s and scan to get $s.voxelCount$
  Add $s.voxelCount$ to subscene total
  $\forall\, q \in Q$ update $q.voxelCount$ from intersection with s
    Find centre cut plane using voxel count array
    if $d < b_d$ then find kd-tree for each side of cut

The problem with this approach is that every slice in the current scene needs to be read to determine the splitting axis. Instead, we use a series of slab-projection slices in the skd-tree partitioning algorithm. The skd-tree partitioning solution proceeds in the same way as a traditional kd-tree except that a series of slab-projection slices provides non-empty voxel information instead of raw data slices. The slab-projection slices are produced during the first step of preprocessing, as described in Section 4.2.

At the start of the algorithm, all slab-projection slices are assumed to be in main memory. The minimum slab width in the slab-projection slice preprocessing step is used to insure that this criteria is met. This circumvents the need to read any data from disk memory during kd-tree partitioning. The main advantage of using slab-projection slices is that empty space information is consolidated prior to the partitioning process. This has a major impact on the time spent in determining the splitting plane at each iteration of the recursion.

4.4. Out-of-core bricking

Out-of-core bricking finds a DAG of bricks using the series of slab-projection slices obtained in the first step of preprocessing. The brick precedence order, with respect to a specific viewing vector, is stored in a DAG along with brick bounding box extents.

For each slab-projection slice, we find the set of candidate-blocks or cropped, non-empty regions. From these candidates we cut bricks of a predetermined size. For GPU rendering, the bricks should be sized to fit the maximum texture memory and satisfy any hardware-specific dimension requirements. However, bricks of this size may be too small to control the algorithm run time for DP partitioning. In this case, larger bricks are used for data distribution, and these are cut to the target GPU texture size prior to rendering. We use bounding boxes of bricks, not physical data, in our brick grouping DP algorithm. At render time, these bounding boxes are used to create volume texture bricks, sized to fit the rendering hardware, from pre-cropped slices.

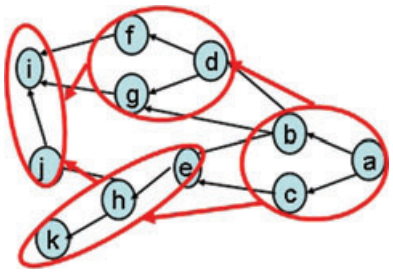


Figure 4: *Partition of a DAG of bricks (blue) into a DAG of cells (red). This is a feasible solution where each cell represents a render node assignment.*

5. Dynamic Programming

The kd-tree solution does not attempt to solve an explicit cost function, rather operates under the assumption that evenly distributing the data is always the best solution. It also restricts the number of render nodes to a power of two. We propose a DP partitioning solution that can readily adapt to different system configurations by using a well-defined cost function and does not have any restriction on the number of render nodes. We start with a high level overview, then describe our solution in terms that are comparable to those used in DPMPP.

5.1. Brick grouping overview

We propose a LBND solution that starts with a set of non-empty bricks that covers all regions of interest in the volume, connected with a DAG that represents the viewing order of these bricks. The general idea is to split the scene into cells, which each contain one or more connected bricks. Figure 4 illustrates a partition of a DAG of bricks into a DAG of cells. This is a feasible solution where each cell represents a render node assignment. In the following discussion, we use uppercase variables for input variables and lowercase ones for calculated variables and indices.

The solution is found incrementally in stages, where a stage corresponds to a brick in the input DAG, and with each additional stage, one additional brick is added. We use b_i to label the brick in stage i. Each feasible partition p for stage i is defined by a DAG of cells containing all bricks in stages from i to $B - 1$, where there are B bricks. The overall optimal solution is the optimal partition for stage 0, which is actually the partition for all stages and represents a render-node assignment.

Cells in partition p are numbered 1 to p_r, where p_r is the number of cells in p. The number of bricks in the k-th cell in partition p is n_k. Each cell corresponds to a composited image to be produced by a render node, where this image is not interleaved with any other render node image. Interleaving occurs when the number of cells in the partition exceeds the number of render-nodes so that there exists at least one render node, a, such that every image rendered on a cannot be combined locally. In other words, for some images, v_1 and v_2, rendered on v, and image w_1 from a different render node, the relative priority order is v_1, w_1, v_2. This results in an additional internode composition and network transfer and may require additional image buffer space for storing v_2 until the v_1, w_1 composited image is ready.

If the network transfer time is high compared with the rendering time, then interleaving images between render nodes results in a slower end-to-end rendering time. Good load balancing is achieved if each render-node is assigned a proportionate amount of the data and not necessarily by minimizing the sum of rendering times. If there are N render nodes, then the total number of partition cells in the optimal solution with a high network transfer cost is X. We use this to prune the feasible solution search space by placing an upper bound, M, on the number of bricks per cell, where M is derived from the total number of non-empty voxels divided by N. This is the equivalent of the resource constraint of MPP and allows us to define our brick grouping solution using the same terms as the DPMPP solution.

5.2. Objective function

We use a cost metric to choose the optimum partition at each stage. For partition p, the cost includes the direct rendering time per brick, R, plus local image composition time, L, plus network transfer time, X, which includes internode image composition. Rendering time is defined in terms of rendering hardware texture bricks. If the input bricks are a multiple of this size, as explained in Section 5.5, then the cost function is scaled accordingly. The correctness of the solution is determined by the accuracy of the cost model.

The cost function presented here reflects the costs of our current cluster implementation. An advantage to using DP is that the global cost formula can be adapted to reflect different implementations such as multithreading. Each brick is projected locally and projection sizes do not vary significantly for orthogonal projection rendering with a given viewing vector; so, a constant rendering time is a good approximation. Although a variable rendering time could be used, a constant simplifies the cost description. The cost of image composition depends on whether any contributing image is sent over the network or not. Local image compositing takes place in the rendering hardware. The most recently rendered image is composited with another image, which must be loaded into memory. Internode image compositing includes network communication time as well.

The input to our problem includes a volumetric data set consisting of a set of brick bounding boxes, a DAG representing the image composition priority order of these bricks and the following set of rendering system parameters:

- B: number of bricks;
- N: number render-nodes;
- M: maximum bricks per cell;
- R: rendering time per brick;
- L: local composition time;
- X: network transfer time.

The network transfer cost for p is $X \times (p_r - 1)$, the rendering cost is $\max(R \times n_k), \forall k \in p$ and the local image composition cost is $\sum_{k=1}^{p_r}(L \times (n_k - 1))$. The total cost of p is

$$c_p = X * (p_r - 1) + \max(R \times n_k) + \sum_{k=1}^{p_r}(L \times (n_k - 1)). \quad (1)$$

5.3. Locally optimal solution

Stages are processed in decreasing order. At each stage, a list of feasibly optimal partitions is determined. Each partition represents adding the brick associated with the current stage, to some partition encountered so far. We determine the cost of every feasibly optimal DAG of cells for each stage, i, and mark the lowest cost of these as the local optimum. A feasible cell to add brick I to, is one that has a neighbour of brick I (corresponding to stage i) and contains no more than M bricks. At each stage, each feasible cell is considered. The algorithm compares a set T of transitions. Transition t consists of $t.postcondition$, $t.operation$, $t.cost$ and $t.solution$. Partition $t.postcondition$ contains bricks for stages $i + 1$ to $B - 1$, and $t.operation$ involves either concatenation of brick b_i to a cell in partition $t.postcondition$ or the creation of a new cell containing only b_i. The cost of each transition, $t.cost$, is computed as described in Section 5.2. Partition $t.solution$ contains bricks for stages i to $B - 1$. The notation here is similar to that used in DPMPP, except that we use $t.solution$ instead of $t.precondition$.

The goal is to find a partition which assigns approximately the same portion of the volume to each render-node, without violating the given precedence order. The optimal transition for stage i is $t^*[i]$, and $c^*[i] = t^*[i].cost$ is the cost of the optimal solution for stage i. We use brackets to distinguish stage optimal transitions from the following interim values, which are calculated for adding brick b_i to partition $t.postcondition$.

- $t_0.solution = t.postcondition$ plus new cell containing only brick b_i;
- $t_k.solution = t.postcondition$ with brick b_i added to cell k.

Only feasible transitions are considered. The optimal groups of cells for different stages can overlap. A globally optimal solution is obtained using DP because the complete set of feasibly optimal solutions is evaluated at each stage, and the optimal path through these solutions is traversed from the initial stage for the final solution. The local optimality condition ensures that the largest cells are selected first. The algorithm processes the brick DAG starting with the last node.

In summary, partitioning stage i, entered with transition set T, where k is a cell, proceeds as follows:

$\forall t.postcondition \in T$
  Create $t_0.solution$
  Calculate cost
  $\forall k \in t.postcondition$ that contains a neighbour of b_i
    if $|k| >= M$ and $t_k.solution$ does not cause interleaving
    with another cell in p
      Create $t_k.solution$
      Calculate cost
$t_i^* =$ lowest cost transition

Stage 0 is the final stage evaluated, and the solution to our DP problem minimizes the objective function, $c^*[0]$, while satisfying the following constraints:

1. $t^*[0].solution$ assigns each brick to exactly one cell;
2. Input brick DAG precedence order is not violated.

5.4. Example stage

At each stage, the set of feasibly optimal partitions is compared. Each connected set of bricks with up to M bricks, which includes the new brick, is considered. Figure 5 illustrates stage d, which compares feasible solutions for the placement of brick b_d in a system where $M = 2$. The value of $M = 2$ is not typical, but is used here for a simple illustration. Stage d is entered with postconditions p_0, p_1 and p_2. Brick b_d can be concatenated to a cell in both p_0 and p_2, but for p_1, the only cell with a link to brick b_d already has the maximum number of bricks; so, a new cell is started. In addition to the solutions shown are partitions with brick b_d added as a separate cell to p_0 and to p_1.

5.5. Algorithm analysis

The principle of optimality [Bel57] states that the optimal solution for the current stage is optimal, regardless of what policies or conditions led to this stage. If every feasibly optimal partition is considered for each stage, then the resulting solution is globally optimal because it observes this at each stage.

Stage i is entered with set p_{i+1}, where p_{i+1} is the set of partitions $t.solution\ \forall t \in t_{i+1}$. The time-complexity for this algorithm is $O(\sum_{i=0}^{B-1}(|P_i|))$ for B stages. For a DAG constructed from a tree with the addition of a sink node that is the child node of all leaf nodes, where A is the average number of children per node in the brick DAG, there are A^m unique cells of an arbitrary size, m, adjacent to i, and each of these is part of a unique partition. Stage i solutions include each of these concatenated with b_i and also includes each of these with an additional cell containing only brick b_i. This represents the worst case, since some of these solutions are not unique for a general DAG. If cells are restricted to contain no more than M bricks, then the number of unique solutions is $|P_i| = \sum_{i=1}^{M-1}(A^m) + \sum_{i=1}^{M}(A^m)$. This is controlled by the maximum number of bricks per cell, M.

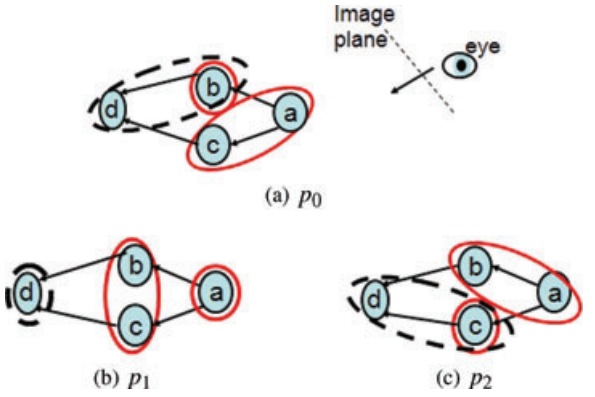


Figure 5: *Stage d with $M = 2$. Evaluate partitions that add brick b_d to postconditions p_0, p_1 and p_2. Not all feasible solutions are shown. (a) Brick b_d concatenated with b_b, (b) new cell is started and (c) Brick b_d concatenated with b_c.*

To prevent the brick grouping algorithm from becoming intractable, we reduce the average number of feasible solutions. We do this by increasing the size of input data bricks to reduce the average number of bricks per cell in each feasible solution. The size of bricks is calculated so that the average number of bricks per render node is less than the number, M, set by the user prior to building the DAG. Brick dimensions are each a multiple of the target GPU texture brick size. Bricks are further divided up into the final textures using a local grid prior to rendering. The DP solution obtained is optimal with respect to the input bricks.

6. Results

To validate our skd-tree partitioning and brick grouping algorithm, we have used them to distribute data for the Visible Korean. We compare these with a grid partition, preprocessed in parallel on the render nodes. We use a 3D grid, which defaults to 2D for all clusters in our experiments except for the 64 node. The granularity of the grid used for distributing data between nodes is set such that the number of grid cells is

equal to the number of render nodes. For the Visible Korean data set, the z-axis is four times as long as the x- and y-axes; so, the kd-tree algorithm is modified slightly to start with several z-cuts.

We compare out-of-core preprocessing to distributed preprocessing. Distributed preprocessing uses a grid for distributing slices prior to reading. To avoid unnecessary internode image transfers, the granularity of the grid is set so that an equal contiguous portion of the data is assigned to each render node. The grid cell size is equal to the whole scene extents, divided by the number of render nodes. Each raw data slice is sent to every render node that overlaps it. Empty cells are marked in the grid as data is read and cropped.

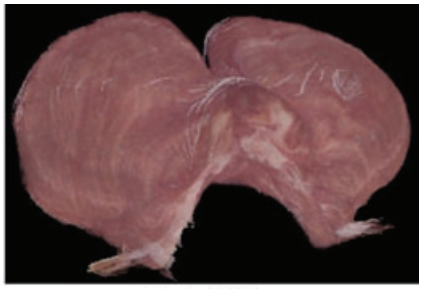
6.1. Test environment

Cost parameters used for the DP partitioning are based on using the ServerNetII and HP Sepia-2a card for image communication and composition and the VolumePro1000 or GeforceFX5800 for rendering. On the MDS cluster, there is a two-tier cost function for network transfer. For each frame, the first image transfer from each node occurs through a DVI output across the SeverNetII, and internode composition time is $X1$. All other transfers are done over the Ethernet and require a framebuffer readback, with internode composition time $X2$. If p_r, the number of cells in partition p, is more than the number of render-nodes, N, in the system, then the internode composition cost for p is $X1 \times (N-1) + X2 \times (p_r - N)$.

The Visible Korean Male data includes 8,590 digitally captured photographic anatomic images of serially sectioned planes with photographic image resolution of $2,468 \times 1,407$ and 24 bits colour, with a slice-interval of 0.2 mm. The data set is accompanied by a corresponding 40 GB set of 8-bit colour *mask* slices with several regions of interest marked with different colours, for a total of 120 GB. Mask slices are used to index a transfer function at run time. We have rendered several segmented regions of data from the Visible Korean Human Project on our cluster. The goal of segmentation here is to segment out empty space for the sake of illustrating the load balancing problem on a scene with an uneven distribution of empty space. It does not preclude changing colouring and translucency using a transfer function during rendering. The cerebellum and brain stem are shown in Figure 6. They have been rendered on a GPU using a gradient mask volume and pre-segmented texture slices. The Visible Korean Male bones, cerebrum and lungs, rendered on VolumePro1000 hardware, are shown in Figure 7.

6.2. Timing

The partitioning times for traditional kd-tree, skd-tree partitioning and brick grouping are compared in Table 1. We also



(a) Cerebellum

(b) Brain Stem

Figure 6: *GPU-rendered images of segmented regions from the Visible Korean data set: (a) cerebellum and (b) brain stem.*

ran experiments where cropped data was distributed with an octree subdivision; however, the maximum data rendered are generally not reduced by using this strategy. The skd-tree is a better choice because it allows the partition planes to adapt to the data and avoids additional overhead in managing multiple subvolumes assigned to a render node. The skd-tree uses an order of magnitude less preprocess time on average that the traditional one that requires multiple slice reads. The brick grouping preprocessing time does not grow significantly as long as the size of the maximum number of bricks per render node assignment is constant. Preprocessing for the grid partitioning is completed in the slice preprocessing phase, so, is not included in this table. Grid preprocessing is significantly simpler and faster. However, for interactive rendering, the improvement in rendering speed achieved by our partitions is more important.

A comparison of parallel and single node slice preprocessing, on an eight-node cluster, is shown in Table 2. The time required to segment, crop and distribute each data slice is the same for the skd-tree and brick grouping DP

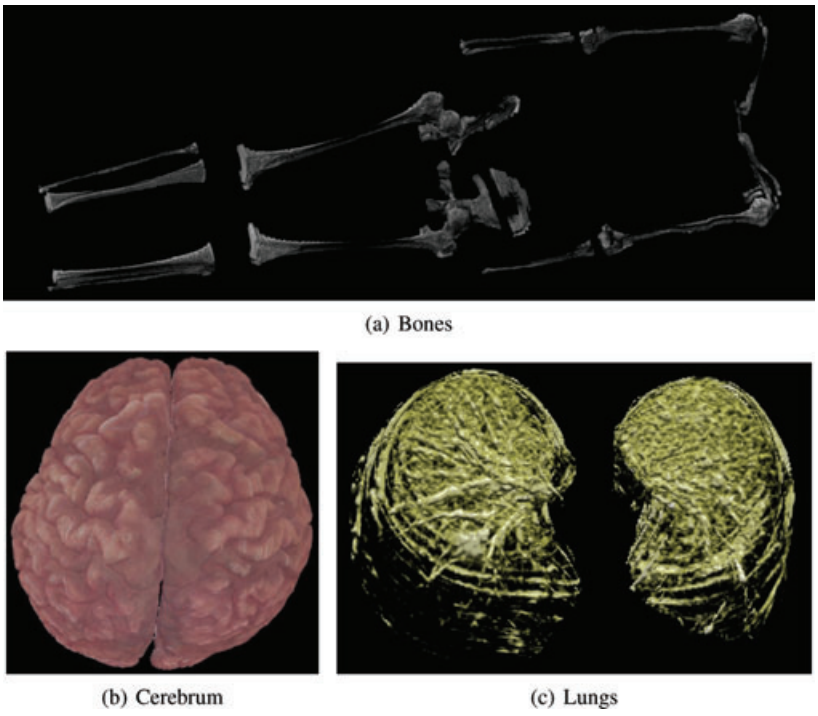


Figure 7: *VolumePro1000-rendered images from Visible Korean data set: (a) bones, (b) cerebrum and (c) lungs.*

Table 1: *Partition preprocessing time (sec) for kd-tree, skd-tree, and brick grouping DP (including bricking) for the Visible Korean bones on clusters with 8, 16, 32 and 64 nodes.*

Cluster nodes	8	16	32	64
Kd-tree	792	1697	3508	7129
Skd-tree	213	215	218	221
Brick grouping	251	252	253	255

algorithms. The time for data distribution is higher for the grid partition because it is done prior to cropping. However, the slice preprocessing is done in parallel, and the additional overhead for marking grid cells during this process is negligible. Rendering times for all except the bones data are interactive (32 frames per second (fps)). The speedup is low for the cerebellum, lungs, brain stem and cerebrum because most of the empty space is cropped prior to data distribution. However, rendering of the cropped bone data is unsuccessful without using either skd-tree or brick grouping DP. This is because the cropped bone data do not fit when the grid partitioning is used, and a large amount of data is loaded into the VolumePro1000 memory during rendering, causing the system to hang up. Using the skd-tree and brick grouping DP algorithms both reduced the data set to fit into the total memory of the VolumePro1000 cards on our cluster. The average frame rate for the bones data rendered on eight nodes was 1.35 fps with skd-tree partitioning and 2.6 fps using DP partitioning.

Table 2: *Empty space (percent) and slice preprocessing time (sec) on eight nodes for the Visible Korean data set.*

	Cerebellum	Brain Stem	Bones	Cerebrum	Lungs
Data size (GB)	3.61	2.1	117.43	8.7	22.61
Empty space (%)	85	90	81	89	82
Slice preprocessing	4.7	0.45	109.2	8.01	21.03
Parallel slice preprocessing	0.8	0.34	18.4	1.36	3.54

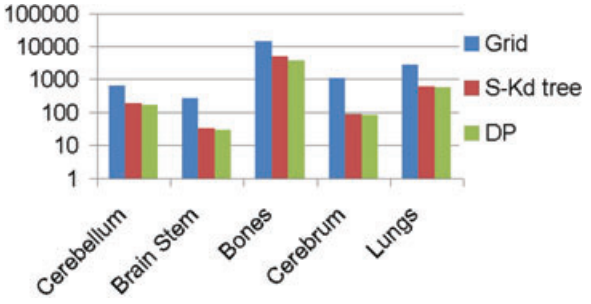


Figure 8: *Data reduction with cropping. Maximum data assigned to any node for an eight-node cluster. Logarithmic scale used because of large range in results.*

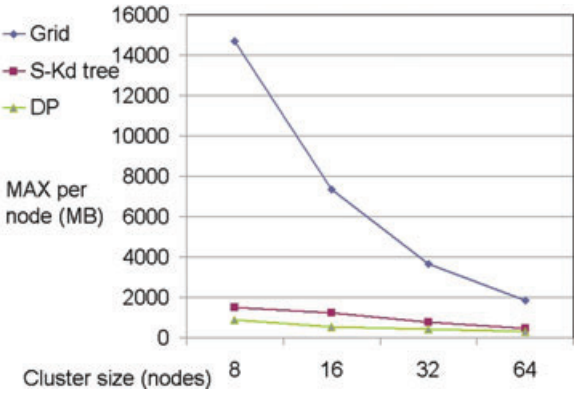


Figure 9: *Load balance scalability. Maximum data assigned to any render-node in 8, 16, 32 and 64 node clusters.*

6.3. Load balancing results

Data cropping is only useful in reducing end-to-end rendering time if it results in reducing the maximum data rendered on any node. Figure 8 shows the maximum data assigned per node using our algorithms compared with the grid partition. The largest decrease in data rendered compared with a grid partition is due to cropping because the maximum data rendered with the grid is the same as the full portion of data assigned prior to cropping, with some nodes rendering no data. This is due to the high grid granularity required on the intra-node distribution level. Using a fine granularity local grid reduces rendering time on some nodes. However, it doesn't improve the end-to-end render time because it is bound by the slowest node, which contains no empty cells even within the finer granularity. All of the data sets show large reductions for both the skd-tree partitioning and brick grouping algorithm and DP partitions compared with the grid partition. The smaller maximum data per node for DP partition is due to the use of a cost function in incrementally finding the partition. Load balancing improvements over the skd-tree partitioning and brick grouping algorithm partition is most significant for data sets that have large variations in empty space. This demonstrates that a cost-driven DP approach can reduce rendering time.

Partitioning using DP was similar, and in some cases identical, to the skd-tree partitioning and brick grouping algorithm partition for the cerebellum, lungs, brain stem and cerebrum. This is because most of the empty space is cropped prior to data distribution. The speedup of DP compared with skd-tree for the bones is probably due to the fact that we restricted the depth of the skd-tree to force the number of leaf nodes to be equal to the number of render nodes to avoid interleaving of images between render nodes. The speed up due to empty space comes from the fact that data is distributed in a manner that only requires a single image to be sent from any render node. This is the result of the cost function that assigns a high penalty for data transfers. For an octree to meet this criterion, the data must be statically distributed prior to cropping, resulting in an uneven distribution of empty space. In a system where the transfer function cost is not dominant, multiple image compositions/transfers per node may be part of the DP solution. The load balancing was most improved for the bones because there is a wider variation in empty space distribution for the segmented bones in the original data set. For these cases, it is clearly advantageous to preprocess the data using one of the algorithms presented. When there is a large variation of empty space in the data set, static parallel partitioning results in poor load balancing. The grid assigns zero data to some nodes for several of the data sets because of the high grid granularity required on the intra-node distribution level.

The scalability of load balancing is illustrated in Figure 9 for the bones on different sized clusters. Although the grid partition improves with a larger cluster because of the finer grid granularity, the number of render nodes that remain idle also increases. The amount of data assigned per node does not reach our more adaptive techniques, and it is expected that this gap will persist for any system size.

7. Conclusion and Future Work

Managing massive data sets is a fundamental requirement for distributed volume rendering. We have introduced practical methods for reducing data early in the preprocessing pipeline and for out-of-core data distribution. Scalable priority-constrained data distribution is an open area of research. We have introduced a solution using DP, which allows a cost function to drive the distribution process, and a kd-tree solution. Our skd-tree partitioning and brick grouping algorithms each finds a good partition with a moderate preprocessing time using a series of slab-projection slices. As demonstrated by our test results, our solution allows the scene partition to be controlled so that data is distributed more evenly between render-nodes. The portion of the scene assigned to each render-node is dictated by the distribution

of non-empty regions, resulting in better load balancing than with traditional partitioning methods.

In our skd-tree partitioning and brick grouping algorithm and DP partitions, natural boundaries of non-empty regions are cropped prior to data distribution, resulting in a smaller maximum data assigned per node compared with a grid partition with the same number of cells. In our experiments, scenes with large portions of non-uniformly distributed empty space show more dramatic improvements using our approach than those with more homogeneous data distributions.

Skd-tree partitioning has some advantages over the DP model. For instance, it is more suitable for perspective projection. The main drawback of the DP is that a solution for each octant is required, and some replication of bricks is necessary. The other drawback is that it is intractable if the number of decision branches at any stage is not controlled by the average number of neighbouring bricks per brick. However, this algorithm demonstrates the potential to reduce rendering time using a cost-driven DP approach.

A promising area of future work is dynamic load balancing. Interactive transfer function updates are possible with our method. Recent developments in dynamic scanner technologies produce time varying data scans, which are very well suited for these render clusters due to their tremendous size. Further research is needed for adapting the partition found during our s-kd or DP preprocessing. One solution would use an incremental update approach to move partitions as regions of non-empty voxels move throughout the scene.

Acknowledgements

This work has been partially supported by NSF grant CCF-0702699. The Visible Korean Project data set is courtesy of Humintec, Korea. We also thank HP for providing us with the Sepia-2a hardware and TeraRecon for supplying us with VolumePro1000 hardware.

References

[Bel57] Bellman R. E.: *Dynamic Programming*. Dover, 1957.

[Cha02] Chalmers A.: *Practical Parallel Rendering*. A K Peters, 2002.

[CNS*02] Chan E., Ng R., Sen P., Proudfoot K., Hanrahan P.: Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware* (Sept. 2002), pp. 69–78.

[EKE01] Engel K., Kraus M., Ertl T.: High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware* (Sept. 2001), 9–16.

[FK05] Frank S., Kaufman A.: Distributed volume rendering on a visualization cluster. *CAD/Graphics*, 2005, pp. 371–376.

[FRK01] Fajardo R., Ryan T., Kappelman J.: Assessing the accuracy of high-resolution X-ray computed tomography of primate trabecular bone by comparisons with histological sections. *American Journal of Physical Anthropology 118*, 1 (2001), 1–10.

[GJ79] Garey M., Johnson D.: *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, January 1979.

[GPKM03] Ghosh A., Prabhu P., Kaufman A., Mueller K.: Hardware assisted multichannel volume rendering. In *Proceedings of Computer Graphics International*, *IEEE Computer Society* (July 2003), pp. 2–7.

[Hei05] Heirich A.: Optimal automatic multi-pass shader partitioning by dynamic programming. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2005), pp. 91–98.

[Kno06] Knoll A.: A survey of octree volume rendering methods. In *Proceedings of the 1st IRTG Workshop*. Dagstuhl, Germany, June 2006.

[KW03] Krüger J., Westermann R.: Acceleration Techniques for GPU-Based Volume Rendering. In *Proceedings of the IEEE Visualization* (2003), pp. 287–292.

[KWPH06] Knoll A., Wald I., Parker S., Hansen C.: Interactive isosurface ray tracing of large octree volumes. *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pp. 115–124.

[LMK03] Li W., Mueller K., Kaufman A.: Empty space skipping and occlusion clipping for texture-based volume rendering. In *Proceedings of the IEEE Visualization* (Oct. 2003), pp. 317–324.

[LMS*01] Lombeyda S., Moll L., Shand M., Breen D., Heirich A.: Scalable interactive volume rendering using off-the-shelf components. In *Proceedings of IEEE Symposium on Parallel Visualization and Graphics* (Oct. 2001), pp. 115–121.

[MHS99] Moll L., Heirich A., Shand M.: Sepia: scalable 3D compositing using PCI pamette. In *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines* (April 1999), pp. 146–155.

[MLM*03] Muraki S., Lum E., Ma K., Ogata M., Liu X.: A PC cluster system for simultateous interactive volume

modeling and visualization. In *Proceedings of IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (October 2003), pp. 95–102.

[MMD06] Marchesin S., Mongenet C., Dischler J.: Dynamic load balancing for parallel volume rendering. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization* (2006), 43–50.

[MSE06] Müller C., Strengert M., Ertl T.: Optimized volume raycasting for graphics-hardware-based cluster systems, *Eurographics Symposium on Parallel Graphics and Visualization*, pp. 59–66.

[MWMS07] Moloney B., Weiskopf D., Mller T., Strengert M.: Scalable sort-first parallel direct volume rendering with dynamic load balancing. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization* (May 2007), 45–52.

[PCH*06] Park J., Chung J., Hwang S., Shin B., Park H.: Visible Korean Human: Its techniques and applications. *Clinical Anatomy 19* (2006), 216–224.

[PSL*98] Parker S., Shirley P., Livnat Y., Hansen C., Sloan P.-P.: Interactive ray tracing for isosurface rendering. In *Proceedings of IEEE Visualization* (1998), pp. 233–238.

[RLV*04] Riffel A., Lefohn A. E., Vidimce K., Leone M., Owens J. D.: Mio: fast multipass partitioning via priority-based instruction scheduling. In *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2004), pp. 35–44.

[WIK*06] Wald I., Ize T., Kensler A., Knoll A., Parker S.: Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics 25*, 3 (2006), 485–493.