Parallel Cell Projection Rendering of Adaptive Mesh Refinement Data

Gunther H. Weber[1,2,3] Martin Öhler[2] Oliver Kreylos[1,3] John M. Shalf[3] E. Wes Bethel[3]
Bernd Hamann[1,3] Gerik Scheuermann[2]

[1] Center for Image Processing and Integrated Computing (CIPIC), Department of Computer Science,
One Shields Avenue, University of California, Davis, CA 95616-8562, U.S.A.
[2] AG Graphische Datenverabeitung und Computergeometrie, FB Informatik, University of Kaiserslautern,
Erwin-Schrödinger Straße, D-67653 Kaiserslautern, Germany
[3] Visualization Group, National Energy Research Scientific Computing Center (NERSC),
Lawrence Berkeley National Laboratory, One Cyclotron Road, Berkeley, CA 94720, U.S.A.

Abstract

Adaptive Mesh Refinement (AMR) is a technique used in numerical simulations to automatically refine (or de-refine) certain regions of the physical domain in a finite difference calculation. AMR data consists of nested hierarchies of data grids. As AMR visualization is still a relatively unexplored topic, our work is motivated by the need to perform efficient visualization of large AMR data sets. We present a software algorithm for parallel direct volume rendering of AMR data using a cell-projection technique on several different parallel platforms. Our algorithm can use one of several different distribution methods, and we present performance results for each of these alternative approaches. By partitioning an AMR data set into blocks of constant resolution and estimating rendering costs of individual blocks using an application specific benchmark, it is possible to achieve even load balancing.

CR Categories: D.1.3 [Concurrent Programming]: Parallel Programming; I.3.2 [Graphics Systems]: Distributed/network graphics; I.3.3 [Picture/Image Generation]: Display algorithms

Keywords: volume rendering, adaptive mesh refinement, load balancing, multi-grid methods, parallel rendering, visualization

1 Introduction

Physical phenomena can vary widely in scale. Large regions in space can exist where a physical variable varies only slightly, and thus may be adequately represented at low resolution. Other regions may require higher resolutions to capture rapid changes. In 1984, AMR was introduced to computational physics by Berger and Oliger [1984]. AMR represents a spatial domain as nested structured grids of increasing resolution, and provides the ability to locally increase resolution only where it is needed. [Berger and Oliger 1984] used a scheme where refining grids can be rotated with respect to a parent level. A modified version [Berger and Colella 1989] of their algorithm was published later, where all refining grids are axis-aligned with respect to the parent level. AMR has become increasingly popular, also outside the computational physics community. Today, it is used in a large variety of applications. For example, [Bryan 1999] used the technique to simulate astrophysical phenomena using a hybrid approach combining AMR grids and particles.

Based on an efficient software cell-projection volume renderer, we have developed a framework for parallel volume rendering of AMR data. Even though cell-projection [Ma and Crockett 1997] was introduced primarily for rendering unstructured meshes, it also leads to efficient implementations for structured meshes. Our method partitions an AMR hierarchy using a k-d tree [Bentley 1975]. This partition is view-independent and computed offline in a preprocessing step. We have developed and compared several partition strategies that we briefly summarize.

Uniform root-level subdivision ignores the hierarchical nature of AMR data and partitions a root level into blocks of constant size. Refined cells are handled during rendering by recursive descending into finer levels.

Weighted root-level subdivision partitions a root level into blocks at approximately constant computational cost. The AMR hierarchy is only considered to compute weights. Locations for subdivision are chosen independently from boundaries of refining grids. During the rendering process refining grids are handled by descending recursively.

Homogeneous subdivision subdivides AMR levels recursively until each part only covers one grid of a given level, i.e., until it corresponds to a region represented at constant resolution. The resulting grid parts are distributed evenly among processors.

Weighted homogeneous subdivision partitions AMR levels in the same way as homogeneous subdivision. The computational cost for rendering a constant-resolution region is estimated and associated with that region as its *weight*. Grid parts are distributed among processors such that the sum of associated weights is approximately the same for all processors.

Our framework supports rapid development and testing of new distribution strategies and volume rendering techniques.

2 Related Work

Initial work in AMR visualization focused on converting AMR data to suitable conventional representations and visualizing them. [Norman et al. 1999] described a method that visualizes AMR data using standard toolkits. Their method converts an AMR hierarchy

into an unstructured grid composed of hexahedral cells. The resulting unstructured grid is then used for visualization with standard algorithms. When converting AMR data to an unstructured mesh, its main advantage, the implicit definition of grid connectivity, is lost. Thus, [Norman et al. 1999] extended VTK to handle AMR as first-class data structure. [Max 1993] described sorting schemes for cells for volume rendering and described their application to AMR data. [Ma 1999] described parallel rendering of structured AMR data resulting from simulations using the PARAMESH framework [MacNeice et al. 2000]. He described two approaches for volume rendering of AMR data. One method resamples a hierarchy on an uniform grid at the finest resolution. The resulting grid is evenly subdivided and each part rendered on an individual processor. A second method preserves the AMR structure.

[Weber et al. 2001a] presented two volume rendering schemes for Berger-Colella AMR data. One scheme is a hardware-accelerated renderer for previewing; the other scheme supports progressive refinement rendering of AMR data based on cell projection [Ma and Crockett 1997]. [Weber et al. 2001b] described a method to extract isosurfaces from AMR data. To avoid resampling, their method interprets locations of cell-centered data values as vertices of a dual grid. Resulting gaps between hierarchy levels are filled via a generic stitching scheme. [Weber et al. 2001c] discussed using dual-grids and stitch-cells to define a consistent interpolation scheme for high-quality volume rendering of Berger-Colella AMR data. [Kreylos et al. 2002] described a framework that partitions a Berger-Colella AMR hierarchy partitioned in blocks of constant resolution using a k-d tree. Resulting blocks are distributed among processors and rendered using either a texture-based hardware-accelerated approach or a software-based cell-projection renderer. [Ligocki et al. 2003] described *ChomboVis*[1], a framework for the visualization of hierarchical computations using AMR.

[Kähler and Hege 2002] introduced a scheme to partition Berger-Collela AMR data in blocks of constant resolution. Aiming to minimize the number of generated constant-resolution blocks, their approach utilizes a heuristic based on assumptions concerning the placement of refining grids by an AMR simulation. [Kähler et al. 2003] developed a method that uses AMR hierarchies for rendering sparse volumetric data. Given a transfer function, this method computes a transfer-function-specific AMR hierarchy for a volume data set and renders it using the algorithm of [Kähler and Hege 2002]. [Kähler et al. 2002] used existing tools to render simulation results of a forming star. By specifying a transfer function and a range of isovalues [Park et al. 2002] produced volume-rendered images of AMR data based on hierarchical splatting, see also [Laur and Hanrahan 1991]. Their method converts an AMR hierarchy to a k-d-tree structure consisting of blocks of constant resolution, which are rendered back-to-front using hierarchical splatting.

3 AMR Data Format

Figure 1 shows a simple 2D AMR hierarchy produced by the Berger–Colella method. The basic building block of a d–dimensional Berger-Colella AMR hierarchy is an axis-aligned, structured rectilinear grid. Considering the 3D case, each grid g consists of hexahedral cells and is positioned by specifying its local origin. AMR typically uses a *cell-centered* data format, i.e., dependent function values are associated with cells/cell centers. Data values are stored in arrays as location and connectivity can be inferred from the regular grid structure.

An AMR hierarchy consists of several levels Λ_l comprising one or multiple grids. All grids in the same level have the same cell size. A hierarchy's *root level* Λ_0 is the coarsest level. Each level Λ_l may be refined by a finer level Λ_{l+1}. A grid of a refined level is referred to as a *coarse grid* and a grid of a refining level as a *fine grid*. A *refinement ratio* r specifies how many fine grid cells fit into a coarse grid cell, considering all axis-directions. This value is always a positive integer. A refining grid refines an entire level Λ_l, i.e., it is completely contained in the region covered by that level but not necessarily in the region covered by a single grid of that level. Each refining grid can only refine complete grid cells of the parent level, i.e., it must start and end at the boundaries of grid cells of the parent level. The Berger-Colella scheme [Berger and Colella 1989] requires the existence of a layer with a width of at least one grid cell between a refining grid and the boundary of the refined level.

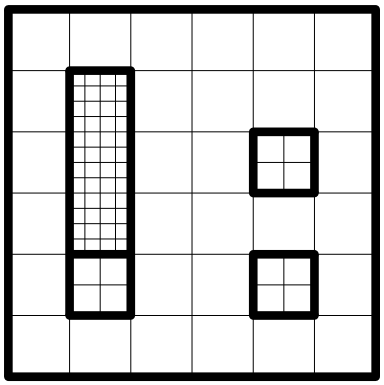


Figure 1: AMR hierarchy consisting of five grids and three levels. Level boundaries are shown as bold lines.

4 Design Considerations

One can differentiate volume rendering methods by their underlying illumination models (i.e., the "optical properties" of transfer functions) and by their operation in *image* or *object space*. Two illumination models are widely used in volume rendering: The absorption and emission light model, described, for example, by [Max 1995] and the Phong-based light model by [Levoy 1988]. We chose the absorption and emission light model as it leads to efficient implementations. Within cells, we use constant interpolation, i.e., the sample value located at the cell center is assigned to all positions within the cell. This allows an exact evaluation of the light-model and preserves the AMR hierarchy in rendered images. To achieve more efficiency, we chose orthographic projection over perspective projection.

Image-space-based algorithms, including the commonly used ray casting algorithm, see [Sabella 1988], operate on pixels in screen space as "computational units," i.e., they perform computations on a per-pixel basis. Object-space-based methods, like cell projection, see Ma and Crockett [1997], operate on 3D grid cells. Parallelizing volume rendering can be done in image space or in object space, see [Crockett 1997]. Image-space parallelization subdivides the image plane to distribute computing among multiple processors. Each processor renders a subset of pixels in an image. Object-space parallelization subdivides the domain of a data set and assigns grid cells to processors. We chose object-space based parallelization, as the hierarchical nature of AMR data facilitates efficient subdivision of the grids. We chose cell-projection as an object-space-based rendering methods, as it leads to an elegant implementation of domain subdivision. Furthermore, using cell-projection simplifies reaction to changes in resolution, i.e., it is possible to render finer grids at a higher resolution.

For implementation of the parallel renderer we chose the *Message Passing Interface* (MPI) library over the *Parallel Virtual Ma-*

[1] Joint effort of the Applied Numerical Algorithms Group and the Visualization Group at LBNL. See `http://seesar.lbl.gov/anag/chombo/chombovis.html`.

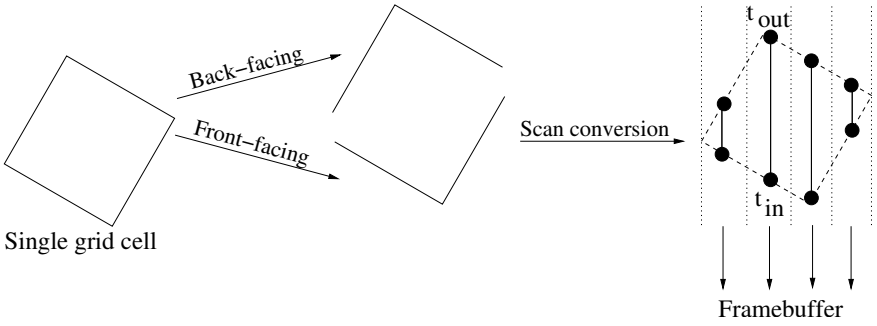


Figure 2: Cell-projection process.

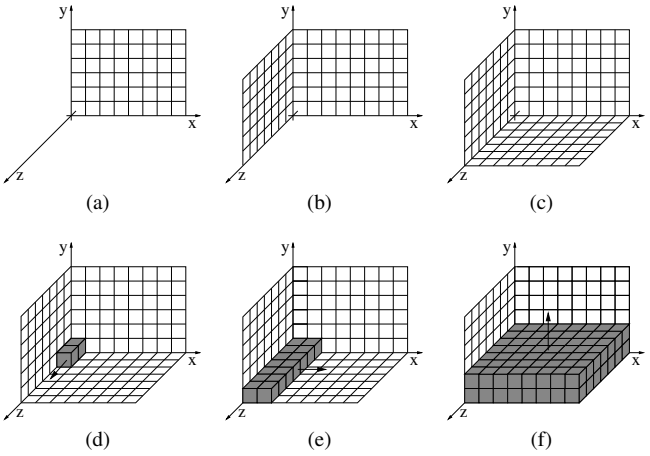


Figure 3: Rendering order of grid cells — all components of $\vec{tv}$ (the vector pointing toward the viewer, see Figure 4(a)) being positive. First, all back-facing faces of the first layer of cells in each direction are rendered (a) – (c). Second, all cells are rendered. The order in which axes are handled (first-z−-then-x-then-y order) is arbitrary. Only the order according to which cells are handled along an axis is important.

chine (PVM) framework. MPI is commonly used in AMR simulations, thus making our framework more compatible with other applications, including numerical simulation. Furthermore, MPI is the de facto standard for parallel supercomputers. Vendor-specific adaptations for different architectures exist, supporting the utilization of specific hardware optimizations by linking to a vendor-provided library. Instead of adopting the classic master-slave model, we chose a symmetric implementation to avoid communication bottlenecks. Each processor computes the complete distribution of grid parts and selects a subset based on its index. However, we are using a binary-tree image compositing scheme that pairs processors in each compositing step. In each step, one processor of each pair receives an intermediate partial image from its "neighbor" and performs a compositing operation. The final composited image resides in the buffer of processor zero.

5 Software-based Cell-Projection Done Efficiently

5.1 Overview

Cell projection [Ma and Crockett 1997] is an object-space-based volume rendering method, similar in nature to ray casting. Both methods trace rays through a volume, accumulating light along the path of a ray. Ray casting operates on a per-pixel basis, using one ray for each pixel. Cell-projection-based methods construct "ray segments" for cells and merge them with existing ray segments.

Usually, a priority queue is maintained for each pixel collecting all ray segments contributing to that pixel. Figure 2 shows the fundamental idea of cell projection. Boundary faces of all cells are divided into three groups, *front-facing* faces (with normals directed toward the viewer), *back-facing* faces (with normals directed away from the viewer), and *view-perpendicular* (with normals perpendicular to the viewing direction). First, the back-facing faces are scan-converted into a buffer. For each pixel influenced by the cell, this buffer holds a depth corresponding to an exit parameter value, called t_{out}, along the ray. Second, the front-facing faces are scan-converted. For each generated pixel, the depth corresponding to the entry parameter value, called t_{in}, along the ray is computed. The entry parameter value t_{in} and corresponding scalar value are read from the buffer, and the ray segment reaching from t_{in} to t_{out} is constructed. Usually, this ray segment is then inserted into the ray-segment queue of the corresponding pixel and merged with adjacent ray segments in that queue.

When cells are sorted using the scheme of [Max 1993], for example, and rendered in back-to-front or front-to-back order, the queue for collecting ray segments is not necessary. Newly generated ray segments are always adjacent to already computed ray segments and can be composited directly in the frame buffer. Another advantage of this method is that it allows us to avoid duplicate scan conversion of a cell's boundary faces. When rendering unsorted cells, back-facing and front-facing faces must be rendered to determine correct ray-segment length. In contrast, when rendering presorted cells, it is sufficient to render the front-facing faces of a cell. All back-facing faces are already rendered as front-facing faces of cells "behind" the current cell.

5.2 Cell Sorting and Front-face Determination

Determining the correct back-to-front cell order in AMR grids is straightforward. For orthographic projections, rendering order can be determined based on view direction. Cells are enumerated by three nested loops, one loop for each axis. Axes can be processed in an arbitrary order. Along each axis, cells must be rendered in correct order. For each loop, this order can be determined based on the sign of the corresponding component of the vector $\vec{tv}$ (a vector directed toward the viewer), according to the axis handled by the loop. If it is positive, cells are enumerated in ascending axis direction. If it is negative, cells are enumerated in descending axis direction. If it is zero, an arbitrary choice is made.

Before rendering cells and generating ray segments, all cell faces lying on back-facing boundary-faces of the overall AMR grid that are not view-perpendicular must be scan-converted. These are the back-facing faces of cells that do not lie in front of any grid cell.

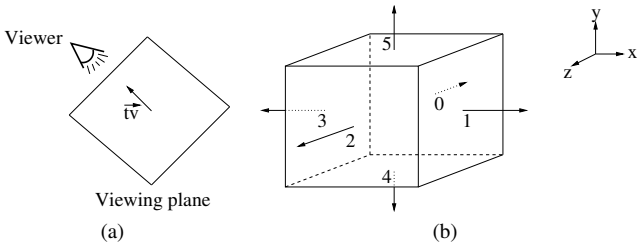


Figure 4: Determining front- and back-facing faces. (a) The vector $\vec{tv}$ is perpendicular to the viewing plane, pointing toward the viewer. (b) Face numbering used in Table 1.

Face #	Front-facing	Back-facing	Perpendicular
0	$tv_z < 0$	$tv_z > 0$	$tv_z = 0$
1	$tv_x > 0$	$tv_x < 0$	$tv_x = 0$
2	$tv_z > 0$	$tv_z < 0$	$tv_z = 0$
3	$tv_x < 0$	$tv_x > 0$	$tv_x = 0$
4	$tv_y < 0$	$tv_y > 0$	$tv_y = 0$
5	$tv_y > 0$	$tv_y < 0$	$tv_y = 0$

Table 1: Criteria used for checking whether a cell face is front-facing, back-facing, or should not be rendered.

Figures 3 (a)–(c) illustrate the procedure for a choice of $\vec{tv}$ where all components are positive. If one component of $\vec{tv}$ is zero, the corresponding face is perpendicular to the viewing direction and discarded. Subsequently, the front facing faces of all cells are scan-converted. Ray segments are generated and composited in the frame buffer. Figure 3 shows the order of scan-conversion used for boundary faces and cells when all components of $\vec{tv}$ are positive.

When dealing with more general grid cells, each boundary face must be checked individually whether it is back-facing or front-facing. This step can be done, for example, by using the scalar product between face normal and $\vec{tv}$. For axis-aligned rectilinear grids, this test can be performed based on viewing direction. Figure 4 shows the numbering of the faces of a cell of a rectilinear grid. Table 1 lists the criteria used to determine whether a face is front- or back-facing. Front- and back-facing faces are the same for each cell in all grids. It is sufficient to determine front-facing faces once.

5.3 Boundary Face Scan-conversion

We render cell boundary faces using a modified version of the polygon scan-conversion algorithm developed by [Gordon et al. 1994] that is based on a method developed by [Kaufman 1988]. Before rendering a polygon, its vertices are projected onto the viewing plane, and point coordinates are rounded to integers. During the scan-conversion process it is assumed that coordinates are specified counter-clockwise. Gordon et al.'s method starts by determining "critical points" of a polygon, i.e., vertices that constitute a local minimum or are first of a set of vertices that together form a local minimum in y-direction. Boundary faces of rectilinear cells are convex quadrilaterals and have only one minimum. If two adjacent vertices share the same value for the y-coordinate, i.e., if they are connected by a horizontal line segment, we consider the vertex with the lower index to be the critical point. During the determination of critical points we also detect polygons that span one pixel in y-direction and discard them.

The algorithm starts by inserting the left minimum and right maximum edge originating from the critical point vertex into an active-edge table (AET). During the scan-conversion process this data structure holds the left and right edge intersecting the current scan-line. For general polygons considered by [Gordon et al. 1994], this structure is an array, as the scan-line can intersect several polygons. Our scan-converter is optimized for convex quadrilaterals and only stores two pointers to AET elements since a convex quadrilateral intersects a scan-line only twice, except for horizontal boundary lines coinciding with a scan-line. For each scan-line, x-coordinates and depth on the left and the right side of the polygon are calculated by linear interpolation. Depth information is not rounded, as exact values are needed for the determination of ray-segment lengths. If a scan-line consists of only one pixel of the polygon it is discarded; otherwise, depth values are computed for all pixels between the x-coordinates by linear interpolation. Ray segments are created by reading the previous depth value and applying the illumination model. If a scan line coincides with the end of an edge, the corresponding pointer referring to the AET is replaced with its successor until the next edge turns down.

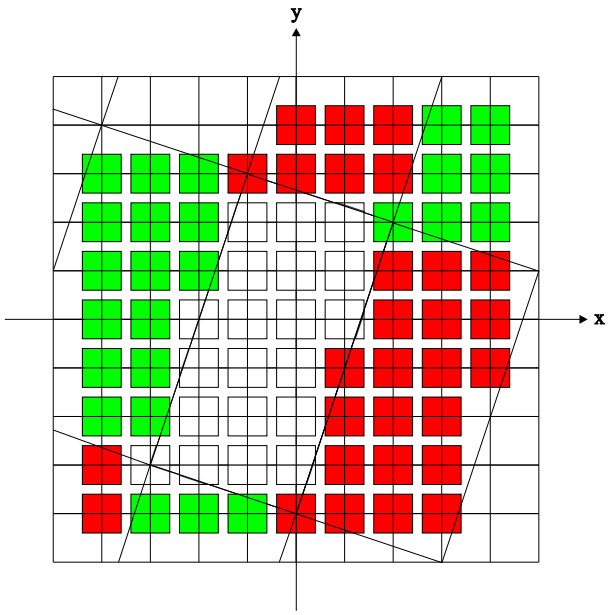


Figure 5: Scan-converted polygon illustrating rules used to determine whether a pixel belongs to the polygon.

When generating images with a cell-projection method it is important that rasterized polygons sharing an edge do not overlap. We use the following set of rules to prevent polygon edge overlap during scan conversion:

R1. Integer intersection points of a polygon edge with a scan-line belong to a polygon, if they lie on its left edge. If they lie on its right edge, they do not belong to the polygon.

R2. Non-integer intersection points of a polygon edge with a scan-line are rounded down. The corresponding pixel belongs to a polygon if it lies on its right edge. If it lies on its left edge, it does not belong to the polygon.

R3. If a pixel corresponds to intersection points on the left and right edges of a polygon it lies outside the polygon.

The white center polygon in Figure 5 illustrates these rules: The pixel at the lower-left corner of the polygon has an integer intersection point and lies on its left edge. Consequently, according to R1, it belongs to the polygon. According to R1, the pixel at the upper-right corner of the polygon does not belong to the polygon. Considering R2, all pixels with non-integer intersection points on the left and lower polygon edge do not belong to the polygon. (They belong to the neighboring polygon.) All pixels bordered by the upper

and right polygon edges do belong to the polygon. The pixel at the upper-left corner lies on the left and the right edges of the polygon and does not belong to the polygon (R3). During scan-conversion, we maintain a list of all positions modified, i.e., covered by a cell. This list is used in the compositing scheme, see Section 7, and to speed up clearing the frame buffer by only erasing pixels modified during rendering.

5.4 Ray-segment Generation

We use constant interpolation in individual cells, i.e., the sample value associated with a cell is assigned to all points inside the cell. Consequently, all points in a cell have the same optical properties, i.e., emission color and opacity. It is possible to solve the differential equations for light absorption and emission analytically in a cell and obtain "correct" opacity and emission values for a ray segment intersecting the cell. Each ray segment in a cell is characterized by an entry parameter value t_{in}, i.e., the distance from the viewing plane at which a ray enters the cell measured along the ray, and an exit parameter value t_{out}, i.e., the distance from the viewing plane at which the ray "exits" the cell. The value of t_{in} is obtained by scan-converting the front-facing faces of a cell. The value of t_{out} is read from the frame buffer containing the results from scan-converting the front faces of cells (behind the current cell) that coincide with the back-facing faces of the current cell. Emission color and opacity are defined by the cell's associated scalar value via a transfer function.

[Max 1995] described the absorption and emission light model using an extinction coefficient τ_{Cell} instead of opacity α_{Cell} per unit length. This extinction coefficient can be obtained from the opacity as

$$\tau_{\text{Cell}} = -\ln(1-\alpha_{\text{Cell}}) \,. \tag{1}$$

Using the formulation from [Max 1995] we obtain the transparency from t_{in} to an arbitrary parameter value along the ray in the cell as

$$T_{\text{Cell}}(s) = \exp\left(-\int_{t_{\text{in}}}^{s} \tau_{\text{Cell}} dk\right) = (1-\alpha_{\text{Cell}})^{s-t_{\text{in}}} \,. \tag{2}$$

The opacity of a ray segment is

$$\alpha_{\text{Seg}} = 1 - T_{\text{Cell}}(t_{\text{out}}) = 1 - (1-\alpha_{\text{Cell}})^{t_{\text{out}}-t_{\text{in}}} \,. \tag{3}$$

Furthermore, the color (intensity) of a ray segment can be computed as

$$C_{\text{Seg}} = \int_{t_{\text{in}}}^{t_{\text{out}}} C_{\text{Cell}} \tau_{\text{Cell}} T(s) ds = C_{\text{Cell}} \alpha_{\text{Seg}} \,, \tag{4}$$

using again equations from [Max 1995]. Combining contributions of individual ray segments is equivalent to compositing pixels with a color C_{Seg} and an opacity α_{Seg} using pre-multiplied alpha values, see Porter and Duff [1984], i.e.,

$$\begin{aligned} \alpha_{\text{Combined}} &= 1-(1-\alpha_{\text{Seg,front}})(1-\alpha_{\text{Seg,back}}) \quad \text{and} \\ C_{\text{Combined}} &= \alpha_{\text{Combined}}\Big(\big(1-\alpha_{\text{Seg,front}}\big) C_{\text{Seg,back}} + C_{\text{Seg,front}}\Big) \,. \end{aligned} \tag{5}$$

6 Partitioning and Load Balancing

6.1 Overview

Our method stores a domain partition as a k-d tree [Bentley 1975]. A k-d tree is a generalization of a binary search tree to an arbitrary number of dimensions. Each level partitions a domain in two regions along an axis-perpendicular plane. The "left" subtree corresponds to points in space whose coordinates in partition direction have values smaller than or equal to the partition position. The "right" subtree corresponds to points whose coordinates in partition direction have values larger than the partition position. The subdivision direction is usually cycled among the three coordinate axes in the 3D case. We skip subdivision directions, when no "sensible" subdivision position along that direction exists. When using an object-space-based subdivision for parallelizing volume rendering, regions must be rendered in correct order. We use the k-d tree to quickly compute the correct render order for each of the partitioned regions of space. At each node of the k-d tree, the domain is subdivided along an axis-perpendicular plane. The compositing order can be determined by considering the component of $\vec{tv}$ corresponding to the partition direction. If it is positive the left subtree must be rendered first; if it is negative the right subtree must be rendered first; and if it is zero both subtrees can be rendered in arbitrary order. We assume that subdivision schemes are view-independent. It is sufficient to compute a k-d tree subdivision once per time step of a data set. We compute subdivisions offline in a pre-processing step. To assign regions of the domain, i.e., leaves of the k-d tree, to individual processors they are numbered in rendering order. We assign a set of sequentially adjacent regions to each processor.

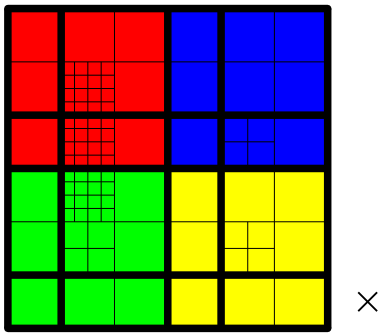


Figure 6: Uniform subdivision of root level into equal-sized blocks. Bold lines indicate boundaries between subdivision blocks. Colors indicate distribution among four CPUs (red, green, blue and yellow for CPU one to four, respectively) assuming that the data set is viewed from the lower-right corner (marked by "X" in the figure).

6.2 Uniform Root-level Subdivision

Given an AMR hierarchy, uniform root-level subdivision constructs a k-d tree of a user-specified height, that partitions AMR data into collections of cells. Each node of the tree splits its associated region into two parts of nearly equal size, i.e., equal number of cells. Figure 6 shows uniform subdivision of the AMR hierarchy from Figure 1.

Since uniform subdivision ignores grid boundaries, refined cells must be detected and processed during rendering. We implemented a solution method based on recursively descending the hierarchy. While rendering a data set, a test is performed for each grid cell checking whether it is refined by the next finer level. If a finer level exists, the r^3 refining grid cells (r being the refinement ratio) are rendered instead of the current coarser cell. The correct rendering order of refining cells is determined using the criteria described in Section 5. Each refining cell is checked recursively to determine potential further refinement.

6.3 Weighted Root-level Subdivision

Similarly to uniform subdivision, weighted root-level subdivision ignores grid boundaries of an AMR hierarchy during subdivision. The goal of this approach is to obtain a subdivision of a given

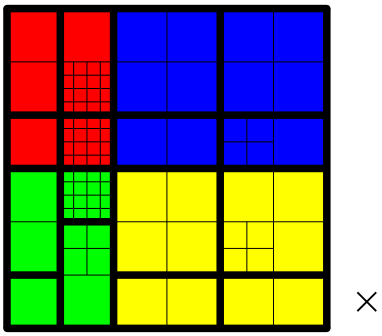


Figure 7: Weighted subdivision of root level, ignoring grid boundaries. Bold lines indicate boundaries between subdivision blocks. Colors indicate distribution among four CPUs (red, green, blue and yellow for CPU one to four, respectively) assuming that the data set is viewed from the lower-right corner (marked by "X" in the figure).

CPU type	c_0	c_1	c_2
1.0 GHz AMD Athlon	1.00	0.60	0.50
1.2 Ghz AMD Athlon	1.00	0.65	0.54
1.4 GHz AMD Athlon	1.00	0.71	0.58
2.4 Ghz Intel Xeon	1.00	0.63	0.54
375 MHz IBM Power 3	1	0.77	0.71

Table 2: Constants for weighted distribution for different processors.

AMR hierarchy into regions that will imply approximately equal rendering cost. With each region we associate an estimate of rendering cost, used as a weight. A subdivision plane is chosen using a greedy method as follows: Initially, the subdivision plane is placed in the middle of the current domain. Weights are computed for the two subdomains. If both subdomains have equal weight, the subdivision plane has optimal position and the algorithm terminates. Otherwise, the plane is moved into the subdomain with the larger associated weight. Moving the plane in this way decreases rendering cost for that subdomain while increasing rendering cost for the other one. We calculate the weight difference before and after moving the plane, and continue moving the plane as long as it decreases the weight difference. Our algorithm terminates when moving the plane increases the difference instead of decreasing it, or the partition plane would reach the border of a subdomain.

We estimate rendering cost of a subdomain based upon its number of cells. As the cost of rendering cells is dominated by generating ray segments (mainly incurred by computing evaluating the power function), we must also consider the fact that rendering refined cells is computationally more expensive than rendering unrefined cells. Therefore, we assign a weight of one to unrefined cells of the root level, i.e., $c_0 = 1$. Based on an application-specific benchmark it is possible to determine relative weights for refined cells. Table 2 shows weights for an AMR hierarchy consisting of three levels. The constants specify the times necessary to render a single cell of a given AMR level with respect to rendering times for a single cell of the root level. These constants are measured by rendering a cell of the appropriate level from a viewing direction of $\vec{t_v} = (1,1,1)$. When viewing a cell in this direction no cell faces are axis-perpendicular. A maximum number of faces must be rendered and the "footprint" of the cell on the screen has maximal size. The associated weight w of a subdomain is

$$w = \sum_{l=0}^{\#\text{Level}} n_l c_l \,, \tag{6}$$

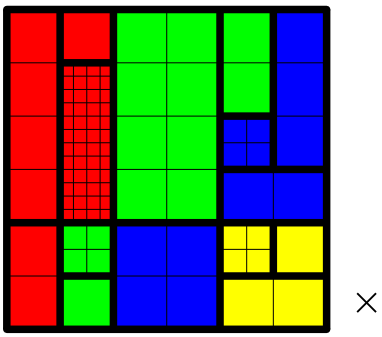


Figure 8: Homogeneous subdivision of AMR hierarchy. Bold lines indicate boundaries between subdivision blocks. Colors indicate distribution among four CPUs (red, green, blue and yellow for CPU one to four, respectively) assuming that the data set is viewed from the lower-right corner (marked by "X" in the figure).

where n_l is the number of level l cells. This sum is computed by recursively descending in the hierarchy. Figure 7 shows weighted subdivision of the root level for the AMR hierarchy from Figure 1, using relative cell weights of $c_0 = 1$, $c_1 = 0.75$ and $c_2 = 0.7$.

6.4 Homogeneous Subdivision

Both subdivision strategies discussed so far ignore the hierarchical nature of AMR data during subdivision. When using weighted root-level subdivision, only the impact on the cost of rendering a subgrid is considered for subdivision. Resulting regions usually encompass several grids of the original hierarchy, leading to data duplication and poor memory utilization. By considering grid boundaries during the subdivision step, we partition an AMR hierarchy into "homogeneous" blocks consisting only of cells from a single refinement ration. Each homgeneous block contains only cells from one grid of the original AMR hierarchy. This property allows us to avoid data duplication. Because all cells in a homogeneous block are from a constant refinement ratio, it is possible to render them efficiently, avoiding tests for refinement of individual cells and recursion.

Subdivision of an AMR data set uses only information about the AMR hierarchy. Actual data values for individual grids do not need to be loaded, and subdivision can be performed on a single machine, requiring only a small amount of memory. We construct the k-d tree level by level. For level l, we locate all leaf nodes of the current k-d tree that correspond to grids of level $l-1$. Each of these leaves is replaced by a k-d tree that is constructed as follows: We determine all grids of level l that overlap the leaf region, i.e., the region associated with the current leaf. Since grids may only partially overlap the leaf region, they are clipped against the leaf region to obtain the grid subset contained in the leaf region. Along the current subdivision direction, we store every position in subdivision direction where a refining grid starts or ends in a list.

After sorting the resulting list and removing duplicate elements, we choose the middle element of the resulting list as subdivision position. (If the list contains an even number of elements, we choose the smaller of the two middle elements as subdivision position. If the list is empty, we skip the corresponding subdivision direction.) For each of the two regions associated with a subtree of the current leaf, we identify all grids that overlap that region and clip them against the boundaries of that region. Cycling among the three axis directions, we repeat this process recursively until all leaf regions are homogeneous and overlap only a single grid. For the root level, we start with an empty tree that covers the complete domain and construct a k-d tree analogously to creating the tree for a leaf.

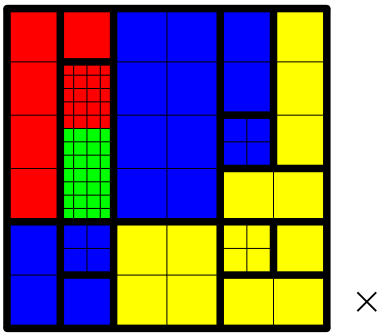


Figure 9: Weighted homogeneous subdivision of AMR hierarchy. Bold lines indicate boundaries between subdivision blocks. Colors indicate distribution among four CPUs (red, green, blue and yellow for CPU one to four, respectively) assuming that the data set is viewed from the lower-right corner (marked by "X" in the figure).

By terminating k-d tree construction after a user-specified fixed level-number it is possible to render only a part of an AMR hierarchy. Figure 8 shows the results of homogeneous subdivision of the AMR hierarchy from Figure 1. Grid subsets are ordered in back-to-front order and then distributed among processors. Each processor loads the complete partition information and renders nearly the same number of sequentially numbered grid subsets.

6.5 Weighted Homogeneous Subdivision

Weighted homogeneous subdivision uses the same k-d tree subdivision as homogeneous subdivision. Instead of distributing resulting grid subsets evenly among processors, an estimate of rendering cost is used as a weight for each leaf of the k-d tree. This weight is obtained by multiplying the number of grid cells in the leaf region by the weight of a single cell of the appropriate level. The weight of a single cell is the same as the one used in weighted root-level subdivision. Regions are distributed among processors using a greedy method. To achieve nearly equal load balance, each processor needs to render regions with a total weight of $w_{\text{Ideal}} = w_{\text{Total}}/n_{\text{Processors}}$, where w_{Total} is the rendering cost for the complete AMR hierarchy. Processor p selects its assigned regions as follows: If k is the last leaf rendered by processor $p-1$, processor p adds the remainder of that region, i.e., the part that was not rendered by the previous processor, to its "assignment list." (An exception to this rule applies to the first processor. It does not need to render any partial regions.) Starting with region $k+1$, processor p adds regions to its assignment list until the weight of the current region exceeds the difference $w_{\text{Ideal}} - w_{\text{Curr}}$. During each step, w_{Curr} denotes the sum of weights of all regions already assigned to processor p.

To achieve a more uniform distribution of weights, the last region assigned to a processor is subdivided as follows: We choose the direction perpendicular to the plane consisting of the least number of cells as partition direction. We divide the difference $w_{\text{Ideal}} - w_{\text{Curr}}$ by the weight of a slice in partition direction (i.e., the number of cells in the slice multiplied by the cell weight of the appropriate level). The result is rounded to obtain the number of slices rendered by the current processor. The remaining slices are rendered by the next processor. (An exception to this rule applies to the last processor that renders all remaining regions.) Figure 9 illustrates the differences in CPU assignments compared to homogeneous subdivision.

Each processor computes assignments for all processors. Doing so avoids the need for waiting for the previous processor to finish its own assignment computation. The index k of the last region rendered by the previous processor and a potential remainder of a subdivided region are determined locally. Performing this computation in parallel avoids added time for communication between processors.

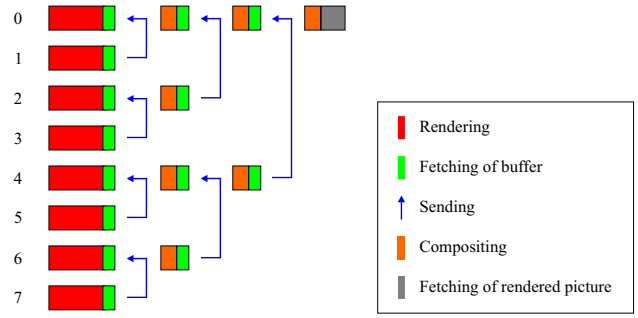


Figure 10: Parallel compositing scheme.

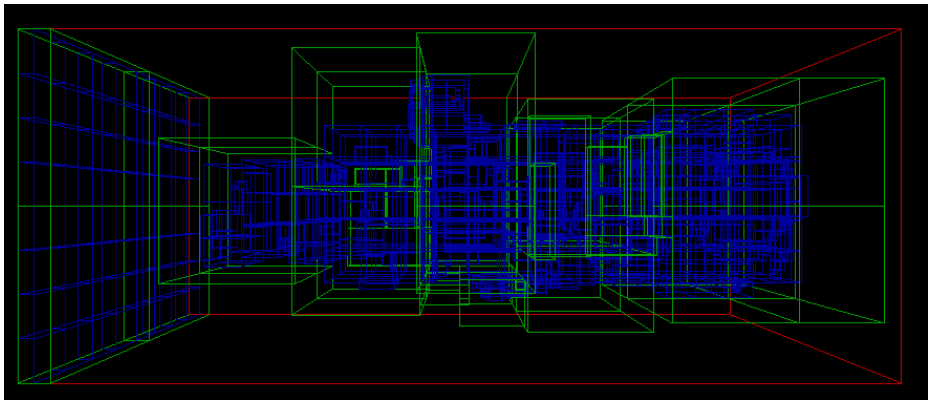
7 Compositing

When all regions are rendered, the partial images are composited. Compositing is done by using alpha blending/compositing of the partial images, see [Porter and Duff 1984]. The compositing scheme is illustrated in Figure 10. In the first step, each odd processor sends its partial image to its lower-indexed neighbor processor that performs the compositing operation. In each subsequent step i only those processors that composited an image in the previous step are considered, i.e., processors with an index of the form $k2^i$. Each processor having an associated odd value of k sends its intermediate partial image to processor $(k-1)2^i$ that performs the next compositing operation. Since regions are assigned to processors in back-to-front order, each processor can composite the partial image received from the other processor "over" the region in its own buffer. At the end of the compositing process, the final image resides in the buffer of processor zero. When transferring partial images between processors for compositing, we only transmit pixels that have been altered during rendering. We perform this step by transferring position, color and alpha value for each altered pixel. In the "fetch picture" stage this representation is converted to a bitmap.

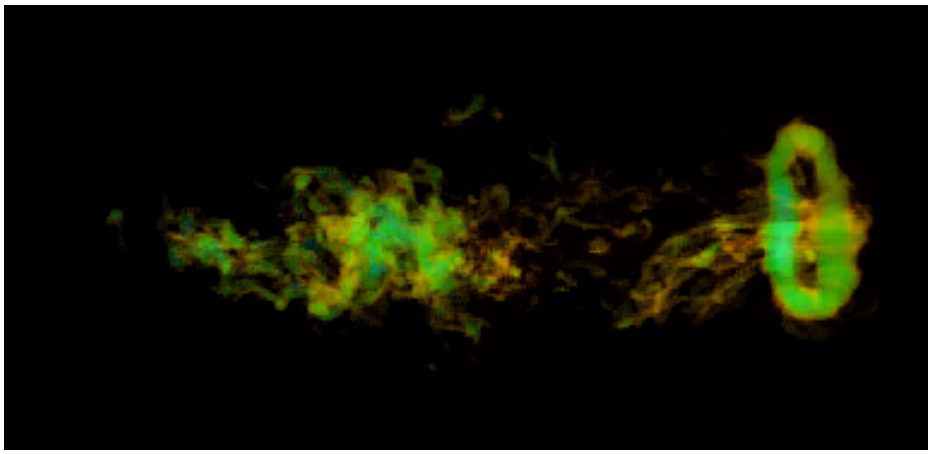
8 Results

Figure 11 shows the last time step of the "Argon Bubble" data set. We used this data, which is courtesy of the Center for Computational Sciences and Engineering (CCSE), Lawrence Berkeley National Laboratory, Berkeley, California, for testing our distribution strategies. It is the result from the last time-step in a simulation of a shock wave passing through an Argon bubble surrounded by another gas. The visualized scalar field is gas density. This simulation data is stored in AMR format using a hierarchy consisting of 885 grids in three levels. All grids in total consist of 1401504 grid cells. Homogeneous subdivision of the AMR hierarchy yields 6002 grid regions. Figure 11(a) shows the grid structure. Figure 11(b) shows the final volume-rendered image. We tested distribution strategies on the following machines:

Linux cluster This configuration is a Linux cluster consisting of four 1.2 GHz Dual-Athlon machines connected via a Gigabit network. For measurements with four processors, we used a single CPU on each machine. For measurements with eight processors, we used both CPUs on each machine. Each machine has 512 MB main memory.



(a)



(b)

Figure 11: (a) Grid structure of "Argon Bubble." The hierarchy consists of 885 grids in three levels with a root grid of $80 \times 32 \times 32$ cells. (b) Volume-rendered image of "Argon Bubble." (Data set courtesy of Center for Computational Sciences and Engineering (CCSE), Lawrence Berkeley National Laboratory, Berkeley, California)

Shared-memory machine This is a PC-based server equipped with two 2.4 GHz Intel Xeon CPUs using hyper-threading to obtain four "virtual" CPUs. The used machine has a total memory of 2 GByte RAM. We used a version of MPICH that supports the shared memory environment on that machine.

IBM SP2 Seaborg is a 10 Teraflop IBM SP RS/6000 located at NERSC's high-performance computing facility. It consists of 416 NightHawk II nodes. Each node contains 16 IBM Power3+ processors running at 375 MHz and 16–64 GBytes of shared memory. The nodes are interconnected using dual 150 Megabyte/s SP/"GX BusColony" switch adaptors forming a fat-tree topology. We used IBM's native MPI implementation.

Figure 12 shows processor utilization for rendering on a four-processor Linux cluster. As expected, uniform subdivision achieves an uneven utilization of processors. Weighted root-level subdivision achieves nearly uniform processor utilization, but requires longer rendering times than subdivisions working on homogeneous grid subsets. This behavior is due to the overhead induced by recursively descending the hierarchy. Recursive descend also causes non-local memory access pattern resulting in poor cache utilization. Working only on data of a single grid and avoiding overhead due to data inhomogeneity, homogeneous subdivision performs better than weighted root-level subdivision, even though rendering cost is not as evenly distributed. Weighted homogeneous subdivision resolves this problem and achieves good rendering speed while near-uniformly utilizing all processors.

Subdivision Strategy	Time [s]	Speedup
Uniform	14.90	2.56
Weighted Root-level	14.67	2.61
Homogeneous	12.35	3.10
Weighted Homogeneous	11.95	3.20

Table 3: Rendering times on Linux cluster using four CPUs.

Subdivision Strategy	Time [s]	Speedup
Uniform	7.83	4.89
Weighted Root-level	7.50	5.10
Homogeneous	7.10	5.39
Weighted Homogeneous	6.21	6.16

Table 4: Rendering times on Linux cluster using eight CPUs.

Tables 3 – 5 show rendering times and speedups for rendering on a Linux cluster and a shared-memory machine. Speedups are measured with respect to rendering data on a single processor using homogeneous subdivision. As expected from considering processor utilization in the previous paragraph, weighted homogeneous subdivision produces the best result. We note that times vary between subsequent runs of our framework and timings are only ac-

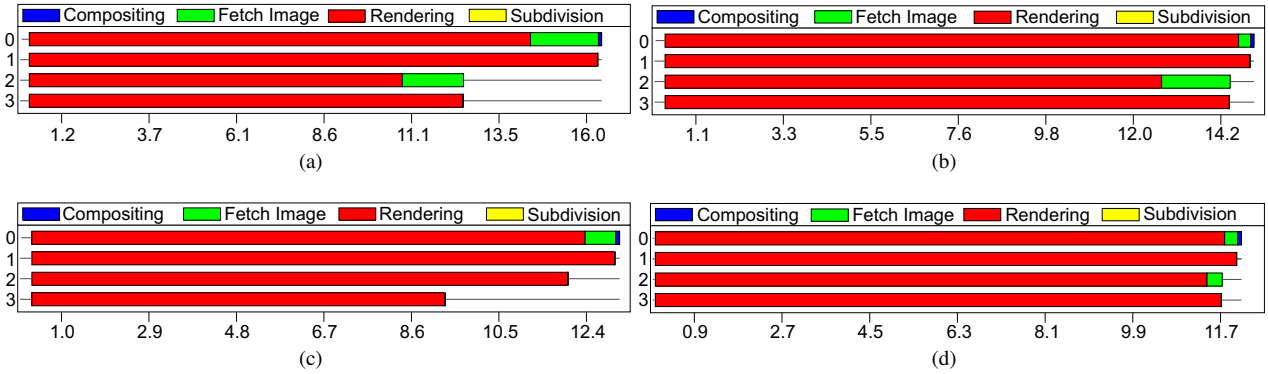


Figure 12: Processor utilization for uniform root-level subdivision (a); weighted root-level subdivision (b); homogeneous subdivision (c); and weighted homogeneous subdivision (d).

Subdivision Strategy	Time [s]	Speedup
Uniform	14.76	2.59
Weighted Root-level	14.60	2.62
Homogeneous	12.16	3.14
Weighted Homogeneous	11.23	3.40

Table 5: Rendering times on shared-memory machine.

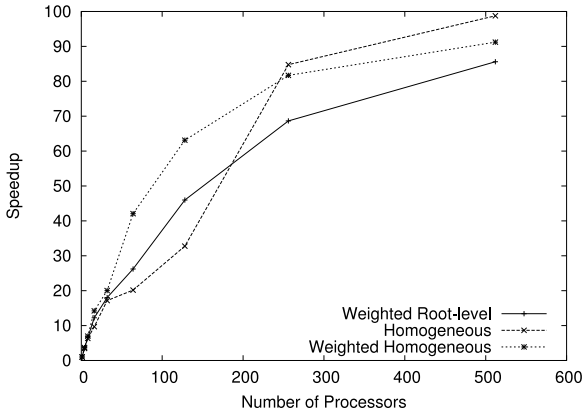


Figure 13: Speedup as function of number of processors (IBM SP2).

curate within approximately one second. Considering these facts, the speedup achieved by weighted homogeneous subdivision is satisfactory.

Table 6 shows rendering times on an IBM SP2. It shows "strong scaling" behavior, typical of increasing the number of processors while holding constant the problem size. A constant problem size typically results in less flattering scaling efficiency when compared to an approach where the problem size is scaled to be proportional to the number of processors, which is the case for "weak scaling" studies. Timing granularity for large-scale parallel applications is typically on the order of approximately one second. Thus, results utilizing more than 128 processors on that system have a lower degree of confidence than the smaller tests. Starting with 256 processors, homogeneous subdivision surprisingly performs better than weighted homogeneous subdivision. The difference in the performance of the models for the very large-scale runs is less than the timing granularity, so we only have limited confidence that these effects are actually real rather than being timing artifacts. However, we believe that these timings are consistent with the observation that the time required for assigning regions to processors is higher for weighted homogeneous subdivision. While rendering time on each processor decreases for a larger number of processors, the time required for computing the subdivision becomes the dominant rendering cost. It is also possible that the granularity of work that can be assigned becomes large compared to the total amount of work that is assigned to each processor — offering less benefit to these fine-grained optimizations.

9 Conclusions and Future Work

We have implemented and compared several distribution strategies for direct volume rendering of AMR hierarchies. Homogeneous subdivision supports efficient rendering of AMR data for different classes of machines. It allows us to avoid data duplication and employ a wide variety of rendering schemes. Homogenizing an AMR hierarchy has also been used for a variety of hardware-accelerated methods for volume-rendering AMR data. While weighted homogeneous subdivision of the domain results in near-uniform processor utilization, we plan to improve the approximation of relative cell weights. It may be beneficial to use view-dependent weights. We intend to consider inhomogeneous PC clusters consisting of machines with varying processor speed and develop subdivision/distribution methods that take these differences into account. Furthermore, we plan to develop a communication-less subdivision strategy that avoids the need for each processor to compute assignments for all other processors, resulting in less overhead and better scalability. During compositing, a major portion of time is spent on sending and receiving partial images. We plan to reduce this overhead by encoding partial images more efficiently using, for example, run-length encoding. We intend to implement binary-swap compositing [Ma et al. 1994] and compare the results.

Acknowledgments

We thank Hartmut Sprengart from the Centrum für Produktionstechnik (CCK)/Lehrstuhl für Fertigungstechnik und Betriebsorganisation (FBK) for permission to use a shared-memory Intel-Xeon machine. This work was supported by the Stiftung Rheinland-Pfalz für Innovation; by the Director, Office of Science, of the U. S. Department of Energy under Contract No. DE-AC03-76SF00098; the National Science Foundation under contract ACI 9624034 (CAREER Award), through the Large Scientific and Software Data Set Visualization (LSSDSV) program under contract ACI 9982251, and through the National Partnership for Advanced Computational Infrastructure (NPACI); the National Institute of Mental Health and the National Science Foundation under contract NIMH 2 P20 MH60975-06A2; and the Lawrence Berkeley National Laboratory.

We thank the members of the AG Graphische Datenverarbeitung und Computergeometrie, Department of Computer Science, University of Kaiserslautern, Germany, the Visualization and Graphics Research Group, Center for Image Processing and Integrated Computing (CIPIC), University of California, Davis, and the Visualization Group, Lawrence Berkeley National Laboratory.

No. of CPUs	Weighted Root-level Time [s]	Speedup	Homogeneous Time [s]	Speedup	Weighted Homogeneous Time [s]	Speedup
1	142.10	1.00	125.45	1.00	124.98	1.00
4	39.87	3.56	36.25	3.46	33.83	3.69
8	21.67	6.55	19.81	6.33	17.89	6.98
16	11.61	12.23	12.93	9.70	8.79	14.21
32	7.88	18.03	7.31	17.16	6.24	20.02
64	5.42	26.21	6.22	20.16	2.97	42.08
128	3.09	45.98	3.83	32.75	1.98	63.12
256	2.07	68.64	1.48	84.76	1.53	81.68
512	1.66	85.60	1.27	98.77	1.37	91.22

Table 6: Rendering times on IBM SP2.

References

BENTLEY, J. L. 1975. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (Sept.), 509–517.

BERGER, M., AND COLELLA, P. 1989. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics 82* (May), 64–84. Lawrence Livermore National Laboratory, Technical Report No. UCRL-97196.

BERGER, M., AND OLIGER, J. 1984. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics 53* (Mar.), 484–512.

BRYAN, G. L. 1999. Fluids in the universe: Adaptive mesh refinement in cosmology. *Computing in Science and Engineering 1*, 2 (Mar./Apr.), 46–53.

CROCKETT, T. W. 1997. An introduction to parallel rendering. *Parallel Computing 23*, 7 (July), 819–843.

GORDON, D., PETERSON, M. A., AND REYNOLDS, R. A. 1994. Fast polygon scan conversion with medical applications. *IEEE Computer Graphics and Applications 14*, 6 (Nov.), 20–27.

KÄHLER, R., AND HEGE, H.-C. 2002. Texture-based volume rendering of adaptive mesh refinement data. *The Visual Computer 18*, 8, 481–492.

KÄHLER, R., COX, D., PATTERSON, R., LEVY, S., HEGE, H.-C., AND ABEL, T. 2002. Rendering the first star in the universe – a case study. In *IEEE Visualization 2002*, IEEE Computer Society Press, Los Alamitos, California, R. J. Moorhead, M. Gross, and K. I. Joy, Eds., IEEE, 537–540.

KÄHLER, R., SIMON, M., AND HEGE, H.-C. 2003. Interactive volume rendering of large sparse data sets using adaptive mesh refinement hierarchies. *IEEE Transactions on Visualization and Computer Graphics 9*, 3 (July–Sept.), 341–351.

KAUFMAN, A. 1988. Efficient algorithms for scan-converting 3d polygons. *Computers & Graphics 12*, 2, 213–219.

KREYLOS, O., WEBER, G. H., BETHEL, E. W., SHALF, J. M., HAMANN, B., AND JOY, K. I. 2002. Remote interactive direct volume rendering of amr data. Tech. Rep. LBNL 49954, Lawrence Berkeley National Laboratory.

LAUR, D., AND HANRAHAN, P. 1991. Hierachical splatting: A progressive refinement algorithm for volume rendering. *Computer Grahpics (Proceedings of ACM SIGGRAPH 91) 25*, 4 (July), 285–288.

LEVOY, M. 1988. Display of surfaces from volume data. *IEEE Computer Graphics and Applications 8*, 3 (May), 29–37.

LIGOCKI, T. J., STRAALEN, B. V., SHALF, J. M., WEBER, G. H., AND HAMANN, B. 2003. A framework for visualizing hierarchical computations. In *Hierarchical and Geometrical Methods in Scientific Visualization*, G. Farin, B. Hamann, and H. Hagen, Eds. Springer Verlag, Heidelberg, Germany, Jan., 197–204.

MA, K., AND CROCKETT, T. W. 1997. A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data. In *IEEE Parallel Rendering Symposium*, IEEE Computer Society Press, Los Alamitos, California, J. Painter, G. Stoll, and K. Ma, Eds., IEEE, 95–104.

MA, K.-L., PAINTER, J. S., HANSEN, C. D., AND KROGH, M. F. 1994. Parallel volume rendering using binary-swap composition. *IEEE Computer Graphics and Applications 14*, 2 (July), 59–67.

MA, K.-L. 1999. Parallel rendering of 3D AMR data on the SGI/Cray T3E. In *Proceedings of Frontiers '99 the Seventh Symposium on the Frontiers of Massively Parallel Computation*, IEEE Computer Society Press, Los Alamitos, California, IEEE, 138–145.

MACNEICE, P., OLSON, K. M., MOBARRY, C., DE FAINCHTEIN, R., AND PACKER, C. 2000. Paramesh: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications 126*, 3 (Apr.), 330–354.

MAX, N. L. 1993. Sorting for polyhedron compositing. In *Focus on Scientific Visualization*, H. Hagen, H. Müller, and G. M. Nielson, Eds. Springer-Verlag, New York, New York, 259–268.

MAX, N. L. 1995. Optical models for volume rendering. *IEEE Transactions on Computer Graphics 1*, 2, 99–108.

NORMAN, M. L., SHALF, J. M., LEVY, S., AND DAUES, G. 1999. Diving deep: Data management and visualization strategies for adaptive mesh refinement simulations. *Computing in Science and Engineering 1*, 4 (July/Aug.), 36–47.

PARK, S., BAJAJ, C., AND SIDDAVANAHALLI, V. 2002. Case study: Interactive rendering of adaptive mesh refinement data. In *IEEE Visualization 2002*, IEEE Computer Society Press, Los Alamitos, California, R. J. Moorhead, M. Gross, and K. I. Joy, Eds., IEEE, 521–524.

PORTER, T., AND DUFF, T. 1984. Compositing digital images. *Computer Graphics (Proceedings of ACM SIGGRAPH 84) 18*, 3 (July), 253–259.

SABELLA, P. 1988. A rendering algorithm for visualizing 3D scalar fields. *Computer Graphics (Proceedings of ACM SIGGRAPH 88) 22*, 4, 51–58.

WEBER, G. H., HAGEN, H., HAMANN, B., JOY, K. I., LIGOCKI, T. J., MA, K.-L., AND SHALF, J. M. 2001. Visualization of adaptive mesh refinement data. In *Proceedings of the SPIE (Visual Data Exploration and Analysis VIII, San Jose, CA, USA, Jan 22–23)*, SPIE – The International Society for Optical Engineering, Bellingham, WA, R. F. Erbacher, P. C. Chen, J. C. Roberts, C. M. Wittenbrink, and M. Groehn, Eds., vol. 4302, SPIE, 121–132.

WEBER, G. H., KREYLOS, O., LIGOCKI, T. J., SHALF, J. M., HAGEN, H., HAMANN, B., AND JOY, K. I. 2001. Extraction of crack-free isosurfaces from adaptive mesh refinement data. In *Proceedings of the Joint EUROGRAPHICS and IEEE TCVG Symposium on Visualization, Ascona, Switzerland, May 28–31, 2001*, Springer Verlag, Wien, Austria, D. S. Ebert, J. M. Favre, and R. Peikert, Eds., EUROGRAPHICS and IEEE TCVG, 25–34, 335 (Color plate).

WEBER, G. H., KREYLOS, O., LIGOCKI, T. J., SHALF, J. M., HAGEN, H., HAMANN, B., JOY, K. I., AND MA, K.-L. 2001. High-quality volume rendering of adaptive mesh refinement data. In *Vision, Modeling, and Visualization 2001*, Akademische Verlagsgesellschaft Aka GmbH, Berlin, Germany and IOS Press BV, Amsterdam, Netherlands, T. Ertl, B. Girod, G. Greiner, H. Niemann, and H.-P. Seidel, Eds., 121–128, 522 (Color plate).