Visualizing Large 3D Geodesic Grid Data with Massively Distributed GPUs

Jinrong Xie[*] University of California-Davis, USA

Hongfeng Yu[†] University of Nebraska-Lincoln, USA

Kwan-Liu Ma[‡] University of California-Davis, USA

Abstract

Geodesic grids become increasingly prevalent in large weather and climate applications. The deluge amount of simulation data demands efficient and scalable visualization capabilities for scientific exploration and understanding. Given the unique characteristics of geodesic grids, no current techniques can scalably visualize scalar fields defined on a geodesic grid. In this paper, we present a new parallel ray-casting algorithm for large geodesic grids using massively distributed GPUs. We construct a spherical quadtree to adaptively partition and distribute the data according to the grid resolution of simulation, and ensure a balanced workload assignment over a large number of processors from different view angles. We have designed and implemented the entire rendering pipeline based on the MPI and CUDA architecture, and demonstrated the effectiveness and scalability of our approach using an example of large application on a supercomputer with thousands of GPUs.

1 Introduction

Advanced supercomputing techniques and systems allow scientists to conduct simulations with detailed numerical weather and climate models. Geodesic grids have become increasingly prevalent in the development of models [1, 17]. This type of grid can facilitate scientists to model the Earth's surface with higher resolutions and higher numerical stability, leading to a simulation of an unprecedented scale. Such a simulation can possibly generate tera- or peta-bytes of data that are typically time-varying and multivariate. The sheer size of data requires scalable and interactive visualization techniques for agile exploration and timely weather and climate prediction. Parallel visualization provides a viable solution to address the vast amount of simulation data, in that data are partitioned and distributed among multiple processing units and visualization calculations are conducted in a divide-and-conquer manner. A parallel visualization algorithm with a balanced workload assignment can achieve scalable performance over a large number of processors, and make it possible to interactively explore massive data.

Geodesic grids were first introduced by Williamson [24] and Sadourny et al. [18] for meteorological applications. A geodesic mesh is constructed by subdividing an icosahedron embedded in a sphere, where a simple bisection operation is iteratively applied on the edges to refine the grids [17]. The iteration number of subdivision can be different over different regions of the sphere, such that more iterations are conducted on the regional areas of interest to generate higher-resolution grids, while lower-resolution grids are placed for the remainder of the surface. The resulting geodesic mesh consists of spherical Voronoi polygons, where most of them are hexagons and the remainder can be pentagons or heptagons. The spherical polygon mesh is then scaled and duplicated along the direction perpendicular to the Earth surface but at different altitudes to construct a set of spherical layers. These layers together form a 3D mesh (or cage) used to model oceanic or atmospheric

[*]e-mail:jrxie@ucdavis.edu

[†]e-mail:yu@cse.unl.edu

[‡]e-mail:ma@cs.ucdavis.edu

IEEE Symposium on Large Data Analysis and Visualization 2014
October 9–10, Paris, France
978-1-4799-5215-1/14/$31.00 ©2014 IEEE

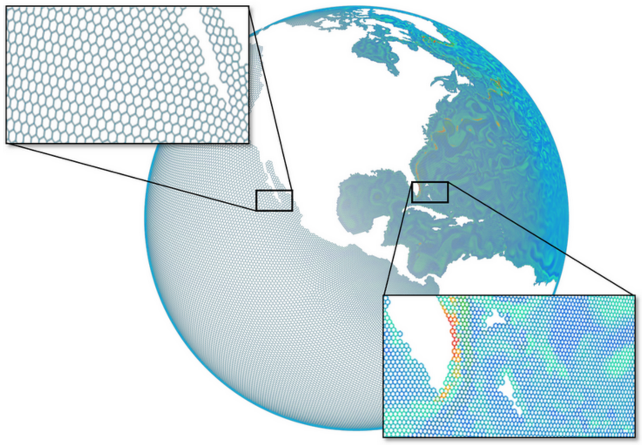


Figure 1: An example of spherical geodesic grids covering the ocean of the Earth surface. The resolutions of Voronoi polygons vary across different regions: higher-resolution grids are used in the regional areas of interest, while lower-resolution grids are placed for the remainder of the surface. Note that this example only displays the mesh of the top surface. For a real simulation, this spherical mesh is scaled and duplicated to construct a set of layers along the direction perpendicular to the Earth surface, resulting in a 3D mesh.

behavior in 3D. We refer interested readers to Xie's work [25] for an overview of grid construction. Figure 1 shows the top surface of a 3D geodesic mesh used in an ocean basin simulation, where the mesh is refined in specific regions of interest, lower-resolution grids are used in other ocean areas, and no grid is placed in the continental area.

Although extensive research has been carried out on parallel visualization, several fundamental challenges have prevented a direct application of current solutions on geodesic grids. First, most parallel visualization solutions rely on data partitioning and distribution schemes that are designed in Cartesian coordinates, while geodesic grids are constructed in the spherical coordinate system. The data representations used in geodesic grids are fundamentally different from grids in Cartesian coordinates. Existing data partitioning and distribution schemes cannot be directly applied to handle geodesic grids. Second, the spherical structure introduces unique visualization requirements in that only the scalar fields in front are desired to be visualized. Given the multiresolution nature of geodesic grids, the data density of visible regions can vary significantly across different view angles. Thus it is difficult to estimate the rendering cost using conventional methods for scientific volume data.

In this paper, we introduce a scalable parallel solution to interactively visualize large-scale geodesic grid data using multiple GPUs. Based on a careful characterization of geodesic grid visualization, we design a new data partitioning and distribution scheme that employs a spherical quadtree to decompose and index the multiresolution spherical grids. This spatial data structure enables us to accurately estimate the rendering load for regions with different resolutions. Moreover, we use the quadtree structure to scatter the regions among the processors and ensure that each processor can be assigned an approximately equal amount of workloads from any viewing direction. Therefore no processors will be idle during the rendering, and the maximum parallelism can be achieved. Our im-

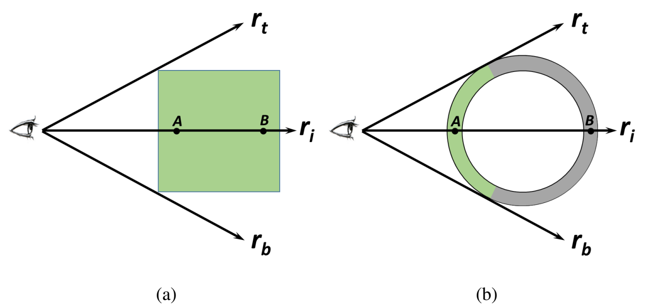


Figure 2: Ray-casting of scalar fields. r_t and r_b correspond to the boundary of the view frustum. A and B are the sampling points along a ray r_i. (a): For a general scalar field, both A and B can be visible, and the entire domain (in green) is involved in the rendering calculation. (b): For a scalar field defined on a spherical mesh covering the Earth, we only render A in the front region (in green), while excludes B in the back region (in gray) that is invisible.

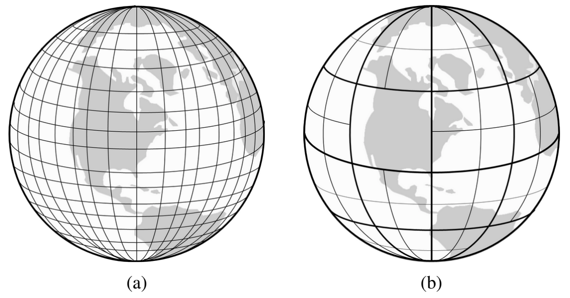


Figure 3: (a) shows a regular partitioning scheme along the latitude and longitude. (b) shows our spherical quadtree based partitioning scheme, which can achieve load balancing with a smaller number of regions and a lower cost of parallel image compositing.

plementation of the entire rendering pipeline is based on the MPI and CUDA architecture and can be directly executed on state-of-the-art supercomputers. The scalability and effectiveness of our solution have been demonstrated using the full extent of geodesic grid data from real-world simulations. Great 3D details can be delivered at an interactive rate to scientists for exploration. In addition, our approach directly takes the original simulation data as input, and thus can be readily extended in support of in-situ visualization during simulation time.

2 Related Work

While researchers have extensively studied visualization of structured and unstructured grids, there is a lack of visualization tools for geodesic grids. In the scientific literature on numerical climatological study, mostly simple polygon renderings are used to visualize the scalar field defined on the top surface of 3D geodesic grids [7]. Such a rendering cannot reveal complex 3D interior structures embedded in grids that are related to detailed ocean processes and cloud processes in the atmosphere.

It is possible to first convert geodesic grid data into conventional representations (such as tetrahedral grid) and then apply suitable algorithms (such as cell-projection [20] or ray-casting [6]) to visualize 3D scalar fields. However, this solution requires computation and storage overhead that can be prohibitively high for current tera/petascale and future exascale simulations. To address this issue, Xie et al. [25] presented a GPU-based ray-casting algorithm that can directly visualize 3D scalar fields defined on raw geodesic grids without data transformation. Their approach achieves interactive rendering rates using a single GPU.

When data size is larger than the available memory capacity of a single machine, a common strategy is to use multiple machines to perform visualization in parallel. Researchers have developed plenty of parallel visualization algorithms for large data. One of the main focuses is to achieve high quality rendering while maintaining scalable performance with an increasing number of processors. Ma et al. [10] developed a parallel cell-projection algorithm for 3D unstructured data from aerodynamics applications. They used a round robin distribution of data cells to achieve an effective static load balancing among the processors. Parker et al. [15] conducted ray tracing using shared-memory multiprocessor machines. Their method enables an interactive isosurface visualization of large volume data with high parallel efficiency. Leaf et al. [8] developed a parallel volume rendering algorithm for large-scale Adaptive Mesh Refinement (AMR) data. They partitioned AMR data into convexly-bounded chunks and distributed them among multiple GPUs using static load balancing to perform distributed ray-casting.

Recent efforts have been made to further improve the scalability of parallel visualization to be on par with simulations, thus enabling in-situ visualization to address future exascale supercomputing challenges [19]. Researchers have directly integrated visualization [26, 4] into simulation routines to operate on in-memory simulation data. We also take in-situ visualization into account in our design, in that our approach can directly process the original grid data used in simulations and achieve high parallel efficiency comparable to simulations.

3 Characterizing Geodesic Grid Visualization

A balanced workload assignment is the key to the scalability of parallel visualization algorithms. Given the unique visualization requirements from geoscience applications, current methods to estimate the workload associated with conventional mesh structures cannot be applied to geodesic grids.

For conventional mesh structures, the entire domain of a scalar field is often counted in workload estimation regardless of view angles. This is because during volume rendering of the field users can adjust transfer functions and assign different opacities to different regions, which implies that the entire domain can be involved in the rendering calculation along an arbitrary ray. As shown in Figure 2(a), along a ray r_i for rendering a general scalar field, both the sampling points A and B can be perceived with a certain configuration of transfer function. Therefore, when designing a parallel visualization algorithm, we may assign the regions containing A and B to different processors, and all processors can be involved in rendering from any view angle.

However, in a geoscience application, scientists typically merely focus on the scalar field of the front Earth surface towards viewers. For the phenomena over the other region, they can rotate the sphere and bring the region to the front for observation. As shown in Figure 2(b), for spherical geodesic grids covering the Earth surface, we just render the sample points in the front visible region along a ray r_i. Although it is technically feasible to make A and B visible, a display of both points can preclude a clear observation and thus is rarely applied in practice. In this case, if we still simply assign the regions of A and B to different processors, some processors can be idle during the rendering if the corresponding regions have been rotated to the back, which may lead to severely unbalanced workloads among the processors.

In addition, as shown in Figure 1, the mesh existence and density can vary greatly in a model of the Earth surface. Thus, rendering workloads can be dramatically different with respect to different view angles. For example, showing the Pacific Ocean would incur a larger amount of rendering calculations compared to a display of the Americas due to a lack of grids in the continental regions. Such use of unstructured and variable-density grids exacerbates the issue of workload assignment, in that the amount of grids assigned to different processors must be carefully balanced with a holistic consideration of view angles and grid resolution distributions.

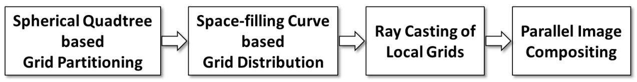

Spherical Quadtree based Grid Partitioning
Space-filling Curve based Grid Distribution
Ray Casting of Local Grids
Parallel Image Compositing

Figure 4: The major steps of our parallel volume rendering framework.

4 Parallel Volume Rendering Framework

A parallel rendering framework is typically comprised of the stages of data partition, distribution, rendering, and image compositing. Given the characterization of geodesic grid visualization, we have taken several design considerations into account for the development of a framework.

First, there are three basic parallel rendering approaches, namely, sort-first, sort-middle, and sort-last [13]. Although the rendering cost of geodesic grids is largely view-dependent, an image-based workload partition scheme may constantly require communication between processors to exchange simulation data when changing views. This communication cost can be prohibitively high for processing large climate simulation data. To this end, we choose sort-last parallel rendering because of its simple workload decomposition for achieving load balancing and no communication overhead involved in the rendering stage.

Second, data partition and distribution are conducted before the rendering stage in sort-last parallel rendering. In order to maximize the parallelism, it is imperative to keep all the processors busy in rendering visible regions. However, if a processor is only assigned one or a few partitions, it will be idle when its regions are facing away from the viewer. As illustrated in Figure 2(b), one intuitive idea is to assign the opposite regions along a ray, such as the regions containing both A and B, to one processor. That is, the processor will be always busy whenever a user is facing towards either A or B. But it is possible for the processor to become idle again when the viewing direction is perpendicular to AB. Thus we need to design a more sophisticated partition and distribution strategy to allow each processor to have visible regions to render from any viewing direction.

Third, besides visible regions, each processor would also be ideally assigned a roughly equal amount of rendering workload from any viewing direction. The data size of each region depends on its local mesh resolution. However, the raw mesh of a simulation data mainly contains the connectivity information rather than the mesh densities. Thus it requires us to find a way to index the data according to the variation of grid resolutions, such that we can quickly quantify the mesh density and then compute the data size for any given region. This functionality is vital for us to accurately estimate the regional rendering cost for load balancing.

These considerations can lead to a design in which we can regularly decompose the spherical surface into a set of patches along the latitude and longitude lines, as shown in Figure 3(a). The number of the patches is sufficiently larger than the number of processors, and then we randomly distribute the patches among the processors. Hence each processor can be assigned the regions scattered over the surface, and a portion of regions are visible from a viewing direction. In addition, by randomization, the amount of data assigned to each processor can be roughly equal. This design is based on fine-grained partitioning and randomization, which is commonly used in distributed computing for load balancing. However, a large number of decompositions can also increase the number of the partial images generated by each processor. These partial images of each processor typically cannot be composited locally, because they correspond to a set of scattered regions whose projections may not be continuous in depth. Thus a significant overhead will be introduced in the parallel image compositing stage for merging a large number of partial images into a final image, and becomes the main performance bottleneck of the entire pipeline.

We design our sort-last parallel volume rendering framework for geodesic grids to address these issues. We first construct a spherical quadtree to cover the surface of a geodesic mesh, as shown in Figure 3(b). The quadtree is refined adaptively according to the count of regional grid cells. In this way, we can quantify the mesh density within the region of any quadtree node. We then partition the grids into the regions corresponding to the leaf nodes of the quadtree, and distribute them among the processors according to the traversal order of the leaf nodes. Hence we can not only control the total number of regions, but also ensure that each processor is assigned a number of regions scattered across the spherical surface. No processors will be idle for any viewing direction, and the rendering load will be balanced among the processors. After each processor renders its assigned regions, we use parallel image compositing to generate the final image. Our approach can lower the number of partitions and significantly reduce the overhead of parallel image compositing compared to the conventional fine-grained partitioning scheme. Figure 4 shows the major steps of our approach.

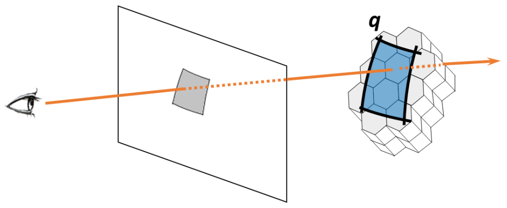


Figure 5: Ray-casting of a quadrant. Given a quadrant q (in blue), we can project it onto a 2D screen space. The gray area corresponds to the pixels of projection. A ray is casted from each pixel to penetrate the grid cells covered by q, and the color and opacity values of sample points are accumulated along the ray to generate the final color of the pixel.

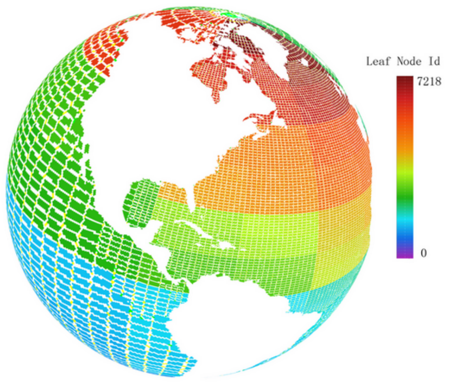


Figure 6: Rendering of a quadtree constructed from a real simulation data.

4.1 Spherical Quadtree based Grid Partitioning

A quadtree is a commonly used data structure to partition a 2D space. A typical quadtree is constructed by recursively subdividing the space into four quadrants or regions until certain criteria are reached. The concept of quadtrees can be naturally extended to decompose spherical surfaces [9, 23, 21] in that the decomposition can be conducted along the latitude and longitude lines instead of X- and Y-axes in a 2D square space.

We use the quadtree to partition spherical geodesic grids such that each quadrant or region will be associated with an approximately equal amount of rendering loads. To achieve this goal, we first need to estimate the rendering cost of a quadrant q. Without loss of generality, we use ray-casting to volume render geodesic grids. As shown in Figure 5, we first project q onto a 2D screen space, and then cast a ray from each pixel to penetrate the grid cells covered by q. For each ray, we sample the scalar field and accumulate the color and opacity values of sampling points to compute the final color of the pixel. Thus the rendering cost is proportional to the number of pixels and the number of cells. We note that the number of pixels depends on the projection and the area of the region. For a static view, the number of pixels projected from different regions can be different. But if we allow users to interactively view the spherical surface from any direction, the amortized number of pixels projected from a region is proportional to the area of the region, because each region on a sphere has an equal probability to be viewed. Thus, for a quadrant q, the rendering cost C_q, is estimated as a linear function of its area S_q and its number of cells G_q:

$$C_q = kS_qG_q \tag{1}$$

where k is a constant, and S_q is computed according to the latitudes lat_1 and lat_2 and the longitudes lon_1 and lon_2 that bound q[1].

This cost model guides us in constructing a quadtree. Given a geodesic mesh, we first use Equation 1 to estimate the total rendering cost C_t using the total spherical surface area and the total cell number. Assume that the number of processors is N and each processor will be assigned m quadrants. The average rendering cost C_{avg} of each quadrant is computed as

$$C_{avg} = \frac{C_t}{mN}. \tag{2}$$

We then start to recursively subdivide the spherical surface to construct the quadtree. We stop subdividing a quadrant if its estimated rendering cost is smaller than C_{avg} or the total quadrant number is larger than mN. In this way, we can construct a quadtree where each quadrant is associated with a similar rendering cost.

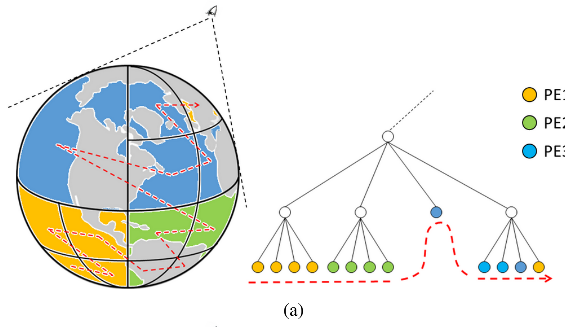
According to the Earth satellite constellation design, three satellites spaced equally around the equator can cover most of the Earth [2]. If we imagine that a user's view point is a satellite around the spherical mesh, this implies that each processor needs to be assigned at least 3 quadrants scattering on the surface. Thus from any viewing direction, a processor can have at least one quadrant that is visible, which can prevent the processor from becoming idle. In practice, we find that $m = 5$ provides us a good performance result. Figure 6 shows a quadtree constructed from a real simulation data. We can clearly see that the finer-grained quadrants are generated to cover the higher-resolution regional areas of interest, while the coarser-grained quadrants cover the remains of ocean area, which matches the distribution of grid resolutions in Figure 1.

With our spherical quadtree based partitioning, the shape of the quadrants can be different across the sphere: they are close to be rectilinear for regions around the equator, but are triangular for regions around the poles, as shown in Figure 3(b). Our cost model considers the surface area of quadrants, and the workload estimation is independent of the location and the shape of a quadrant.

4.2 Space-filling Curve based Grid Distribution

It is desired to assign each processor with a set of quadrants that are as scattered as possible. A straight-forward approach is to randomly assign the quadrants among the processors. However, we can achieve a more appropriate assignment by leveraging the spatial locality encoded in a quadtree. If we use the linear quadtree technique [5] to encode and distinguish quadrants, a pre-order traversal of quadrants leads to the well-known space-filling curve which groups the spatial nearby quadrants together on the spherical surface. Figure 7 shows an example of spherical surface decomposition and the corresponding quadtree. The traversal of leaf quadrants from left to right is equivalent to the zigzag on the spherical surface. Therefore, a strong spatial locality can be clearly identified for these quadrants. This property of quadtree has been widely used in the optimization of data layout. For example, the grid cells can be linearized and saved in persistent storage according to the space-filling curve obtained by quadtree. This storage pattern can guarantee contiguous reads/writes, and improve I/O performance [3].

[1]Our implement is similar to the `areaquad` function of MATLAB to compute surface area of latitude-longitude quadrangle [12].



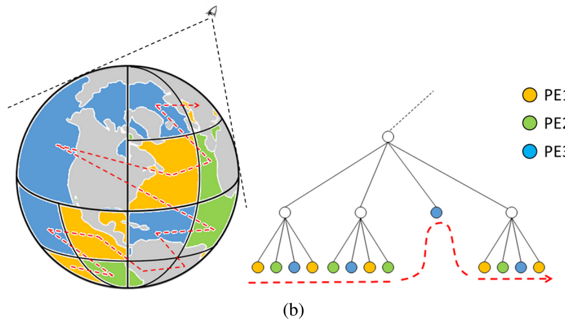


Figure 7: The spatial decomposition of a spherical surface and its corresponding quadtree. A pre-order traversal of quadrants is equivalent to the space-filling curve on the spherical surface. (a) shows that we evenly assign the quadrants among three processors from left to right in the quadtree, and the distribution of their regions is contiguous along the space-filling curve. In this case, each processor's regions may not be always visible from different viewing directions. (b) shows that we assign the quadrants among three processors in round robin, and the neighboring regions are largely assigned to different processors. In this case, a portion of a processor's regions can be visible from any viewing direction.

If we assign the processors along the space-filling curve for parallel visualization, each processor will be responsible for contiguous regions on the surface. Figure 7(a) illustrates such an assignment for three processors distinguished in different colors. However, the regions assigned to a processor can be occluded from certain view points. For example, as shown in Figure 7(a), the green regions cannot be perceived if the viewer is around the north pole, and the processor PE2 will become idle.

We note that the spatial locality becomes the best along the space-filling curve of the decomposition. On the other hand, we may achieve the *worst* locality using the space-filling curve in the opposite way to favor visualization workload assignment. Instead of a contiguous assignment, we distribute the quadrants among the processors in round robin along the space-filling curve. Given a sufficient number of processors, this distribution can guarantee that the neighboring regions are assigned to different processors. As shown in Figure 7(b), the regions of each processor scatter across the spherical surface, and a processor always has a portion of regions that are visible from any viewing direction. Thus, we can keep all processors busy in rendering. In addition, each processor is still assigned an equal number of quadrants, and each quadrant corresponds to a nearly equal amount of rendering cost. This assignment enables our solution to achieve load balancing during users' interactive exploration.

4.3 Ray-Casting of Local Geodesic Grids

After we partition and distribute geodesic grids among the processors, each processor starts to render its local regions, where the GPU-based ray-casting algorithm developed by Xie et al. [25] is employed in this stage.

The algorithm is tailored to process geodesic grids with minimal overhead. First, we directly use the original Voronoi polygonal mesh of simulations in rendering without any intermediate grid transformation. In particular, a set of table-based representations are deployed to manage the mesh in GPU memory for efficient data access. Then, during the process of ray-casting, we leverage the properties of geodesic grids and march rays using the 2D connectivity information of the outer layer. Hence, we do not need to reconstruct full 3D connectivity information, and can further reduce memory and computing overhead. To achieve high quality rendering, an analytic solution has been designed to reconstruct the signal within a geodesic grid cell for scalar value interpolation, gradient estimation, and ray integration. The accuracy of the analytic scalar and gradient interpolation is comparable to the results achieved by central difference numerical computations in simulations.

Each processor iterates through its assigned quadrants and renders the grid cells of each quadrant into a partial image using Xie's algorithm. The rendered results feature high image quality, less memory overhead, and higher computing performance. In addition, no communication is needed in this stage, and thus local grid rendering can scale well given our partitioning and distribution scheme for load balancing.

4.4 Parallel Image Compositing

The partial images rendered by each processor need to be composited (i.e. back-to-front alpha blending) to generate the final image. The parallel image compositing stage requires inter-processor communication, and can become expensive when the number of partial images increases. The most representative parallel image compositing algorithms include direct send [14] and binary swap [11]. Direct send is simple and easy to implement; however in the worst case it needs to exchange $N(N-1)$ messages among N compositing processors, introducing link contention due to its nature of all-to-all communication pattern. Binary swap uses a binary tree style compositing process and reduces the number of messages from $N(N-1)$ to $Nlog(N)$. However, to achieve the best performance, binary swap needs the number of processors to be an exact power-of-two.

Yu et al. [27] presented the 2-3 swap image compositing algorithm that combines the advantages of direct send and binary swap. On one hand, 2-3 swap can be as flexible as direct send in that it can use any number of processors. On the other hand, 2-3 swap involves the number of messages bounded by $O(Nlog(N))$, which is as efficient as binary swap. Peterka [16] presented the Radix-k algorithm that also unifies direct-send and binary swap and has the similar complexity as 2-3 swap. However, it is non-trivial to configure Radix-k for achieving optimal performance [22].

We use 2-3 swap in this work for parallel image compositing. This algorithm has been used in several large-scale parallel rendering applications, and demonstrated its great flexibility and scalability over ten thousands of processors [26]. As we discussed in Section 4.1, each processor can be assigned m quadrants, and the total number of partial images is mN for N processors. Because of the spatial discontinuity of the quadrants assigned to each processor, these partial images are not necessarily contiguous in depth on a processor, and cannot be blended locally before being sent to the other processors. Therefore, one message is required for one partial image in the worst case, and the total number of messages is bounded by $O(mNlog(mN))$. Compared to the fine-grained partitioning and distribution scheme discussed in Section 4, our scheme can minimize the value of m by leveraging spherical quadtrees, and significantly reduce the image compositing cost.

	GCRM	MPAS
resolution	28km	15∼75km
# cells	655362	253746
# vertices	1310720	517338
# edges	1966080	771377
# layers	60	40
# time steps	40	12

Table 1: The GCRM and MPAS data sets used in our evaluation. Both data sets contain multiple variables.

4.5 Implementation

We have employed MPI and CUDA to implement our framework on heterogeneous supercomputers where each node contains both CPUs and GPUs. Given a spherical geodesic grid data set, we first let each processor read the whole mesh information from storage. Because only the connectivity information of the 2D outer layer is stored in the original Voronoi polygonal mesh, the cost of this I/O operation is marginal. Then each processor independently constructs a quadtree using CPUs. The maximum depth of the quadtree depends on the processor number and the grid cell number. The decomposition of quadrants is conducted along the latitude and longitude lines of the sphere. Each processor generates an identical quadtree for the same configurations. Given the total processor number and its rank, each processor independently traverses the tree and finds the set of quadrants that it needs to be responsible for.

The grid cells are organized in the leaf quadrants according to the spatial decomposition of the quadtree. For the grid cells crossing the boundaries of the leaf quadrants, we duplicate them in the relevant quadrants for interpolating scalar values along the boundaries. During the ray-casting of a quadrant, we terminate a ray if we step out of the boundary of the quadrant rather than the boundary of the cells. In this way, we can generate the partial images aligned with the quadrants to facilitate image compositing. After each processor determines its own set of cells, all processors collectively fetch the simulation data from storage using MPI-IO.

Different from Xie's work [25], we do not use OpenGL to render the visible surface of the Voronoi polygonal mesh in this work. Instead, we implement the projection, clipping, and rasterization stages in CUDA for off-screen rendering. Thus, our implement can be executed in an environment without any OpenGL or graphics support. The ray-casting of local grids is implemented using the same method as Xie's [25], where ray marching, interpolation, gradient estimation, and ray integration are entirely conducted in GPU memory using CUDA. After each processor generates its local partial images, these images are first transferred from GPU memory to CPU memory. Then we use the 2-3 swap algorithm to composite the partial images from all processors into the final image, where the communication of 2-3 swap is implemented using MPI and the image blending is performed using CPUs.

We put our CUDA code in a special header file by exploiting the similarity between C/C++ code and CUDA code. We can choose our code to be complied into a CPU or GPU executable according to the availability of GPUs in a system. Thus our implementation is more generic and highly compatible to both homogeneous systems and heterogeneous systems.

5 Results

We have evaluated our framework using two data sets, where one is generated from the Global Cloud Resolving Model (GCRM) and the other is from the Model for Prediction Across Scales (MPAS). Both GCRM and MPAS are developed based on geodesic grids where mesh density varies over the Earth according to the distribution of regions of interest. However, GCRM is mainly used to

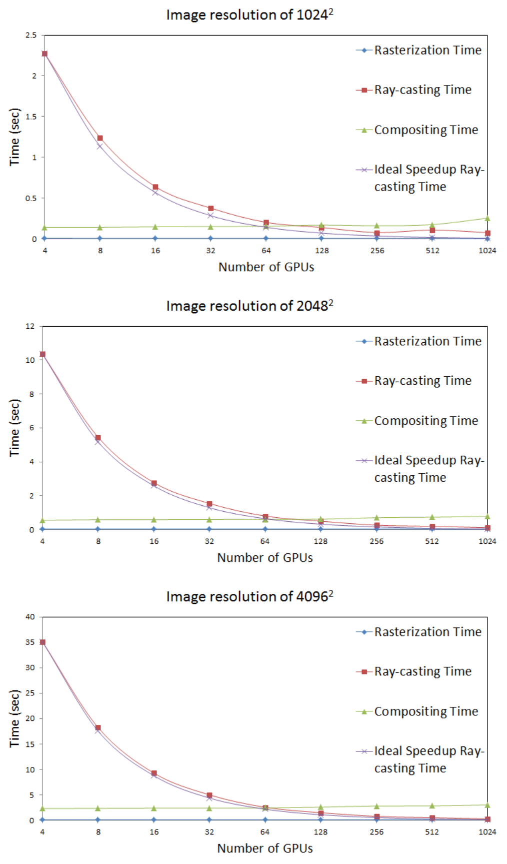


Figure 8: The timing results of the GCRM data set. We measured the timing of the rasterization, ray-casting, and image compositing operations with the number of GPUs ranging from 4 to 1024 and the output image resolutions of 1024^2 (top), 2048^2 (middle), and 4096^2 (bottom). The timing results are plotted in a logarithmic scale.

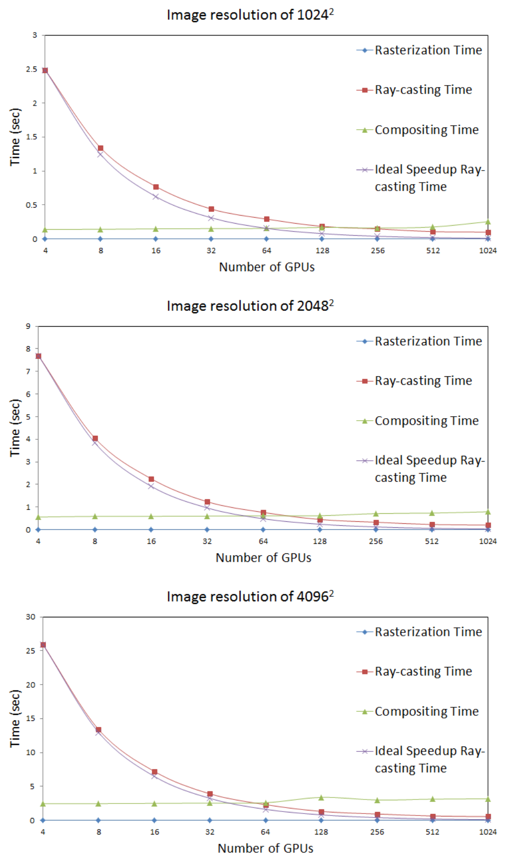


Figure 9: The timing results of the MPAS data set. We measured the timing of the rasterization, ray-casting, and image compositing operations with the number of GPUs ranging from 4 to 1024 and the output image resolutions of 1024^2 (top), 2048^2 (middle), and 4096^2 (bottom). The timing results are plotted in a logarithmic scale.

model cloud processes in atmosphere, and a GCRM mesh covers the entire sphere. MPAS is mainly used to model ocean processes such that the continental areas are not covered in a MPAS mesh. In general, compared to a GCRM mesh, a MPAS mesh exhibits more variation in grid resolutions over the sphere. Table 1 lists the detailed information of the GCRM and MPAS data sets.

We performed our experimental study on Titan, a Cray XK7 system at Oak Ridge National Laboratory. The system contains 18,688 compute nodes, and each node has a conventional 16-core AMD Opteron CPU and an NVIDIA Tesla K20 GPU accelerator with a total 38GB of memory. The compute nodes are connected through a Cray Gemini interconnect.

We conducted a strong scaling test on our approach with an increasing number of GPUs from 4 to 1024. We rendered the data sets using 3 different image sizes, including 1024^2, 2048^2, and 4096^2, and from 10 different viewing directions that are evenly distributed across the sphere. For each viewing direction, we measured the timing results of the major operations, including rasterization, ray-casting, and parallel image composting. The timing results are then averaged over the viewing directions.

Figure 8 shows the timing results for rendering the GCRM data set. The rasteraization time is nearly negligible for all three image resolutions. Our CUDA-based implementation of rasterization can render millions of polygons interactively. For the GCRM data, our rasterization approach achieves a rate of 30 frames per second for 2048^2 images and a rate of 10 frames per second for 4096^2 images using 4 GPUs. The performance is comparable to an implementation using the native OpenGL on similar GPUs. We can see that the ray-casting time dominates the overall time for a smaller number of GPUs, which constantly decreases with the increasing number of GPUs. The measured timing results are close to the ideal speedup time. For a high resolution image of 4096^2, our approach achieves a parallel efficiency of 85% from 4 GPUs to 64 GPUs, and a parallel efficiency of 50% from 4 GPUs to 1024 GPUs.

The compositing time is nearly constant with the increasing number of processors because of its logarithmic complexity. As shown in Figure 8, when the processor number is less than 64, as expected, the compositing time can be hidden by overlapping ray-casting and image compositing. However, when the processor number is larger than 64, the compositing time starts to dominate the overall time. Most existing parallel visualization solutions for supercomputers use CPU-based rendering algorithms, and thus the rendering time is longer than the image compositing time until a large number of processors are used. In our new GPU-based parallel visualization framework, the rendering time can be significantly reduced by leveraging GPUs available on supercomputers, while compositing is still conducted using CPUs and MPI. Thus, the curve of composting time can quickly intersect with the curve of rendering time even for a smaller number of processors. We aim to develop optimization techniques for image compositing in support of parallel GPU-based rendering in the future.

Figure 9 shows the timing results for rendering the MPAS data set that features a high variation of mesh density. Even for this data set, our approach demonstrates a similar performance trend

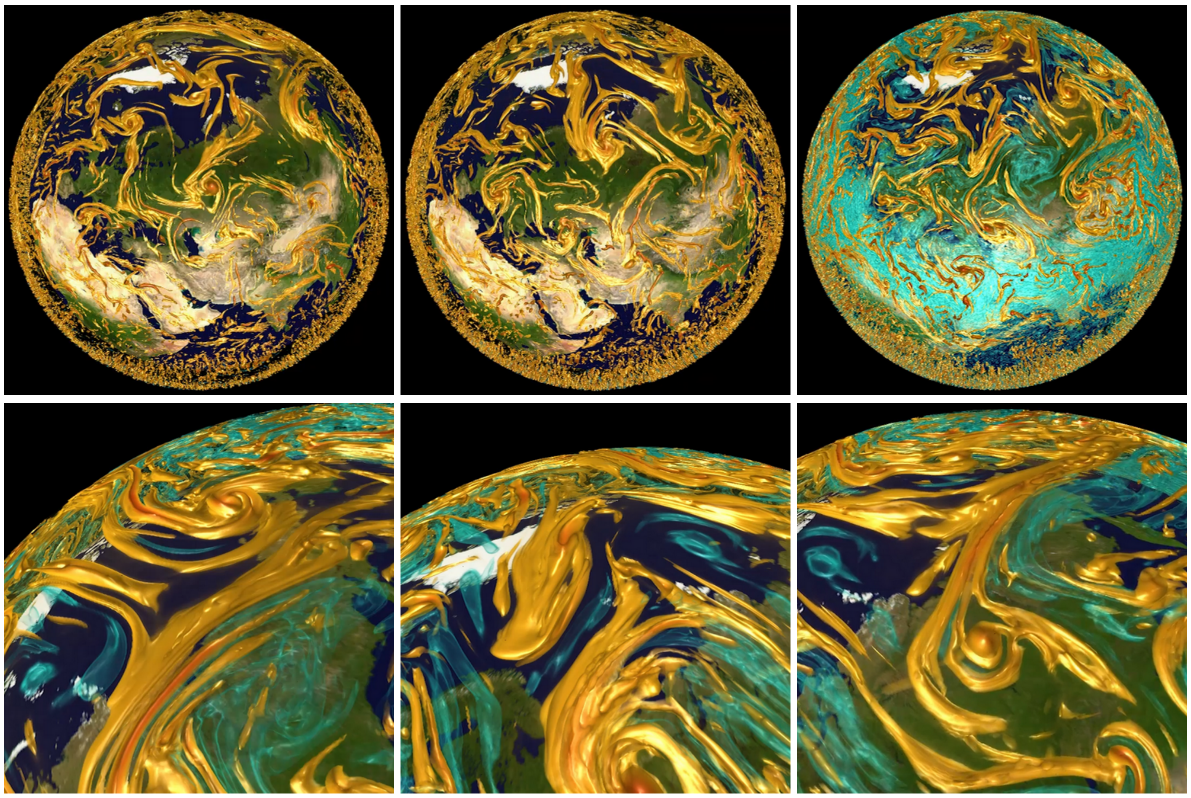


Figure 11: The visualization results of the GCRM data set. The top row of images show the whole global atmosphere vorticity variable over three time steps. The bottom row of images show the close-up views of regions of interest.

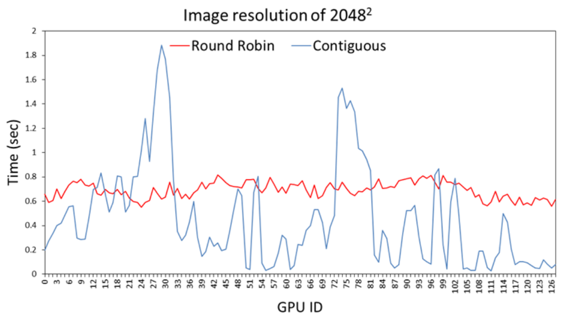


Figure 10: The ray-casting time for each GPU using the MPAS data set and different data distribution schemes. The output image resolution is 2048^2. The blue and red curves correspond to the contiguous data assignment (Figure 7(a)) and our round robin data assignment (Figure 7(b)), respectively.

as for the GCRM data. The ray-casting time is still close to the ideal speedup time. Figure 10 shows the ray-casting time for each GPU when 128 GPUs are used in rendering of the MPAS data set with an output image resolution of 2048^2. The time is averaged over the 10 viewing directions. The difference ratio, defined as $(max\_time - min\_time)/max\_time$, is 32.5% for the red curve and 98.6% for the blue curve. This clearly shows that the workloads are well-balanced among the GPUs from different view directions using our data partitioning and distribution scheme. Figures 11 and 12 show the overviews and zoom-in views of the GCRM and MPAS data sets for three selected time steps. Our high-resolution parallel visualization solution delivers high-quality results that enable scientists to interactively explore fine details of the volume data. A supplementary video demonstrating an interactive exploration of each time-varying data set using our renderer is provided at http://youtu.be/1bspVTsGSY8.

6 Conclusions

We have introduced a scalable solution for visualizing large-scale 3D geodesic grid data using massively distributed GPUs on state-of-the-art supercomputers. Based on a careful characterization of geodesic grids, we use spherical quadtrees to partition and distribute geodesic data. Our design achieves a balanced workload across processors, and makes it practical to interactively visualize large geodesic grid data. Our visualization framework directly takes the original mesh as input, and thus is ready to be integrated with simulations. In the future, we plan to experiment with other data partitioning and distribution schemes, including the ones deployed by the simulations, to enable in-situ visualization. We also would like to develop optimization methods to further reduce the image composting cost.

7 Acknowledgments

This research has been sponsored in part by the National Science Foundation through grants DRL-1323214 and IIS-1255237, and Department of Energy through grants DE-FC02-06ER25777, DE-CS0005334, and DE-FC02-12ER26072 with program managers Lucy Nowell and Ceren Susut-Bennett.

References

[1] C. Chen, J. Bin, and F. Xiao. A global multimoment constrained finite-volume scheme for advection transport on the hexagonal geodesic grid. *Monthly Weather Review*, (140):941–955, 2012.
[2] A. C. Clarke. Extra-terrestrial relays: Can rocket stations give world-wide radio coverage? *Wireless World*, page 306, October 1945.
[3] J. Daily, K. Schuchardt, and B. Palmer. Efficient extraction of regional subsets from massive climate datasets using parallel IO. In *American Geophysical Union, Fall Meeting 2010*, pages IN41A–1360, 2010.

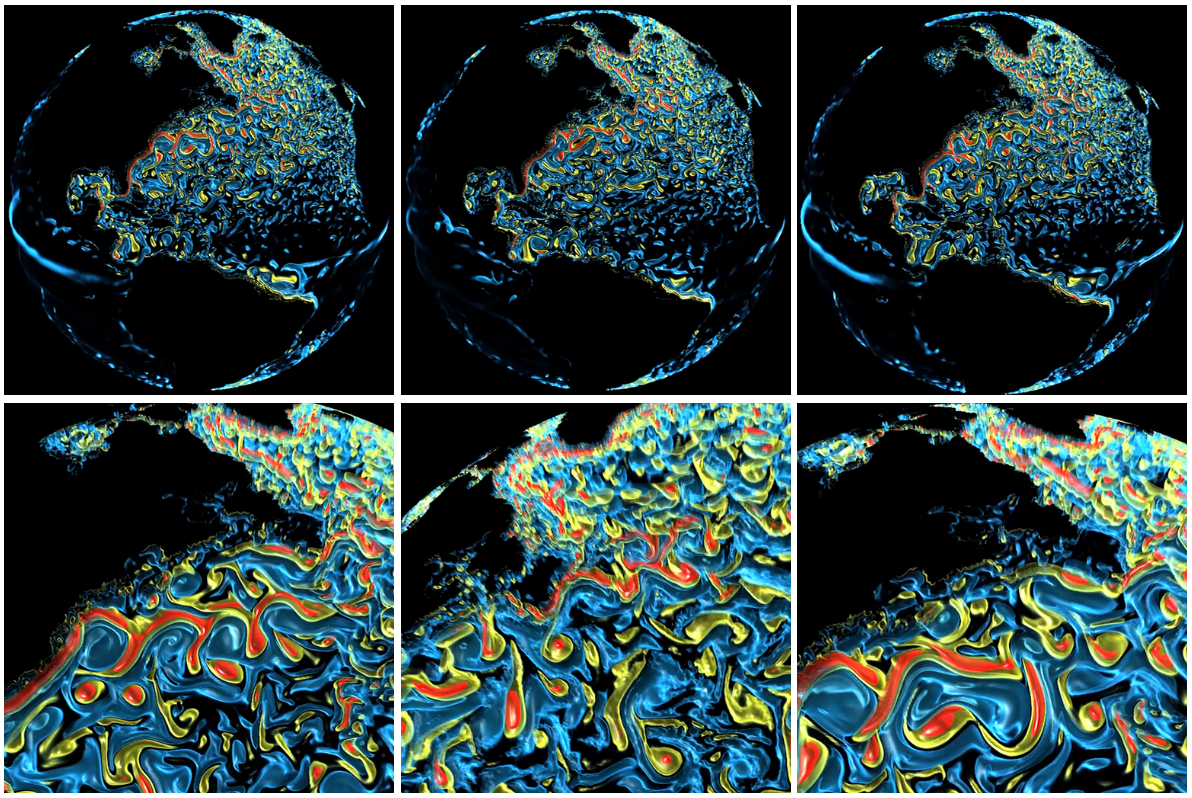


Figure 12: The visualization results of the MPAS data set. The top row of images show the whole global ocean vorticity variable over three time steps. The bottom row of images show the close-up views of regions of interest.

[4] N. Fabian, K. Moreland, D. Thompson, A. Bauer, P. Marion, B. Gevecik, M. Rasquin, and K. Jansen. The paraview coprocessing library: A scalable, general purpose in situ visualization library. In *Proc. of IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 89 –96, October 2011.

[5] I. Gargantini. An effective way to represent quadtrees. *Commun. ACM*, 25(12):905–910, Dec. 1982.

[6] M. P. Garrity. Raytracing irregular volume data. In *Proceedings of VVS*, pages 35–40, 1990.

[7] P. Lauritzen, C. Jablonowski, M. Taylor, and R. Nair. *Numerical Techniques for Global Atmospheric Models*. Springer, 2011.

[8] N. Leaf, V. Vishwanath, J. Insley, M. Hereld, M. Papka, and K.-L. Ma. Efficient parallel volume rendering of large-scale adaptive mesh refinement data. In *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization*, 2013.

[9] F. D. Libera and F. Gosen. Using b-trees to solve geographic range queries. *The Computer Journal*, 29(2):176–180, 1986.

[10] K.-L. Ma and T. Crockett. A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data. In *Parallel Rendering, 1997. PRS 97. Proceedings. IEEE Symposium on*, pages 95–104, 119–20, Oct 1997.

[11] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Comput. Graph. Appl.*, 14(4):59–68, July 1994.

[12] MATLAB. MATLAB R2014a Documentation, 2014.

[13] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Comput. Graph. Appl.*, 14(4):23–32, July 1994.

[14] U. Neumann. Communication costs for parallel volume-rendering algorithms. *IEEE Comput. Graph. Appl.*, 14(4):49–58, July 1994.

[15] S. Parker, M. Parker, Y. Livnat, P.-P. Sloan, C. Hansen, and P. Shirley. Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):238–250, July 1999.

[16] T. Peterka, D. Goodell, R. Ross, H.-W. Shen, and R. Thakur. A configurable algorithm for parallel image-compositing applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 4:1–4:10, 2009.

[17] D. A. Randall, T. D. Ringler, R. P. Heikes, P. Jones, and J. Baumgardner. Climate modeling with spherical geodesic grids. *Computing in Science & Engineering*, 4(5):32–41, 2002.

[18] R. Sadourny, A. Arakawa, and Y. Mlntz. Integration of the nondivergent barotropic vorticity equation with an icosahedral-hexagonal grid for the sphere. *Monthly Weather Review*, (96):351–356, 1968.

[19] J. Shalf, S. Dosanjh, and J. Morrison. Exascale computing technology challenges. In *Proceedings of the 9th international conference on High performance computing for computational science*, VECPAR'10, pages 1–25, 2011.

[20] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. In *Proceedings of VVS*, pages 63–70, 1990.

[21] W. Tobler and Z.-t. Chen. A quadtree for global information storage. *Geographical Analysis*, 18(4):360–371, 1986.

[22] P. Wang, Z. Cheng, R. Martin, H. Liu, X. Cai, and S. Li. Numa-aware image compositing on multi-gpu platform. *The Visual Computer*, 29(6-8):639–649, 2013.

[23] Z. Wartell, W. Ribarsky, and L. Hodges. Efficient ray intersection for visualization and navigation of global terrain using spheroidal height-augmented quadtrees. In *Data Visualization '99*, Eurographics, pages 213–223, 1999.

[24] D. L. Williamson. Integration of the barotropic vorticity equation on a spherical geodesic grid. *Tellus*, (20):642–653, 1968.

[25] J. Xie, H. Yu, and K.-L. Ma. Interactive ray casting of geodesic grids. In *Proceedings of the 15th Eurographics Conference on Visualization*, EuroVis '13, pages 481–490, 2013.

[26] H. Yu, C. Wang, R. W. Grout, J. H. Chen, and K.-L. Ma. In situ visualization for large-scale combustion simulations. *IEEE Computer Graphics and Applications*, 30(3):45–57, 2010.

[27] H. Yu, C. Wang, and K.-L. Ma. Massively parallel volume rendering using 2-3 swap image compositing. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 48:1–48:11, 2008.