TOD-Tree: Task-Overlapped Direct send Tree Image Compositing for Hybrid MPI Parallelism and GPUs

A.V.Pascal Grosset, Manasa Prasad, Cameron Christensen *Student Member, IEEE,* Aaron Knoll *Member, IEEE,* and Charles Hansen *Fellow, IEEE*

Abstract—Modern supercomputers have thousands of nodes, each with CPUs and/or GPUs capable of several teraflops. However, the network connecting these nodes is relatively slow, on the order of gigabits per second. For time-critical workloads such as interactive visualization, the bottleneck is no longer computation but communication. In this paper, we present an image compositing algorithm that works on both CPU-only and GPU-accelerated supercomputers and focuses on communication avoidance and overlapping communication with computation at the expense of evenly balancing the workload. The algorithm has three stages: a parallel direct send stage, followed by a tree compositing stage and a gather stage. We compare our algorithm with radix-k and binary-swap from the IceT library in a hybrid OpenMP/MPI setting on the Stampede and Edison supercomputers, show strong scaling results and explain how we generally achieve better performance than these two algorithms. We developed a GPU-based image compositing algorithm where we use CUDA kernels for computation and GPU Direct RDMA for inter-node GPU communication. We tested the algorithm on the Piz Daint GPU-accelerated supercomputer and show that we achieve performance on par with CPUs. Lastly, we introduce a workflow in which both rendering and compositing are done on the GPU.

Index Terms—Distributed volume rendering, image compositing, parallel processing.

1 Introduction

As the power of supercomputers increases, scientists are running more and more complex simulations that use thousands of nodes and generate huge amounts of data. Moving these datasets is often inconvenient due to their sheer size, and so analysis and visualization are increasingly done on the same High Performance Computing (HPC) system where the data was generated. Distributed volume rendering on HPC systems usually involves three stages: loading, rendering and compositing. In the loading stage, the data is divided among the nodes, using, for example, a k-d tree [1] or the domain decomposition of the simulation. Each node renders the data it has to an image, using an algorithm such as direct volume rendering; finally in the compositing stage, the nodes exchange and blend the images they have to create one image representing the whole dataset. The I/O stage is usually very expensive when visualizing data [2] and is a big problem in its own right. This is beyond the scope of this paper. Here, our focus is on rendering and especially compositing. When few nodes are being used, the rendering stage is usually slower than compositing but as the number of nodes increases, compositing becomes the dominant cost. Thus, fast rendering requires a fast compositing algorithm. This is especially important for in-situ visualization where supercomputing time is a precious resource [3] and visualization should add minimal overhead. In this paper, our focus is on the compositing stage of distributed volume rendering on HPC systems.

Increases in computing power are no longer being achieved through faster clock speed but rather through extensive parallelism. Nodes in supercomputers now have CPUs with at least 8 cores, many-core co-processors with 60 cores and GPUs with hundreds of cores (blocks). These nodes have peak performances on the order of several hundreds of gigaflops or even teraflops. Also, although cores on a chip can share data very quickly using threads and shared memory, inter-node communication through the network is much slower. Minimizing inter-node communication is one of the major challenges of exascale computing [4].

To adapt to this change in architecture, algorithms are being developed that minimize inter-node communication. Previously it was common to have one MPI process per core, but now the trend is to have one MPI process per node and use threads and shared memory inside a node. Work by Mallon et al. [5] and Rabenseifner et al. [6], summarized by Howison et al. [7], [8], shows that the hybrid MPI model results in fewer messages between nodes and less memory overhead, eventually outperforming MPI-only at every concurrency level. With multi-core / many-core CPUs, Howison et al. found that using threads and shared memory inside a node and MPI for inter-node communication is much more efficient than using MPI for both inter-node and intra-node for visualization. However, the two most commonly used compositing algorithms, binary-swap [9] and radix-k [10], focus on load balancing and not on communication avoidance. When these algorithms were developed, load balancing was of prime importance. But with modern systems, it is more important to minimize

- *A.V.P Grosset, C Christensen, A Knoll and C Hansen are with the Scientific Computing and Imaging Institute at the University of Utah, Salt Lake City, UT, 84112. Email:{pgrosset, cam, knolla}@sci.utah.edu, hansen@cs.utah.edu.*
- *Manasa Prasad is with Google, Mountain View, California, CA, 94043. Email: pbmanasa@gmail.com*

Manuscript received 25 November, 2015; revised X February, 2016.

communication at the expense of equally balancing the workload, given the massive amount of computing power that a node has and the comparatively low bandwidth between nodes. Radix-k and binary-swap can be split into two stages: compositing and gathering. Moreland et al. [11] show that the compositing time decreases as the number of processes grows, but the gathering time increases even more, therefore the total overall time increases.

It is hard to predict what the architecture of future supercomputers will be: it could be many-core CPU-only systems or GPU-accelerated systems. Having an algorithm that can work on both CPU-only and GPU-accelerated supercomputers is thus very important. Currently (October 2015), 2 of the top 10 of the Top 500 supercomputers [12] are equipped with Nvidia GPUs. GPUs have been so successful for General Purpose computing on GPU (GPGPU) that although they were initially developed for accelerating graphics, they are now mostly used in supercomputers for computing rather than for graphics. Until recently, GPUs could be used only for compute or graphics, but not both. However, Nvidia Tesla class GPU K20 and above can run both graphics and compute at the same time. Moreover, whereas inter-node communication between GPUs previously had to go through the CPU, with the introduction of GPU Direct Remote Direct Memory Access (RDMA), GPUs can communicate directly over a network with minimal latency. These two changes allow us to do both rendering and compositing on the GPU since GPUs are at least twice as fast as CPUs for raycast rendering [13].

In this paper, we introduce the Task Overlapped Direct send Tree (TOD-Tree) image compositing algorithm, which minimizes communication and focuses on overlapping communication with computation. This paper is an extension of our previous work [14] where we compared the performance of this algorithm with radix-k and binary-swap on an artificial and combustion dataset and showed that we generally achieve better performance than these two algorithms in a Hybrid OpenMP/MPI parallelism setting on the Stampede and Edison supercomputers. Here, we extend this algorithm to GPU-accelerated supercomputers.

The new contributions are:

- development of a multi-GPU compositing algorithm based on TOD-Tree that takes advantage of modern GPU capabilities.
- scaling to 4096 GPUs on Piz Daint, a GPU-accelerated supercomputer.
- a workflow that allows seamless transfer, with minimal latency, of renderings from an OpenGL context to a CUDA context and uses GPU Direct RDMA for compositing.

Whereas volume rendering is often done on GPUs, compositing is usually done on the CPU [15], [16]. In this work, we do both on the GPU. The only image compositing algorithm that we have found for GPUs is parallel direct send in the vl3 system [17], which has been scaled to 128 GPUs on the Tukey computer cluster at Argonne. In this paper, we scaled to 4096 GPUs on the GPU-accelerated supercomputer Piz Daint. As far as we know, this is the most an image compositing algorithm has been scaled using GPUs. We compared the performance of TOD-Tree scaled to 4096 nodes on two CRAY XC30 systems: Edison, a CPU-only supercomputer; and Piz Daint, a GPU-enhanced supercomputer. We show that GPU compositing achieves performance on par with CPU compositing for 2K x 2K and 4K x 4K images, and even better performance for 8K x 8K images.

Most visualization software uses OpenGL and shaders to do volume rendering on GPU. However, GPU Direct RDMA, which allows GPUs to talk across a network, does not work in OpenGL; it only works using CUDA. So after rendering in OpenGL, we need to switch over to CUDA for image compositing. Transferring data from OpenGL to CUDA can be easily done using the CUDA OpenGL Interoperability runtime. The usual render target for OpenGL offscreen rendering are textures, which are mapped to CUDA arrays using the CUDA OpenGL Interoperability. CUDA arrays reside in texture memory but GPU Direct RDMA does not work with texture memory, only device memory. Moving data from texture memory to device memory can be quite expensive, so we instead render to an OpenGL buffer object that can be mapped to device memory. The workflow we introduce shows how to do rendering and image compositing using the GPU and what is required to modify existing systems to do all the visualization on the GPU. This could be very useful for in-situ visualization where simulation and visualization can proceed in parallel on the CPU and GPU, respectively. As far as we know, this is the fist time this workflow has been used.

The paper is organized as follows: in Section 2, the previous work section, different compositing algorithms that are commonly used and GPU volume rendering systems are described. In Section 3, the TOD-Tree algorithm is presented, its theoretical cost is described and we present a workflow for visualization of GPUs that do not involve CPU. Section 4 shows the results of strong scaling for an artificial dataset and a combustion simulation dataset, and the results obtained are explained. Section 5 discusses the conclusion and future work.

2 Previous Work

Volume rendering is now commonly used for visualization. Many supercomputers such as Piz Daint and Stampede allow their users to use software such as Paraview [18] and VisIt [19] for distributed volume rendering. There are three main approaches to parallel rendering: sort-first, sort-middle and sort-last [20]. Sort-last is the most commonly used approach. In sort-last, each process loads part of the data and renders it to an image. A depth value is associated to each image. In the compositing stage, the processes exchange and blend images according to the depth information to create a final representation of the dataset. There is no communication in the loading and rendering stages, but the compositing stage is very communication intensive, and therefore many different algorithms have been developed to address compositing.

2.1 Image compositing

The most commonly used compositing algorithms are direct send, binary-swap and radix-k.

Direct send is the oldest of the three and can refer to serial direct send or parallel direct send. In serial direct send, all the processes send their data to the display process, which blends them in a front-to-back or back-to-front order. There is a massive load imbalance in serial direct send that makes it quite slow. Parallel direct send [21], [22] is a two-stage process. In the first stage, each process is made responsible for a different section of the final image. Processes then send any sections for which they have data, but for which they are not responsible, to their rightful owners, and receive sections for which they are responsible. These sections are then blended in the correct order. During the gather stage, all processes send their authoritative section to the display process, which puts them in the right position in the final image. The SLIC compositing algorithm by Stompel et al. [23] is essentially an optimized direct send. Pixels from the rendered image from each process are classified to determine if they can be sent directly to the display process (non-overlapping pixels) or will require compositing. Then processes are assigned sections of the final image for which they have data, and pixel exchanges are done through direct send.

Binary-swap, introduced by Ma et al. [9], is an improvement on binary tree compositing techniques. In binary tree compositing, processes are paired and arranged in a tree structure. The number of stages required for compositing corresponds to the depth of the tree. At each stage, a leaf sends its data to the other leaf in its pair, which means that half of the processes become inactive at each stage, thereby creating load imbalance. In binary-swap, all processes remain active until the end. Initially, each process is responsible for the whole image. The processes are sorted by depth and arranged in pairs, and at each stage, each leaf becomes responsible for half of the section for which it was initially responsible. They exchange the sections they do not need and blend the section they receive, which continues until each process p has $1/p$ of the whole image. The display process then gathers sections from each process to create the final image. Yu et al. [24] extended binary-swap to deal with non-power of 2 processes.

Radix-k was introduced by Peterka et al. [10]. Here, the number of processes p is factored in r factors so that k is a vector where $k = [k_1, k_2, ..., k_r]$. The processes are arranged into groups of size k_i and exchange information using parallel direct send. At the end of a round, each process is authoritative on a different section of the final image for its group. Processes with the same authoritative section are arranged in groups of size k_{i+1} and exchange information. This goes on for r rounds until each process is the only one authoritative on a section of the image. The display process then gathers data from all the other processes in the gather stage. If the vector k has only one value equal to p, radix-k behaves like direct send. If each value of k is equal to 2, it behaves likes binary-swap.

Radix-k, binary-swap and direct send are all available in the IceT package [25], which also adds several optimizations such as telescoping and compression, described in [11].

Also, recognizing that communication is the main bottleneck in image compositing, Pugmire at al. [26] used a Network Processing Unit (NPU) to speed up the communication while Cavin et al. [27] used shift permutation to get the maximum cross bisectional bandwidth from InfiniBand Fat-Trees to speed up communication. These improvements tie compositing algorithms to specific hardware network infrastructure, rather than providing a more general software solution.

Howison et al. [7], [8] compared volume rendering using only MPI versus using MPI and threads, which can be seen as a predecessor to this work. They clearly establishe that using MPI and threads, is the way forward as it minimizes exchange of messages and results in faster volume rendering. However, for compositing, Howison et al. used only MPI_Alltoallv but do mention in their future work the need for a better compositing algorithm. Our work presents a new compositing algorithm for hybrid OpenMP/MPI.

2.2 Rendering and compositing on the GPU

Many systems, such as Chromium [28] and Equalizer [29], have been developed for parallel rendering on GPUs. Direct volume rendering using either a slicing [30] or raycasting [31] approach has been done on the GPU. Muller et al. [32] and Fogal et al. [33] have developed distributed memory volume renderers for GPU that use shaders and OpenGL. For compositing, Fogal et al. used a tree-based compositing from IceT and Muller et al. used direct send. In both cases, compositing involved copying data out of the GPU before inter-node communication with MPI. Recently, Xie et al. [15] used up to 1024 GPUs for rendering on the Titan supercomputer, a Cray XK7 system, at Oak Ridge National Laboratory, but they used the CPU for image compositing. The only instance we found where it was explicitly mentioned that compositing was done on GPUs is the vl3 system by Rizzi et al. [17]. They compared the performance of serial and parallel direct send scaling up to 128 Nvidia Tesla M2070 GPUs but do not mention the use of GPU Direct RDMA for image compositing.

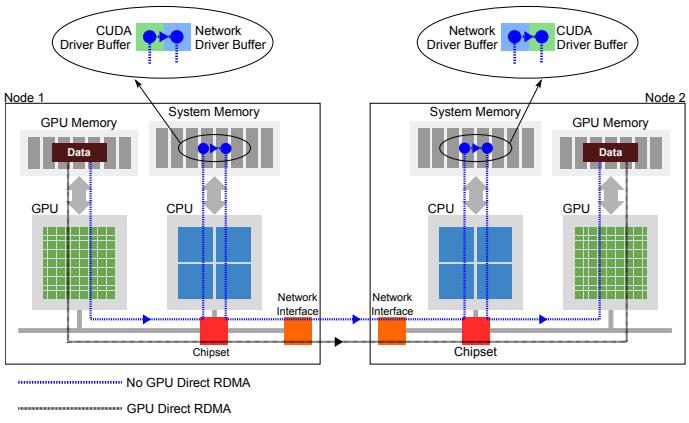

Fig. 1: Inter-node GPU communication with and without GPU Direct RDMA.

Currently, the only way for GPUs to communicate directly across a network is through CUDA. In 2011, Wang et al. [34] proposed an MPI design that integrates CUDA data movement with MPI; they achieve a 45% improvement in one-way latency. GPU Direct RDMA [35] was then introduced in CUDA 5.0. Potluri et al. [36] mentioned a MPI 69% and 32% for 4 Byte and 128 KB messages, respectively, for

Send/MPI Recv using GPU Direct RDMA on infiniband systems. Now, GPU Direct RDMA is available in MVAPICH2, OpenMPI and CRAYMPI. In the worst case, without GPU Direct RDMA, 5 copies are needed, as shown in figure 1, to transfer data between GPUs found in different nodes. The data is first copied from the GPU's memory to the CUDA driver buffer's memory found in main memory. Another copy transfers the data to the network driver buffer, also in main memory. The next copy takes the data across the network to the network driver buffer in the destination node. There, another copy is needed to transfer the data to the CUDA driver buffer and, finally, a last copy sends the data to the GPU's memory [37]. However, using GPU Direct RDMA, only 1 copy is required to transfer data between GPUs across nodes.

Since rendering is mostly done in OpenGL rather than CUDA, the CUDA OpenGL interoperability, provided as part of the CUDA Runtime API, can be used as a bridge between CUDA and OpenGL. Initially, it was not possible on Tesla class Nvidia GPUs used in supercomputers to run both CUDA and OpenGL at the same time but this capability is now available in the the Nvidia K20m, K20X, K40 and K80 GPUs [38]. The only additional requirement for running OpenGL is to have an X Server, which is needed to create an OpenGL context. Klein and Stone [39] describe how to get OpenGL working on a Cray XK7 accelerator. Also, some GPU-accelerated supercomputers, such as the Piz Daint supercomputer in Switzerland, have an X Server module that can be loaded as needed.

3 Methodology

As mentioned before, distributed volume rendering has three stages: loading, rendering and compositing. At the end of the rendering stage, each node has an image with an associated depth. To ensure correct visualization, the images need to be blended in the correct depth order. Therefore, image and depth are provided to TOD-Tree and other compositing algorithms such as direct send, binary swap and radix-k.

3.1 Algorithm

The TOD-Tree (Task-Overlapped Direct send Tree) algorithm has three stages. The first stage is a grouped direct send. It is followed by a k-ary tree compositing stage. The display process then gathers data in the display stage. In all stages, asynchronous communication is used to overlap communication and computation. We first describe the algorithm conceptually.

Each node has a list of nodes sorted from smallest to largest depth. In the first stage, the nodes are arranged into groups of size r, which we will call a locality, based on their position in the depth-ordered list. Each node in a locality will be responsible for a region equivalent to $1/r$ of the final image. If r is equal to 4, there are 4 nodes in a locality, as shown in stage 1 of figure 2, and each is responsible for a quarter of the final image. The nodes in each locality exchange sections of the image in a direct send fashion so that at the end of stage 1, each node is authoritative on a different $1/r$ of the final image. The colors red, blue, yellow and green in figure 2 represent the first, second, third and fourth quarters of the final image on which each node is authoritative on. Also in figure 2, there are 25 processes initially. In this case, the last locality will have 5 instead of 4 nodes, and the last node, colored orange in the figure, will send its regions to the first r node in its locality but will not receive any data. In the second stage, the aim is to have only one node that is authoritative on a specific $1/r$ region of the final image. The nodes having the same region at the end of stage 1 are arranged in groups of size k based on their depth information. Each node in a group sends its data to the first node in its group, which blends the pixels, similar to k-ary tree compositing [40], [24], [9]. This stage can have multiple rounds. For example, in stage 2 of figure 2, 6 processes have the same quarter of the image, therefore two rounds are required until only one node is authoritative on a quarter of the image. Finally, these nodes blend their data with the background and send it to the display node, which assembles the final image, stage 3 in figure 2.

We now describe in detail how we implement each stage of the algorithm, paying attention to the order of operation to maximize overlapping of communication with computation.

Algorithm 1: Stage 1 - Direct Send

```
Determine the nodes in its locality
Determine the region of the image the node owns
Create a buffer for receiving images
Advertise the receive buffer using async MPI Recv
if node is in first half of locality then
    Send front to back using async MPI Send
else
    Send back to front using async MPI Send
Create a new image buffer
Initialize the buffer to 0
if node is in first half of region then
    Wait for images to come in front-to-back order
    Blend front to back
else
    Wait for images to come in back-to-front order
    Blend back to front
Deallocate receive buffer
```

Algorithm 1 shows the setup for the direct send stage. There are a few design decisions to make for this part. Asynchronous MPI send and receive allows overlap of communication and computation. Posting the MPI receive before the send lets messages be received directly in the target buffer, instead of being copied to a temporary buffer and then copied to the target buffer. To minimize link contention, not all nodes try to send to one node. Depending on where they are in the locality, the sending order is different. The buffer used as the sending buffer is the original image rendered in that node. To minimize memory use, there is only one blending buffer and so the data must be available in the correct order for blending to start. The alternative would be to blend on the fly as images are received, but this requires creating and initializing many new buffers, which can have a very high memory cost when the image is large. The tests we carried out revealed that the gains in performance were not significant enough to outweigh the cost of allocating

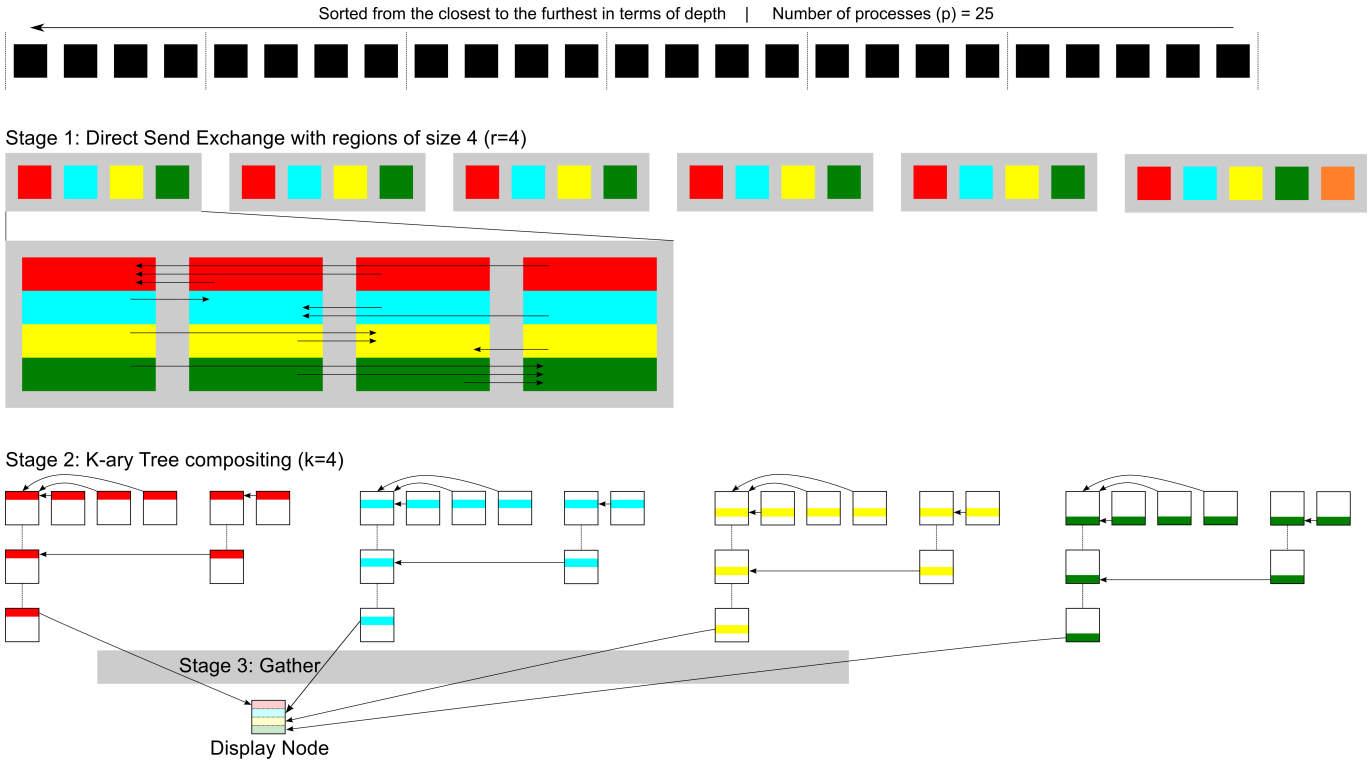

Fig. 2: The three stages of the compositing algorithm with r=4, k=4, and the number of nodes p=25. Red, blue, yellow and green represent the first, second, third and fourth quarters of the image.

that much memory. The blending buffer also needs to be initialized to 0 for blending, which is a somewhat slow operation. To amortize this cost, this is done after the MPI operations have been initialized so that receiving images and initialization can proceed in parallel.

Algorithm 2: Stage 2 - Tree Region

```
Determine if the node will be sending or receiving
Create a buffer for receiving images
for each round do
    if sending then
        Send data to destination node
    else
        Advertise receive buffer using async MPI Recv
        if last round then
            Create opaque image for blending received images
            Create alpha buffer for blending transparency
            Blend current image with the background
            Receive images
            Blend in the opaque buffer
        else
            Receive images
            Blend in image buffer created in stage 1
Deallocate image buffer created in stage 1
Deallocate receive buffer
```

The second stage is a k-ary tree compositing, shown in algorithm 2. Again, the receive buffer is advertised early to maximize efficiency. Another optimization that has been added is to blend with the background color in the last round while waiting for data to be received, thereby overlapping communication and computation. Also, alpha is needed when compositing but not in the final image. Therefore, while blending in the last round, the alpha channel is separated from the rest of the image. It is still used for blending in that stage but is not sent in the gather stage, which allows the last send to be smaller and makes the gather faster.

Algorithm 3: Stage 3 - Gather

```
Create empty final image
if Node has data then
    Send opaque image to display node
else
    if display node then
        Advertise final image as receive buffer
Deallocate send buffer from stage 1
```

Finally, the last stage of the algorithm is a simple gather from the nodes that still have data. Since the images have already been blended with the background in the previous stage, no computation is needed in this stage. The display node creates the final image, which is also the receive buffer, and indicates where data from each of the final senders should be placed. As soon as all the images are in, compositing is done. Also, at the end of this stage, the send buffer used in stage 1 is deallocated. Deallocation in earlier stages of the algorithm often involves waiting for images to

be sent, but in stage 3, the images should have already been sent and so no waiting is required. This has been confirmed in some tests we carried out.

The two parameters to choose for the algorithm are the number of regions r and a value for k. r determines the number of regions into which an image is split for load balancing purposes. As the number of nodes increases, increasing the value of r results in better performance. k is used to control how many rounds the tree compositing stage has. It is usually best to keep the number of rounds low.

3.2 Workflow for rendering on the GPU

OpenGL and shaders are the most obvious choice for doing volume rendering on the GPU, but the only technology that allows GPUs to talk across a network is GPU Direct RDMA, which is available only in CUDA. In this section, we describe the workflow that allows the seamless transfer of data rendered from the OpenGL graphics pipeline to CUDA using the CUDA OpenGL interoperability runtime.

All OpenGL programs need an OpenGL context. To create a context on Linux, the operating systems that most HPC systems use, an X server is required. The X server is a program that sits on top of the driver and handles input and output from an application. To create a context, the Xlib library is used to connect to the X server, and GLX is then used to create a context. On desktop systems, an X server is usually started by default, but on compute nodes of HPC systems, the X server might have to be explicitly started using $\#SBATCH --constraint = startx$ in the job submission script to the job scheduling system. In the future, once most GPU drivers in supercomputers have support for EGL [41], we should not have to initialize an X server anymore to create an OpenGL context. Figure 3 shows the interaction that goes on with the GPU, driver, X server, libraries and application.

Compute nodes in supercomputers are rarely connected to displays. OpenGL rendering is, therefore, usually offscreen targeted to a framebuffer object or renderbuffer object, both of which are usually mapped to texture memory in OpenGL. When using CUDA OpenGL interoperability, they will be mapped to texture memory in CUDA, but GPU Direct RDMA does not work from texture memory. There are two ways to map data from texture memory to device memory in CUDA. It can be copied to device memory using $cudaMemcpyFromArray$ and $cudaMemcpyDeviceToDevice$ or through a CUDA kernel. However, in some tests that we ran, we found both approaches to be slow for large textures. Using $cudaMemcpyFromArray$, it took about 5 ms for a 4,096 x 4,096 RGBA32F image and about 21 ms for 8,192 x 8,192 RGBA32F image. Therefore, instead of rendering to a framebuffer object, we render to an OpenGL Buffer Object, more specifically to a $GL\_TEXTURE\_BUFFER$, which is mapped to device memory when using CUDA OpenGL interoperability.

A $GL\_TEXTURE\_BUFFER$ can store up to 134,217,728 million pixels (a maximum image size of 8,192 x 16,384 pixels) and behaves like a regular OpenGL texture but is only one-dimensional. To store the output of a fragment shader to it, we need to map the (x,y) screen coordinates to a one-dimensional position in GLSL as follows:

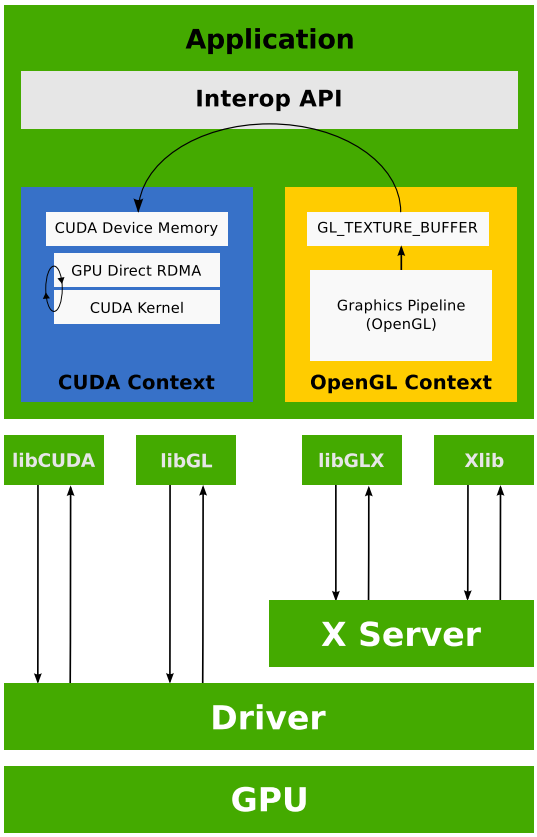

Fig. 3: OpenGL and CUDA interaction with the GPU.

Listing 1: Computing fragment location

```
int index;
index = (int(floor(gl_FragCoord.y)) - minY)*
 width +(int(floor(gl_FragCoord.x)) - minX));
```

where width is the width of the screen, minX and minY are the minimum x and y coordinate, and gl_FragCoord is an OpenGL variable that stores the coordinates of a fragment in screen space.

The steps to render to a $GL\_TEXTURE\_BUFFER$ instead of a framebuffer in OpenGL are:

1) initialize a $GL\_TEXTURE\_BUFFER$ and bind it to a texture
2) pass the texture, its width and height and minimum x and y values to the shader
3) to receive the uniform in the shader:
   $layout(rgba32f, binding = X)\ coherent\ uniform\ imageBuffer\ imgOut;$
   (where X is the texture number and imgOut is the name of the texture)
4) compute the index of where to store the fragment as shown in listing 1
5) use imageStore to store the fragment

Once rendering is done, the texture buffer object can be mapped to CUDA device memory using the $cudaGraphicsGLRegisterBuffer$ function. Then CUDA kernels are used for blending and GPU Direct RDMA for communication. Rendering to an OpenGL buffer object instead of the usual framebuffer is key in the workflow,

shown in figure 4, since it allows latency to be kept to a minimum by not having to copy any data. Also, changing the rendering target to a texture buffer object in an existing program should be quite straightforward.

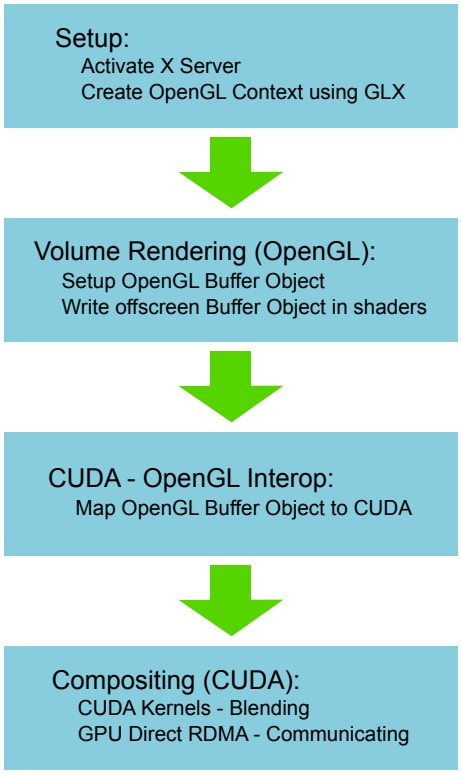

Fig. 4: Workflow for GPU rendering.

3.3 Implementation

The same compositing algorithm, presented in the algorithm section, is used for both CPU and GPU. The only changes needed are blending and memory allocation. For blending, CUDA kernels are used on the GPU instead of OpenMP with vectorization on the CPU, and memory allocations and deallocations are through $cudaMalloc$ and $cudaFree$. No change is needed to use GPU Direct RDMA in the program; the same MPI calls are made but the buffers used are in CUDA device memory. In the job script submitted to job scheduling system, $export\ MPICH\_RDMA\_ENABLED\_CUDA = 1$ is needed to activate GPU Direct RDMA, which is verified in the program by checking the environment variable using $getenv(\text{“}MPICH\_RDMA\_ENABLED\_CUDA\text{”})$.

For the rendering stage, on the GPU, OpenGL 4.4 and GLSL shaders were used to implement ray casting volume rendering. The same algorithm was implemented in C++ for the CPU.

3.4 Theoretical Cost

We now analyze the theoretical cost of the algorithm using the cost model of Chan et al. [42], which has been used by Peterka et al. [10] and Cavin et al. [27]. Let the number of pixels in the final image be n, the number of processes be p, the time taken for blending one pixel be γ, the latency for one transfer be α and the time for transferring one pixel be β. Stage 1 is essentially several direct sends. The number of sends in a group of size r per process is $(r-1)$ and the number of compositings is $r-1$. Since each of the r groups will do the same operation in parallel, the cost for stage 1 is: $(r-1)[(\alpha+\frac{n}{r}\beta)+\frac{n}{r}\gamma]$

The second stage is a k-ary tree compositing. r tree compositings are taking place in parallel. Each tree has p/r processes to composite. The number of rounds is $log_k(p/r)$. For each round, there are at most $k-1$ sends. The cost for the k-ary compositing is: $log_k\frac{p}{r}[(k-1)[(\alpha+\frac{n}{r}\beta)+\frac{n}{r}\gamma]]$

The cost for the final gather stage is: $r(\alpha+\frac{n}{r}\beta)$.

The final total cost would thus be:
$(2r+(k-1)log_k\frac{p}{r}-1)(\alpha+\frac{n}{r}\beta)+(r+(k-1)log_k\frac{p}{r}-1)\frac{n}{r}\gamma$

The cost for radix-k, binary-swap and direct send is available in the work of Cavin et al. [27] and Peterka et al. [10].

These equations are useful but fail to capture the overlap of communication and computation. It is hard to predict how much overlap there will be as communication depends on the congestion in the network, but from empirical observations, we saw that the equation acts as an upper bound for the time that the algorithm will take. For example, the total time taken for 64 nodes on Edison was 0.012s for a 2048x2048 image (64MB). We now calculate the time using the equation and performance values for Edison on the NERSC website [43]: α is at least $0.25x10^{-6}s$ and the network bandwidth is about 8GB/s, so for one pixel (4 channels each with a floating point of size 4 bytes) $\beta = 1.86x10^{-9}s$. The peak performance is 460.8 Gflops/node, so $\gamma = 8.1x10^{-12}s$. The theoretical time should be around 0.015s. The model effectively gives a maximum upper bound for the operation, but more importantly this calculation shows how much time we are saving by overlapping communication with computation. In the tests we carried out, we never managed to get 8GB/s bandwidth; we always got less than 8GB/s, and yet the theoretical value is still greater than the actual value we are measuring.

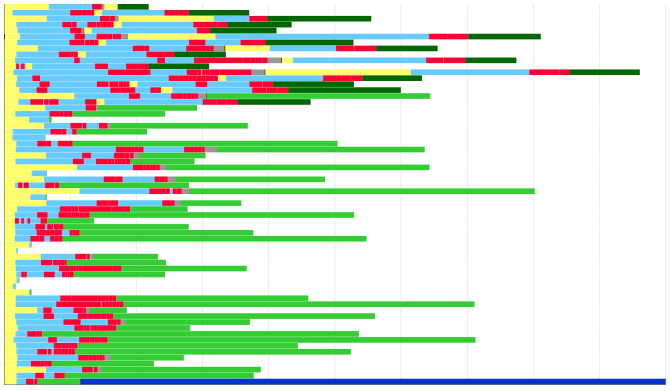

Fig. 5: Profile for 64 nodes for 2048x2048 (64MB) image on Edison at NERSC with r=16, k=8. Red: compositing, green: sending, light blue: receiving, dark blue: receiving on the display process. Total time: 0.012s.

Figure 5 shows the profile for the algorithm using an internally developed profiling tool. All the processes start with setting up buffers and advertising their receive buffer, which

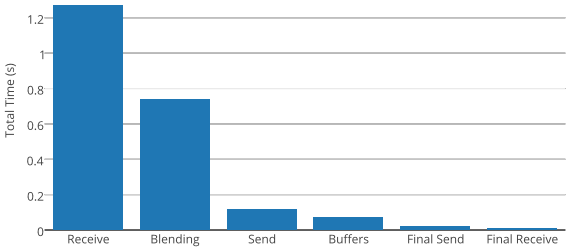

Fig. 6: Breakdown of different tasks in the algorithm.

is shown colored yellow in the diagram. This is followed by a receive/waiting to receive section, colored blue, and blending section, colored red. All receive communication is through asynchronous MPI receive whereas the sends for stage 1 are asynchronous and the rest are blocking sends. The dark green represents the final send to the display node, and the dark blue indicates the final receive on the display node. As can be clearly seen, most of the time is being spent communicating or waiting for data to be received from other nodes. A breakdown of the total time spent by 64 nodes on Edison is shown in figure 6.

As previously mentioned, the most time-consuming operations are send and receive, which is one of the reasons why load balancing is not as important anymore, and using tree style compositing is not detrimental to our algorithm.

4 Testing and Results

Most supercomputers have CPUs with many cores, and some are also enhanced by coprocessors such as Nvidia Tesla GPUs, which have thousands of cores. In this paper, we run our algorithm on both types of systems. On CPU many-core architectures, we have compared our algorithm against radix-k and binary-swap from the IceT library [11]. We are using the latest version of the IceT library, from the IceT git repository (http://public.kitware.com/IceT.git), as it has a new function icetCompositeImage, which, compared to icetDrawFrame, takes in images directly and is thus faster when provided with pre-rendered images. This function should be available in future releases of IceT. Since IceT was not built to run on GPU, we could not extend our performance comparison directly on Piz Daint. Instead, we compared the performance of TOD-Tree between CPU and GPU compositing on Edison and Piz Daint, since both are CRAY XC30 systems.

The three systems used for testing are the Stampede supercomputer at TACC, the Edison supercomputer at NERSC and the Piz Daint supercomputer at CSCS. Stampede uses the Infiniband FDR network and has 6,400 compute nodes, which are stored in 160 racks. Each compute node has an Intel SandyBridge processor, which has 16 cores per node (two sockets and one Intel Xeon E5-2680 per socket) with a peak performance of 346 GFLOPS. Each node also has an Intel Xeon Phi SE10P. The peak performance of Stampede is 8.5 PFLOPS [44] [12]. Since IceT has not been built to take advantage of threads, we did not build with OpenMP on Stampede. Both IceT and our algorithm were compiled with g++ and O3 optimization. Edison is a Cray X30 supercomputer that uses the dragonfly topology for its Aries interconnect network. The 5,576 nodes are arranged into 30 cabinets. Each node is an Intel IvyBridge processor with 12 cores (Intel Xeon E5-2695v2) and has a peak performance of 460.8 GFLOPS/node. The peak performance of Edison is 2.57 PFLOPS [43] [12]. To fully utilize a CPU and be as close as possible to its peak performance, both threads and vectorization should be used. Both SandyBridge and IvyBridge processors have 256 bit wide registers that can hold up to eight 32-bit floating point values; only when doing 8 floating point operations on all cores can we attain peak performance on one node. Crucially, IvyBridge processors offer the vector gather operation, which fetches data from memory and packs them directly into SIMD lanes. With newer compilers, this can improve performance dramatically. On Edison, we fully exploit IvyBridge processors using OpenMP [45] and auto-vectorization with the Intel15 compiler. Finally, Piz Daint is a Cray X30 supercomputer that uses the dragonfly topology for its Aries interconnect network. The 5,272 nodes are arranged into 28 cabinets. Each node has an Intel SandyBridge processor with 8 cores (Intel Xeon E5-2670) that has a peak performance of 211 GFLOPS and an Nvidia Tesla K20X GPU that has a peak performance of 3.95 TFLOPS [46]. The peak performance of Piz Daint is 7.787 PFLOPS [47] [12]. On Piz Daint, we ran TOD-Tree on the GPU using GPU Direct RDMA for communication and CUDA kernels for computation. The GPU on Piz Daint is much more powerful than the CPU on Edison: the Tesla K20X on Piz Daint has a peak performance of 3.95 TFLOPS compared to the 460.8 GFLOPS on Edison.

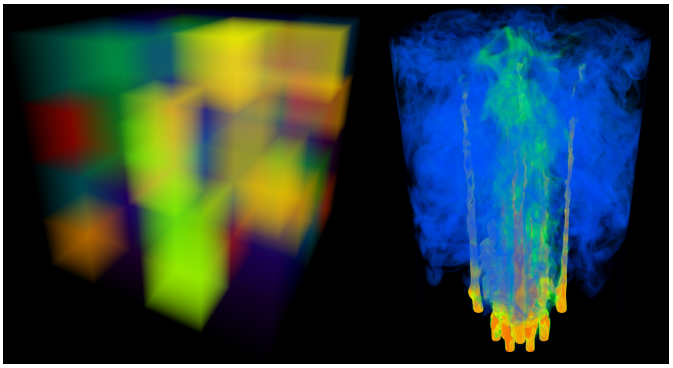

Fig. 7: Left: Synthetic dataset, Right: Combustion dataset.

The two datasets used for the tests are shown in figure 7. The artificial dataset is a square block where each node is assigned one sub-block. The simulation dataset is a rectangular combustion dataset where the bottom right and left regions are empty. The artificial dataset is a volume of size 512x512x512 voxels, and the images sizes for the test are 2048x2048 pixels (64 MB), 4096x4096 pixels (256 MB) and 8192x8192 pixels (1 GB). The combustion dataset is a volume of size 416x663x416 voxels. For the image size, the width has been set to 2048, 4096 and 8192. The heights are 2605, 5204 and 10418 pixels, respectively.

On Edison at NERSC and Piz Daint at CSCS, we were able to get access to up to 4,096 nodes whereas on Stampede at TACC, we scale only to a maximum of 1,024 nodes. In the next section, we show the performance on the three systems. Each experiment is run 10 times after an initial warm-up

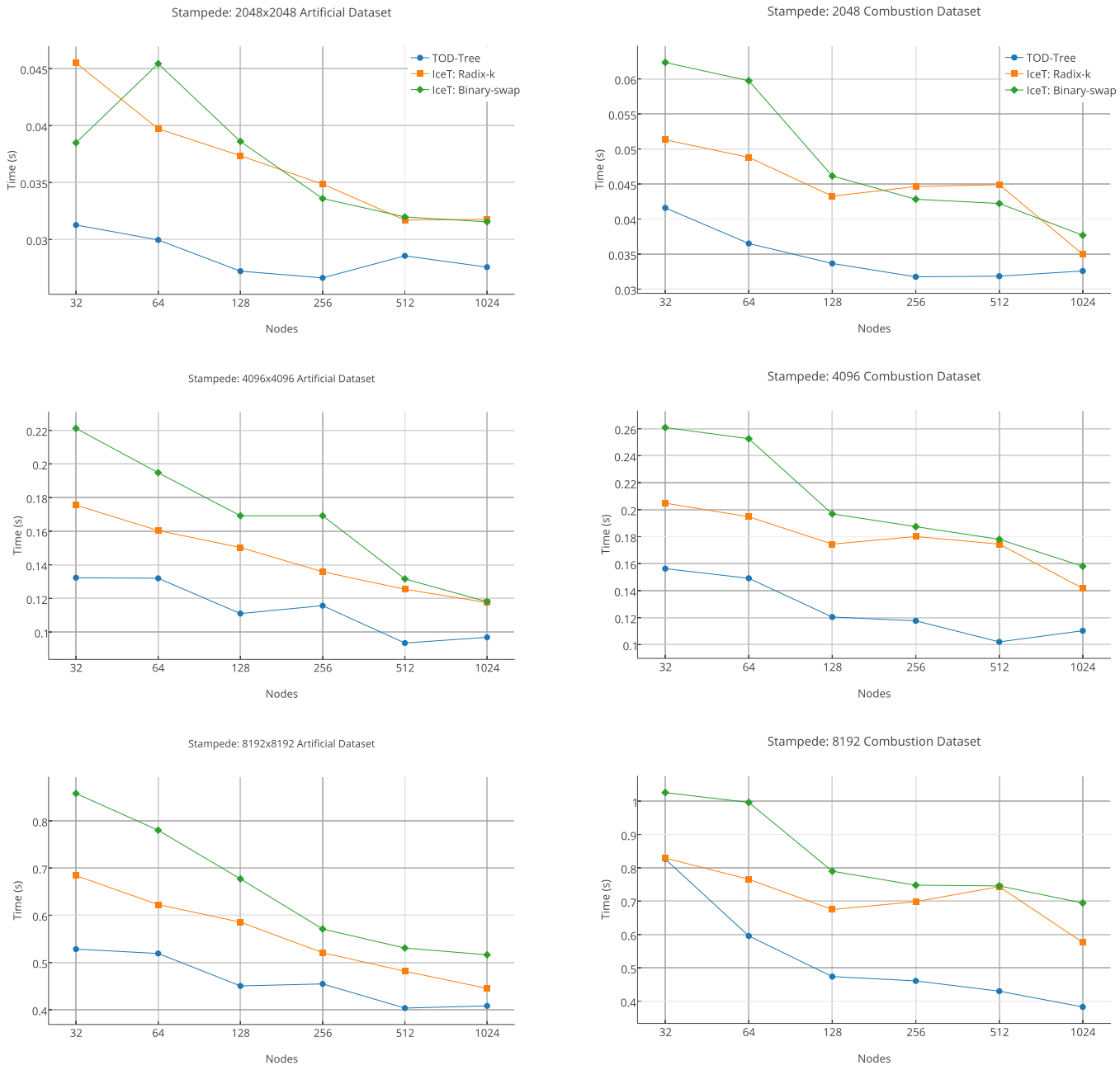

Fig. 8: Scaling on Stampede.

run, and the results are the average of these runs after some outliers have been eliminated.

4.1 Scalability on Stampede

When running on Stampede, threads are not being used for the TOD-Tree algorithm. Both IceT and our implementation are compiled with g++ and O3 optimization to keep the comparison fair and also to point to the fact that it is the overlapping of tasks rather than raw computing power that is most important here. Also, we are not using any compression as most image sizes commonly used are small enough that compression does not make a big difference. At 8192x8192 pixels, an image is now 1GB in size, and having compression would likely further reduce communication.

The left column of figure 8 shows the strong scaling results for artificial data on Stampede. The TOD-Tree algorithm performs better than binary-swap and radix-k. The sawtooth-like appearance can be explained by our use of the same value of r for pairs of time steps; r=16 for 32 and 64 nodes, r=32 for 128 and 256 and, r=64 for 512 and 1024, and only 1 round was used for the k-ary tree part of the algorithm. Thus with r=32, for 256 nodes, there are 8 groups of direct sends whereas there are only 4 groups of direct sends at 128 nodes. Therefore the tree stage must now gather from 7 instead of from 3 processes and so the time taken increases. In addition, instead of waiting for 3 nodes to complete their grouped direct send, now the wait is for 7 nodes. Increasing the value of r helps balance the workload

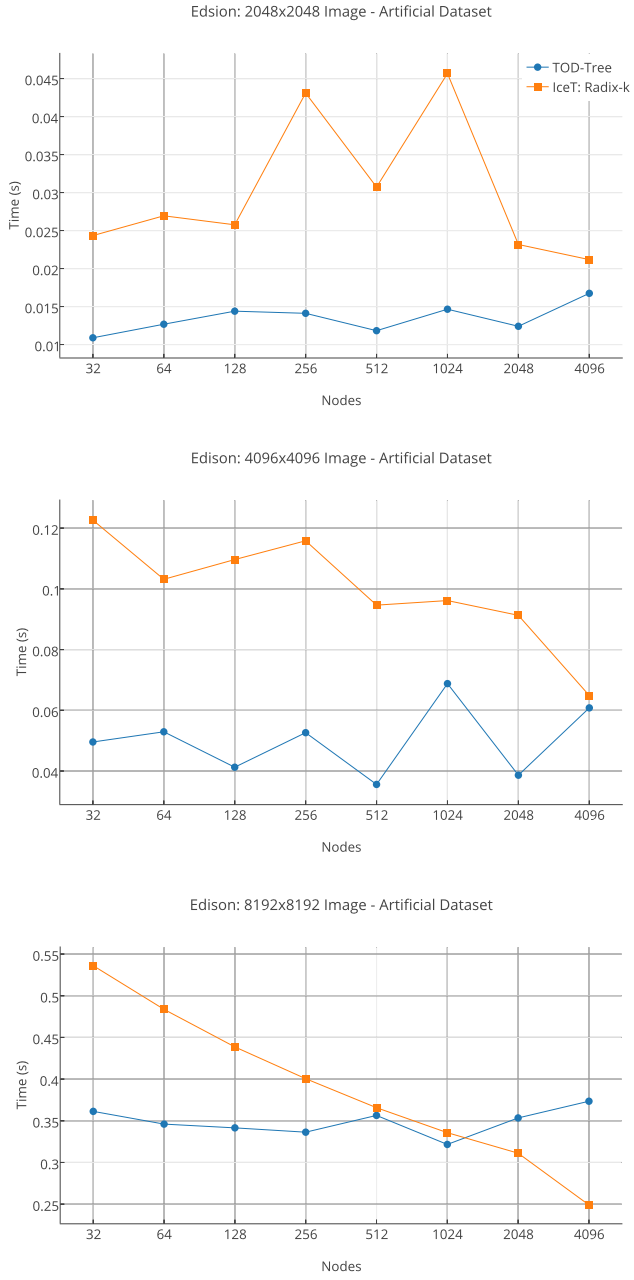

Fig. 9: Scaling for artificial dataset on Edison.

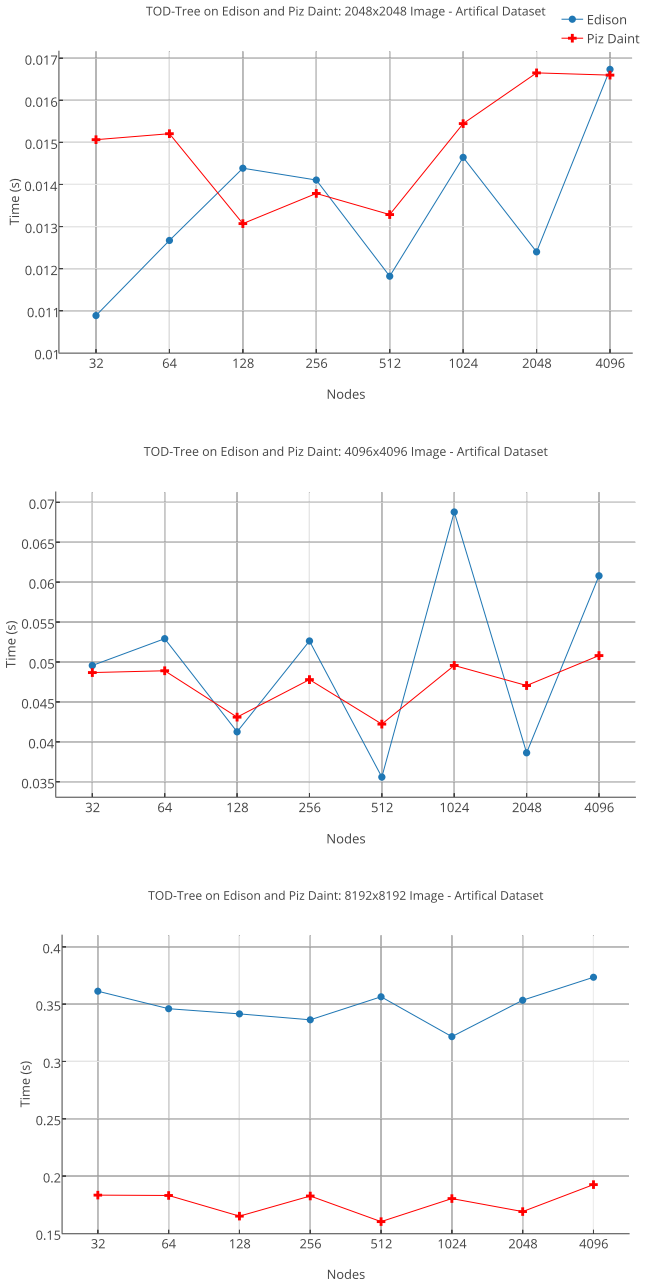

Fig. 10: Comparing scaling for Edison and Piz Daint.

in stage 1 of the algorithm and reduces the number of nodes that have to be involved in the tree compositing, and hence decreases the sending time.

For images of size 2048x2048 pixels, compositing is heavily communication bound. As we increase the number of nodes, each node has very little data and so all 3 algorithms surveyed perform with less consistency as they become even more communication bound and so more affected by load imbalance and networking issues. Communication is the main discriminating factor for small image sizes. For 8192x8192 images, there is less variation as the compositing for 8192x8192 images is more computation bound. Also, at that image size, IceT's radix-k comes close to matching the performance of our algorithm. On analyzing the results for TOD-Tree, we saw that the communication, especially in the gather stage, was quite expensive. A 2048x2048 image is only 64 MB, but a 8192x8192 image is 1GB and transferring such big sizes is expensive without compression, which is when IceT's use of compression for all communication becomes useful.

The right column of figure 8 shows the results for the combustion dataset on Stampede. One of the key characteristics of this dataset is the empty regions at the bottom that create load imbalances. Also, the dataset is rectangular and not as uniform as the artificial dataset, but it resembles more closely what users are likely to be rendering. The load imbalance creates some situations different from those in the regular dataset that affect the IceT library more than

they affect the TOD-Tree compositing because both binary-swap and radix-k give greater importance to load balancing and if the data is not uniform, they are likely to suffer from more load imbalances. Load balancing is less important to the TOD-Tree algorithm.

4.2 Scalability on Edison

On Edison, we managed to scale up to 4,096 nodes. The results for strong scaling are shown in figure 9. The performance of IceT's binary-swap was quite irregular on Edison. For example, for the 4096x4096 image, it would suddenly jump to 0.49 seconds after being similar to radix-k for lower node counts (around 0.11 s). We therefore decided to exclude binary-swap from these scalings graphs. The sawtooth pattern is similar to what we see on Stampede for TOD-Tree. Both TOD-Tree and radix-k show less consistency on Edison compared to Stampede. On Edison, 8192x8192 images at 2048 and 4096 nodes are the only instances where radix-k performed better than the TOD-Tree algorithm. Again, the main culprit was communication time and TOD-Tree not using compression. In the future, we plan to extend TOD-Tree to have compression for large image sizes.

4.3 Scaling on Piz Daint

On Piz Daint, we had access to 3,000 node hours, which did not allow us to run as many tests as on the other platforms, but we still managed to scale up to 4096 nodes/GPUs using the TOD-Tree algorithm for 2048x2048, 4096x4096 and 8192x8192 images for the artificial dataset.

The 2048x2048 image, topmost graph in figure 10, has numerous fluctuations. These fluctuations, however, all take place within 6 milliseconds, meaning that they will barely affect the rendering frame rate. For 2048x2048 images, the overall size of the full image is only 64MB, and the many variations can be explained by the fact that performance is mainly communication bound. These fluctuations decrease as the size of the image increases, and the compositing starts to be more computation bound than communication bound. The average coefficient of variation for compositing time is 10.3% for 2048x2048 images, 3.7% for 4096x4096 images and 1.8% for 8192x8192 images. The sawtooth appearance is similar to what we see on Edison and Stampede since the same values are used for the parameters r and k for the same number of MPI processes on all three systems.

We compared running TOD-Tree on Edison with Piz Daint since we ran with the same number of MPI processes on each, and both Edison and Piz Daint are CRAY XC30 systems with the same dragonfly topology and Aries interconnect network. The compositing times are very close for 2048x2048 and 4096x4096. The difference in time is within 5 milliseconds for 2048x2048 images and usually within 10 milliseconds for 4096x4096 images with a maximum variation of 20 milliseconds at 1024 nodes. For the 8192x8192 image, TOD-Tree is much faster on Piz Daint because we believe that for 8192x8192 image, compositing is more computation bound and computation on Piz Daint is faster than on Edison. If we compare the increase in average compositing time for the 2048x2048 to 4096x4096 image (for which the size increases by 4), we see that it has increased by, on average, 3.7 times on Edison and 3.2 times on Piz Daint. For 4096x4096 to 8192x8192, the average increase in compositing time is 7.2 on Edison compared to 3.7 on Piz Daint, again for a size increase of a factor of 4. The increase in time on the GPU is quite consistent as shown in figure 11.

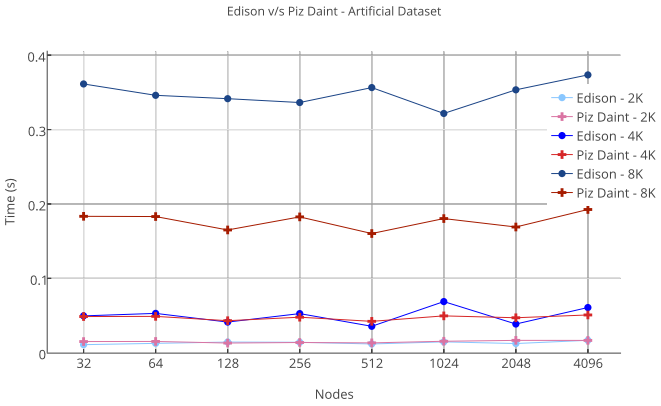

Fig. 11: Comparing scaling on Edison and Piz Daint for 4096 MPI processes.

4.4 Scaling across machines

Figure 12 shows the result of TOD-Tree algorithm on Stampede, Edison and Piz Daint. The values of r and k used are the same on all three supercomputers. As expected, the algorithm is faster on Edison and Piz Daint compared to Stampede: the Aries interconnect on the CRAY XC30 is faster and the nodes have better peak FLOP performance. While on Stampede, we are not using threads; on Edison we are using threads and vectorization and using CUDA kernels on Piz Daint. The gap between the performance is larger for low node counts, as each node has a bigger chunk of the image to process when few nodes are involved, and so a faster processor makes quite a big difference. As the number of nodes increases, the data to process decreases and so the difference in computing power is less important as the compositing becomes communication bound. The sawtooth appearance is present on all three systems. On

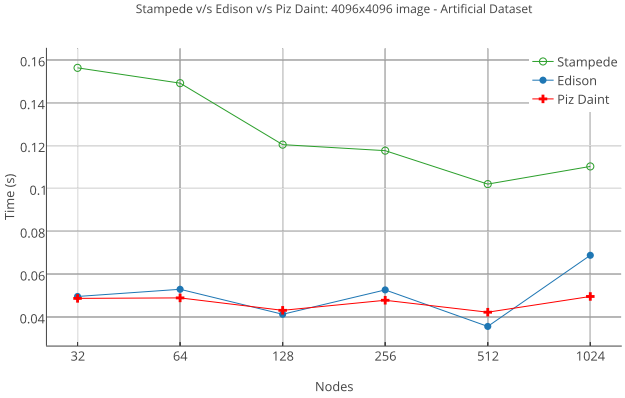

Fig. 12: Comparing Stampede and Edison for up to 1024 nodes for the artificial dataset at 4096x4096 resolution.

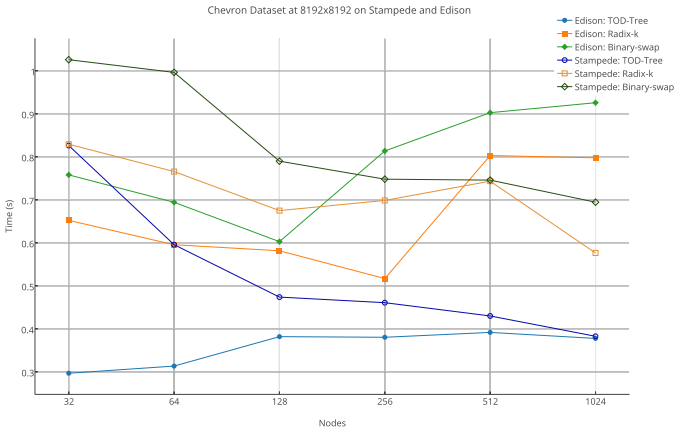

Fig. 13: Comparing Stampede and Edison for up to 1024 nodes for combustion at 8192x10418 resolution.

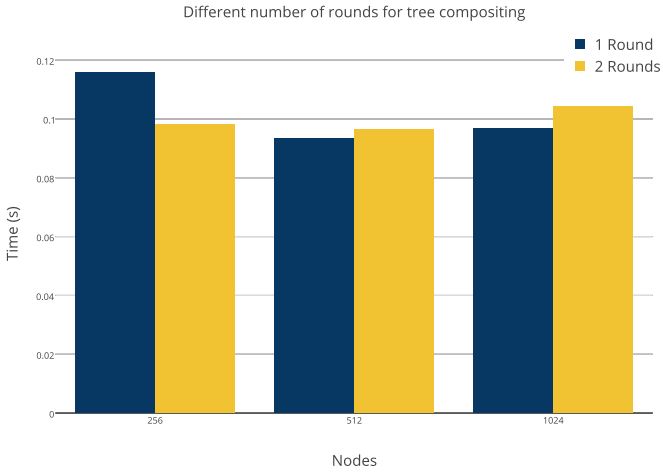

Fig. 14: Varying number of rounds for the artificial dataset for 4096x4096 on Stampede.

average, we are still getting about 16 frames per second for a 256MB images (4096x4096 pixels). At 2048 nodes, the time taken for TOD-Tree decreases, as can be seen in the middle chart of figure 10.

Figure 13 shows the equivalent comparison but with 8192x10418 images for the combustion dataset on Stampede and Edison. It is interesting to note that although the gap in performance of TOD-Tree on these two systems is initially quite large, it decreases as the number of nodes increases, again because initially there is a great deal of computation required, and so having a powerful CPU is beneficial. However, when there is less computation to do, the difference in computation power is no longer that important. IceT performs less consistently for this dataset, probably because of the load imbalance inherent in the dataset.

Also, in all the test cases, we used only 1 round for the tree compositing. For large node counts, more rounds could be used. Figure 14 shows the impact of having a different number of rounds for large node counts on Stampede. For 256 nodes, there is an improvement of 0.018 s but it is slower by 0.003 s for 512 nodes and 0.007 seconds for 1024 nodes. Therefore, having several rounds barely slows down the algorithm and can even speed up the results.

5 Conclusion and Future Work

In this paper, we have introduced an image compositing algorithm, TOD-Tree, for hybrid OpenMP/MPI parallelism and GPU clusters. We have also shown that TOD-Tree generally performs better than the two leading compositing algorithms, binary-swap and radix-k, in a hybrid programming environment. TOD-Tree performs equally well on GPU-accelerated supercomputers, which are even better for large images due to the higher peak performance of GPUs. There is a large difference between the computational power available to one node compared to the speed of inter-node communication. Computation is usually at least one order of magnitude faster than communication, and so algorithms must be designed to pay much more attention to communication than to computation if we are to achieve better performance at scale. Also, we have introduced a workflow that enables us to seamlessly transfer data from OpenGL to CUDA to allow faster overall rendering that can be easily integrated with existing GPU volume rendering systems.

As future work, we would like to add compression for large image sizes. A heuristic should also be added to determine when compression should be turned on or off based on the size of the data. Although 8192x8192 image sizes are quite rare right now (since we lack the ability to display such images properly), they will likely be required in the future, and so taking care of this will make the TOD-Tree algorithm more robust. We would also like to extend our testing to Blue Gene/Q systems because this is the only major HPC platform on which the compositing algorithm has not been tested. We plan to extend testing to the Intel Knights Landing when they are introduced. Finally, we would like to investigate how the change to many-core architectures affects image compositing algorithms. Simple image compositing algorithms such as direct send and tree compositing have been discarded in favor of more complex algorithms, but we probably do not need complex compositing algorithms for small image sizes or few nodes, especially with the huge computing power of CPUs and GPUs. We would also like to study the crossover point from simple to complex algorithms.

Acknowledgments

This research was supported by the DOE, NNSA, Award DE-NA0002375: (PSAAP) Carbon-Capture Multidisciplinary Simulation Center, the DOE SciDAC Institute of Scalable Data Management Analysis and Visualization DOE DE-SC0007446, NSF ACI-1339881, and NSF IIS-1162013.

The authors would like to thank the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing access to the Stampede and Maverick supercomputers, the National Energy Research Scientific Computing Center (NERSC) for providing access to the Edison supercomputer and the Swiss National Supercom-

puting Centre (CSCS) for providing access to the Piz Daint supercomputer.

We would also like to thank Kenneth Moreland for his help with using IceT, Peter Messmer and Thomas Fogal for their help with GPU clusters and Jean Favre and the support staff at CSCS for their help with the Piz Daint supercomputer.

REFERENCES

[1] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975. [Online]. Available: http://doi.acm.org/10.1145/361002.361007

[2] T. Fogal and J. Krüger, "Efficient I/O for Parallel Visualization," in *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*, ser. EG PGV'11. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2011, pp. 81–90. [Online]. Available: http://dx.doi.org/10.2312/EGPGV/EGPGV11/081-090

[3] H. Yu, C. Wang, R. W. Grout, J. H. Chen, and K.-L. Ma, "In Situ Visualization for Large-Scale Combustion Simulations," *IEEE Comput. Graph. Appl.*, vol. 30, no. 3, pp. 45–57, May 2010. [Online]. Available: http://dx.doi.org/10.1109/MCG.2010.55

[4] 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*, ser. VECPAR'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 1–25. [Online]. Available: http://dl.acm.org/citation.cfm?id=1964238.1964240

[5] D. A. Mallón, G. L. Taboada, C. Teijeiro, J. Touriño, B. B. Fraguela, A. Gómez, R. Doallo, and J. C. Mouriño, "Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures," in *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 174–184.

[6] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes," in *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, ser. PDP '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 427–436. [Online]. Available: http://dx.doi.org/10.1109/PDP.2009.43

[7] M. Howison, E. W. Bethel, and H. Childs, "MPI-hybrid Parallelism for Volume Rendering on Large, Multi-core Systems," in *Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization*, ser. EG PGV'10. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2010, pp. 1–10. [Online]. Available: http://dx.doi.org/10.2312/EGPGV/EGPGV10/001-010

[8] M. Howison, E. Bethel, and H. Childs, "Hybrid Parallelism for Volume Rendering on Large-, Multi-, and Many-Core Systems," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 18, no. 1, pp. 17–29, Jan 2012.

[9] K.-L. Ma, J. Painter, C. Hansen, and M. Krogh, "A data distributed, parallel algorithm for ray-traced volume rendering," in *Parallel Rendering Symposium, 1993*, 1993, pp. 15–22, 105.

[10] 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*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 4:1–4:10. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654064

[11] K. Moreland, W. Kendall, T. Peterka, and J. Huang, "An Image Compositing Solution at Scale," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 25:1–25:10. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063417

[12] T. 500. (2015, October) Top 500 List - June 2015. [Online]. Available: http://www.top500.org/list/2015/06/

[13] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 451–460, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1816038.1816021

[14] A. V. P. Grosset, M. Prasad, C. Christensen, A. Knoll, and C. D. Hansen, "TOD-Tree: Task-Overlapped Direct send Tree Image Compositing for Hybrid MPI Parallelism," in *Eurographics Symposium on Parallel Graphics and Visualization, Cagliari, Sardinia, Italy, May 25 - 26, 2015.*, 2015, pp. 67–76. [Online]. Available: http://dx.doi.org/10.2312/pgv.20151157

[15] J. Xie, H. Yu, and K.-L. Ma, "Visualizing large 3D geodesic grid data with massively distributed GPUs," in *Large Data Analysis and Visualization (LDAV), 2014 IEEE 4th Symposium on*, Nov 2014, pp. 3–10.

[16] S. Marchesin, C. Mongenet, and J.-M. Dischler, "Multi-GPU Sort-last Volume Visualization," in *Proceedings of the 8th Eurographics Conference on Parallel Graphics and Visualization*, ser. EGPGV '08. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2008, pp. 1–8. [Online]. Available: http://dx.doi.org/10.2312/EGPGV/EGPGV08/001-008

[17] S. Rizzi, M. Hereld, J. Insley, M. E. Papka, T. Uram, and V. Vishwanath, "Performance Modeling of vl3 Volume Rendering on GPU-Based Clusters," in *Eurographics Symposium on Parallel Graphics and Visualization*, M. Amor and M. Hadwiger, Eds. The Eurographics Association, 2014.

[18] J. Ahrens, B. Geveci, and C. Law, "Paraview: An end-user tool for large-data visualization," *The Visualization Handbook, Elsevier*, p. 717, Jan. 2005.

[19] H. Childs, E. S. Brugger, K. S. Bonnell, J. S. Meredith, M. Miller, B. J. Whitlock, and N. Max, "A Contract-Based System for Large Data Visualization," in *Proceedings of IEEE Visualization 2005*, 2005, pp. 190–198.

[20] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A sorting classification of parallel rendering," *Computer Graphics and Applications, IEEE*, vol. 14, no. 4, pp. 23–32, 1994.

[21] W. M. Hsu, "Segmented Ray Casting for Data Parallel Volume Rendering," in *Proceedings of the 1993 Symposium on Parallel Rendering*, ser. PRS '93. New York, NY, USA: ACM, 1993, pp. 7–14. [Online]. Available: http://doi.acm.org/10.1145/166181.166182

[22] U. Neumann, "Communication Costs for Parallel Volume-Rendering Algorithms," *IEEE Comput. Graph. Appl.*, vol. 14, no. 4, pp. 49–58, Jul. 1994. [Online]. Available: http://dx.doi.org/10.1109/38.291531

[23] A. Stompel, K.-L. Ma, E. B. Lum, J. Ahrens, and J. Patchett, "Slic: Scheduled linear image compositing for parallel volume rendering," in *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, ser. PVG '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 6–. [Online]. Available: http://dx.doi.org/10.1109/PVGS.2003.1249040

[24] 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*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 48:1–48:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413419

[25] K. Moreland, "IceT Users' Guide and Reference," Sandia National Lab, Tech. Rep., January 2011.

[26] D. Pugmire, L. Monroe, C. Connor Davenport, A. DuBois, D. DuBois, and S. Poole, "Npu-based image compositing in a distributed visualization system," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 4, pp. 798–809, Jul. 2007. [Online]. Available: http://dx.doi.org/10.1109/TVCG.2007.1026

[27] X. Cavin and O. Demengeon, "Shift-Based Parallel Image Compositing on InfiniBand TM Fat-Trees," in *Eurographics Symposium on Parallel Graphics and Visualization*, H. Childs, T. Kuhlen, and F. Marton, Eds. The Eurographics Association, 2012.

[28] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski, "Chromium: A Stream-processing Framework for Interactive Rendering on Clusters," *ACM Trans. Graph.*, vol. 21, no. 3, pp. 693–702, Jul. 2002. [Online]. Available: http://doi.acm.org/10.1145/566654.566639

[29] S. Eilemann, M. Makhinya, and R. Pajarola, "Equalizer: A Scalable Parallel Rendering Framework," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 3, pp. 436–452, May 2009. [Online]. Available: http://dx.doi.org/10.1109/TVCG.2008.104

[30] T. J. Cullip and U. Neumann, "Accelerating Volume Reconstruction With 3D Texture Hardware," Chapel Hill, NC, USA, Tech. Rep., 1994.

[31] J. Kruger and R. Westermann, "Acceleration Techniques for GPU-based Volume Rendering," in *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, ser. VIS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 38–. [Online]. Available: http://dx.doi.org/10.1109/VIS.2003.10001

[32] C. Müller, M. Strengert, and T. Ertl, "Optimized Volume Raycasting for Graphics-hardware-based Cluster Systems," in *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization*, ser. EG PGV'06. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2006, pp. 59–67. [Online]. Available: http://dx.doi.org/10.2312/EGPGV/EGPGV06/059-066

[33] T. Fogal, H. Childs, S. Shankar, J. Krüger, R. D. Bergeron, and P. Hatcher, "Large Data Visualization on Distributed Memory multi-GPU Clusters," in *Proceedings of the Conference on High Performance Graphics*, ser. HPG '10. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2010, pp. 57–66. [Online]. Available: http://dl.acm.org/citation.cfm?id=1921479.1921489

[34] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda, "MVAPICH2-GPU: Optimized GPU to GPU Communication for InfiniBand Clusters," *Comput. Sci.*, vol. 26, no. 3-4, pp. 257–266, Jun. 2011. [Online]. Available: http://dx.doi.org/10.1007/s00450-011-0171-3

[35] "GPU Direct RDMA." [Online]. Available: https://developer.nvidia.com/gpudirect

[36] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. Panda, "Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs," in *Parallel Processing (ICPP), 2013 42nd International Conference on*, Oct 2013, pp. 80–89.

[37] A. James, "An introduction to gpudirect," November 2015. [Online]. Available: https://www.brainshark.com/nvidia/intro-to-GPUDirect

[38] Nvidia, "Remote Visualization on Server-Class Tesla GPUs," Nvidia, White Paper WP-07313-001-v01, June 2014.

[39] M. D. Klein and J. E. Stone, "Unlocking the Full Potential of the Cray XK7 Accelerator Mark," in *Cray User Group Conference*. Cray, May 2014.

[40] C. D. Shaw, M. Green, and J. Schaeffer, "Advances in Computer Graphics Hardware III," W. T. Hewitt, R. Gnatz, and D. A. Duce, Eds. New York, NY, USA: Springer-Verlag New York, Inc., 1991, ch. A VLSI Architecture for Image Composition, pp. 183–199. [Online]. Available: http://dl.acm.org/citation.cfm?id=108345.108358

[41] P. Messmer. (2016, January) Egl eye: Opengl visualization without an x server. [Online]. Available: http://devblogs.nvidia.com/parallelforall/egl-eye-opengl-visualization-without-x-server/

[42] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, "Collective Communication: Theory, Practice, and Experience: Research Articles," *Concurr. Comput. : Pract. Exper.*, vol. 19, no. 13, pp. 1749–1783, Sep. 2007. [Online]. Available: http://dx.doi.org/10.1002/cpe.v19:13

[43] NERSC. (2015, February) Edison Configuration. [Online]. Available: https://www.nersc.gov/users/computational-systems/edison/configuration/

[44] TACC. (2015, February) Stampede User Guide. [Online]. Available: https://portal.tacc.utexas.edu/user-guides/stampede

[45] L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998. [Online]. Available: http://dx.doi.org/10.1109/99.660313

[46] NVIDIA. (2015, October) NVIDIA TESLA GPU ACCELERATORS. [Online]. Available: http://www.nvidia.com/content/tesla/pdf/nvidia-tesla-kepler-family-datasheet.pdf

[47] CSCS. (2015, October) Piz Daint. [Online]. Available: http://user.cscs.ch/computing_systems/piz_daint/index.html

A.V.Pascal Grosset received a BSc degree in Computer Science & Engineering from the University of Mauritius, a MSc degree in Computer Graphics from the University of Teesside, England, and is currently working toward a PhD degree in Computing: Graphics and Visualization at the University of Utah. He is a recipient of the 2009 International Fulbright Science & Technology Award. His research interests include visualization and High Performance Computing.

Manasa Prasad received a B.Tech. degree in Information Technology from the SRM University, India and a MSc degree in Computing: Graphics and Visualization at the University of Utah in 2015. Currently she is a Software Engineer at Google.

Cameron Christensen received a BS in Computer Science from the University of Utah. He joined the SCI Institute as a software engineer in 2010 and has developed tools to assemble, visualize, and annotate massive 2D and 3D microscopy images. Cameron is part of the Center for Extreme Data Management Analysis and Visualization (CEDMAV) and contributes to the development of software to process and visualize extreme-scale multiresolution data, with applications in global climate analysis, combustion simulation, microscopy, and education. Cameron is currently pursuing a masters of computing under the advisorship of Prof. Valerio Pascucci. His research interests include streaming data analysis, visualization, and computer graphics.

Aaron Knoll received his Ph.D. from the University of Utah in 2009, researching efficient ray tracing methods for implicit surfaces. He explored fast CPU and GPU volume rendering, multi-field analysis and molecular visualization at postdoctoral fellowships at the University of Kaiserslautern and Argonne National Laboratory. At the Texas Advanced Computing Center at the University of Texas at Austin, he helped deploy visualization software on supercomputers and developed in situ visualization solutions for the Intel Xeon Phi architecture. Now a research scientist at the SCI Institute, he works with Valerio Pascucci and the CEDMAV group on solutions for large particle data, and merging in-core and out-of-core visualization methodologies.

Charles Hansen received a PhD in computer science from the University of Utah in 1987. He is a professor of computer science at the University of Utah and an associate director of the SCI Institute. From 1989 to 1997, he was a Technical Staff Member in the Advanced Computing Laboratory(ACL) located at Los Alamos National Laboratory, where he formed and directed the visualization efforts in the ACL. He was a Bourse de Chateaubriand PostDoc Fellow at INRIA, Rocquencourt France, in 1987 and 1988. His research interests include large-scale scientific visualization and computer graphics.