Eurographics Symposium on Parallel Graphics and Visualization (2016)
W. Bethel, E. Gobbetti (Editors)

Dynamically Scheduled Region-Based Image Compositing

A.V.Pascal Grosset, Aaron Knoll, & Charles Hansen

Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, UT, USA

Abstract

Algorithms for sort-last parallel volume rendering on large distributed memory machines usually divide a dataset equally across all nodes for rendering. Depending on the features that a user wants to see in a dataset, all the nodes will rarely finish rendering at the same time. Existing compositing algorithms do not often take this into consideration, which can lead to significant delays when nodes that are compositing wait for other nodes that are still rendering. In this paper, we present an image compositing algorithm that uses spatial and temporal awareness to dynamically schedule the exchange of regions in an image and progressively composite images as they become available. Running on the Edison supercomputer at NERSC, we show that a scheduler-based algorithm with awareness of the spatial contribution from each rendering node can outperform traditional image compositing algorithms.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture—Parallel processing I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics

1. Introduction

Visualization is increasingly important in the scientific community. Several High Performance Computing (HPC) centers, such as the Texas Advanced Computing Center (TACC) and Livermore Computing Center (LC), now have clusters dedicated to visualization. Most clusters in HPC centers are usually distributed memory machines with hundreds or thousands of nodes, each of which has a very powerful CPU and/or GPU with lots of memory, connected through a high-speed network. The most commonly used approach for parallel rendering on these systems is sort-last [MCEF94]. In sort-last parallel rendering, the data to be visualized is equally distributed among the nodes. Each node loads its assigned subset of the dataset that it renders to an image. During the compositing stage, the images are exchanged, and the final image is gathered on the display node. In this paper, our focus is on the compositing stage of distributed volume rendering.

Image compositing has two parts: computation (blending) and communication. Many algorithms, such as Binary Swap [MPHK93] and Radix-k [PGR*09], have been developed for image compositing. These algorithms try to evenly distribute the computation among the nodes. However, as shown by Grosset et al. [GPC*15], image compositing algorithms should pay more attention to communication than to computation. Nowadays, the computing power of nodes in a supercomputer greatly exceeds the communication speed between nodes. Trying to minimize communication and overlapping communication with computation is more important than focusing on evenly balancing the workload. In this paper, we focus specifically on communication, and threads and auto-vectorization are used to fully benefit from the computational power of CPUs.

The time each node takes to finish rendering its assigned region of a dataset in sort-last parallel rendering is rarely the same. There are several reasons for this, first, it is rare for datasets to have a uniform distribution of data. Figure 1 shows two commonly used test volume datasets that have numerous empty regions after a transfer function has been applied to extract interesting features in each dataset. The nodes assigned to rendering these empty regions have much less work to do and will finish early. Second, when using perspective projection, nodes closer to the camera produce a larger image compared to nodes far from the camera. Rendering a larger image takes more time than rendering a smaller image. Finally, if the user zooms in on one specific region of a dataset, part of the dataset might fall outside the viewing frustum and not need to be rendered. Moreover, the difference in rendering speed is further increased if lighting is used and normals need to be calculated, and if the rendering takes place on a medium-sized cluster where

there are hundreds rather than thousands of nodes, the time taken to render a large image can be substantially greater than the time to render a small image. If we do not want the uneven rendering to slow down compositing, nodes that are done rendering should exchange images only with nodes that are also done rendering, and not wait on nodes which are still rendering. In this paper, we keep track of which nodes are done compositing, and only schedule compositing when nodes have completed rendering.

Figure 1: *Two commonly used test datasets: the Bonsai dataset on the left and Backpack dataset on the right.*

One of the common approaches for load balancing in distributed volume rendering is to split and distribute the dataset based on how long each region will take to render, the approach used by Marchesin et al. [MMD06] and Fogal et al. [FCS*10], rather than just diving the data equally using, for example, a k-d tree. However, arbitrarily assigning data to nodes may not be an effective strategy when considering in situ visualization. Data movement, between nodes or writing to disk, is very costly in large scale simulations. As mentioned by Yu et al. [YWG*10], for in situ visualization, it is the simulation code that dictates the data partitioning and distribution among nodes. Thus, in situ visualization uses the same nodes for visualization as those generating the data in the simulation and is best performed without requiring data movement between nodes or disk. In this paper, we propose that work is scheduled at the compositing stage and does not require data redistribution for balanced rendering. Since the image compositing only transfers sub-images, our proposed technique would be easily integrated with existing in situ visualization and analysis software.

The main contribution of this paper is an image compositing algorithm that uses a scheduler with both spatial and temporal awareness of the compositing process. We start by dividing the final image into a number of regions r and create a depth-ordered list of nodes for each region. Based on the data loaded by each node and the properties of the camera, the contribution of each node to regions of the final image can be determined. Nodes not contributing to a region can then be removed from that region's list. The scheduler also updates the region list after each node is done rendering by eliminating nodes that rendered nothing for a region. This process ensures that a node not contributing to a region is never made to receive data for that region, thus minimizing communication. The algorithm then schedules the exchange of images and ensures that no nodes wait for a node that is still rendering if another option for compositing is available. Thus, when the slowest node is done rendering, most of the regions of the final image have already been composited and there is minimal overhead to assemble the final image. The algorithm uses one MPI rank per node and threads for CPU cores, which Howison et al. [HBC10] showed to be better than one MPI rank per core. Auto-vectorization is also used to fully leverage the compute capabilities of modern CPUs, and asynchronous MPI communication is also used to overlap communication with computation. We compare this scheduling-based image compositing algorithm against TOD-Tree on the Edison supercomputer at NERSC using a box and sphere artificial dataset and a combustion dataset.

The paper is organized as follows: Section 2 describes the commonly used compositing algorithms and techniques that are used to achieve load balance in distributed volume rendering. Section 3 describes our algorithm and the implementation details. The results are presented in section 4 where we also discuss the results of strong scaling on three types of datasets. Finally, the paper is wrapped up in section 5 with the conclusion and future work.

2. Previous Work

Many algorithms have been designed to tackle image compositing in distributed volume rendering. The simplest algorithm is serial direct send in which all the processes involved in rendering send their rendered image to the display node, which then blends them together. This approach results in a massive load imbalance and can be quite slow for large images and many nodes. Parallel direct send [Hsu93], [Neu94] improves on serial direct send by dividing the workload among all the processes. Each process is responsible for one section of the final image and gathers that section from all other processes. In the gathering stage, each process sends its authoritative section to the display node.

Tree-based algorithms have also been used for image compositing. In binary tree compositing, each node is represented as a leaf of the tree. The height h of the tree is $log_2 n$ where n is the number of nodes in the tree. In each subtree, a child sends its data to its sibling for blending and becomes inactive. This exchange goes on for each level of the tree until the final image is at the root (display node) of the tree. Binary Swap [MPHK93] improves on this approach by keeping all nodes active until the gathering stage. Initially, each node is responsible for the whole image. At each level of the tree, nodes responsible for the same region are paired in a subtree and exchange information so that each is responsible for half of the image for which it was initially responsible for. This process continues until there is only one node responsible for each $1/n$ section of the final image. Then the display node gathers these sections from all

n nodes. Binary Swap has been extended for non-powers of 2 by Yu et al. [YWM08]. In Radix-k [PGR*09], instead of grouping nodes in pairs for a round, the size of regions to be grouped is determined by a vector k where $k = k_1, k_2, ...$ In each k_i-sized group, the nodes exchange information, in a parallel direct send way, so that each node in a k_i-sized group is responsible for $1/k_i$ of the final image. All nodes with the same authoritative section of the image are then collected into groups of size k_{i+1}, which continues for i rounds, followed by a gather stage in which each authoritative section is assembled on the display node. These algorithms have been implemented by Moreland et al. [MKPH11] in ICET [Mor11] along with some optimizations for communication.

To account for the much faster compute speed compared to communication speed in supercomputers, Grosset et al. [GPC*15] developed the TOD-Tree algorithm, which focuses on reducing and overlapping communication with computation. TOD-Tree has a parallel direct send stage to balance the workload, followed by k-ary compositing to reduce communication. Also, Howison et al. [HBC12] showed that parallel rendering is faster when one MPI rank is used per node instead of per core. Therefore, we also use one MPI rank per node, and we use threads and auto-vectorization on the CPU.

However, although these algorithms are fast, they do no take into account the contents of the image from each rendering process. They all decide statically for which region a computing process should be responsible and stick to that allocation. A process, then, may be responsible for a region for which it does not have any initial content, which needlessly increases communication. However, some algorithms take into account the image contents of a node. The Scheduled Linear Image Compositing (SLIC) algorithm of Stompel et al. [SML*03] ensures that the region to which a process is assigned is the one to which it contributes. The contribution to the final image from each process is computed based on the data extents loaded by a process and the camera position. Scan lines of the overlapping regions are assigned to processes contributing to them in an interleaving fashion. Also, image regions that do not overlap with other images are directly sent to the display node without any blending. Strengert et al. [SMW*04] used the SLIC algorithm for image compositing on GPU clusters. However, although SLIC has spatial awareness of the contribution of each rendering process, it does not have any temporal awareness, that is, it does not know when a process will finish rendering and is ready to participate in compositing.

Load balancing approaches to distributed volume rendering usually take rendering time into account when compositing. Fang et al. [FSZ*10] use a pipeline approach in which they overlap rendering and compositing. Systems that provide a complete solution to rendering and compositing, such as Equalizer system [EMP09] and Chromium [HHN*02], have knowledge of both compositing and rendering that gives them more flexibility to balance the workload. The Equalizer framework uses the direct send technique of Eilemann et al. [EP07] that splits images into tiles to improve image compositing. Other approaches, such as that of Moloney et al. [MWMS07], use an estimate on the cost to render a pixel to do dynamic load balancing using a sort-first rendering approach, and Muller et al. [MSE07], Fogal et al. [FCS*10], and Marchesin et al. [MMD06] use the previous rendering time in a time varying datatset to estimate the cost of rendering the current timestep. Frey and Ertl [FE11] redundantly distribute blocks of volume data across the rendering nodes to allow for easier load balancing. In this paper, we do not try to move the data between nodes and estimate the rendering time. Instead, we communicate with the rendering nodes to schedule compositing accordingly. Being able to move the data between nodes may help reduce the rendering imbalance among nodes, but in the case of in situ visualization, data movement is often too costly and we have to use the data decomposition dictated by the simulation. In these situations, the only place to deal with load imbalance would be at the compositing stage.

3. Methodology

It is rare for rendering on all the nodes of a distributed memory machine to finish at the same time. Improving compositing time, therefore, requires minimizing the time between when the slowest process finishes rendering and compositing is complete; the orange region in figure 2. For that to happen, processes still rendering should not delay compositing.

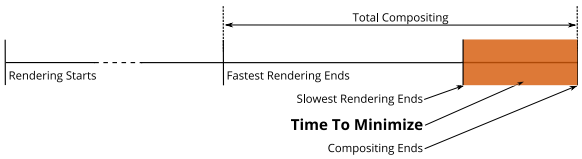

Figure 2: *Rendering + Compositing timeline.*

One of the issues with compositing algorithms such as parallel Direct Send, Binary Swap, Radix-k, and TOD-Tree is their lack of awareness of which processes have finished rendering and which processes are still rendering, which sometimes delays image compositing as some processes wait for images from other processes that are still rendering. Figure 3 shows an example of eight processes doing compositing using Radix-k. Let us assume that two rounds are needed and vector $k = 4, 2$. To get the correct final image, partial images need to be blended in the correct depth order (front-to-back or back-to-front). So, if processes 6 and 0, in figure 3, are still rendering while the remaining processes are compositing, Radix-k will have to wait for 6 and 0 to finish rendering and be stuck in round 1 of parallel direct send for

all regions. The same delay would occur in Binary Swap and TOD-Tree if some nodes waiting to exchange images with nodes that are still rendering since they too lack **temporal awareness**.

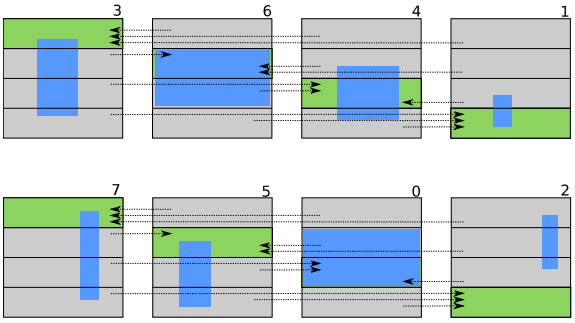

Figure 3: *The first round of Radix-k for eight processes and four regions. The green rectangle shows the region for which each process is responsible and the blue region shows the data from each process. Process 6 and 0 have more data to render and will finish rendering after the other processes.*

The same set of processes can be represented as a graph as shown in figure 4. If we blend exclusively based on depth, processes 4, 1, 7, and 5 can start compositing while waiting for 6 and 0 to finish rendering. Also, since there are never any cycles in the graph, we will refer to it as a chain.

This procedure, however, can still be improved upon. If 6 and 0 do not contain information relevant to the whole image, they should not delay compositing for the whole image. It is common for compositing algorithms to divide an image into regions and allocate each region to a process. If, for example, four regions are used as shown in figure 3, processes 6 and 0 do not contribute to the first and last regions, and so they should not delay compositing for these regions of the image. If we use a chain to represent each region, process 6 and 0 will be omitted from the first and last chain. As the number of processes increases to hundreds or even thousands, the contribution of one process to the whole image decreases. Therefore, stalling the whole compositing because of a few slow rendering processes can be avoided; we need to stall a only few regions. Having **spatial awareness** will help mitigate this issue. Moreover, spatial awareness will prevent the algorithm from making a process authoritative on a region for which it has no data! For example, in figure 3, process 2 is responsible for the last region but has no data contributing to that region. This increases communication as process 2 has to transfer all its data to other regions and needs to get all the data for its responsible region from other processes.

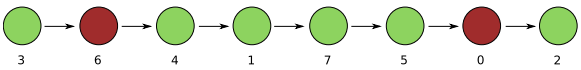

Figure 4: *Nodes sorted by depth in a chain. The red nodes are still rendering while the green nodes are done rendering.*

For our algorithm, we divide the image into a set of r regions with a depth-sorted chain for each region. To create the chains for each region, we can obtain information about the data extents each process is loading using MPI Gather, or if a k-d tree is used to partition the data, this information can be obtained programmatically for each region from the k-d tree. Using the extents and camera information, we can compute the depth of each process and the position and area contributed by each process in the final image. For each chain, we also need to decide which processes will be responsible for gathering information. To try to ensure that different nodes are used to collect information for each chain, the first collector node in the chain for region i is the i^{th} node in the chain. The second is the $(i+r)^{th}$ node. If a chain has fewer than r nodes, the last node is made the collector node for that region. The collector processes are marked with a black circle inside, as in figure 5. The number of regions in this case is 4. The first chain, chain 0 colored pink, has only three nodes. So the last node is set as the collector. The second chain, chain 1 colored cyan, has seven nodes. Therefore, node 1 and node 5 are set as collectors.

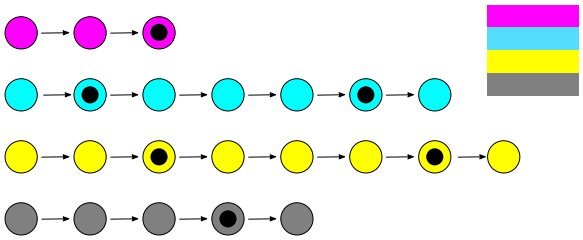

Figure 5: *Four chains, one for each of the four regions (purple, blue, yellow, and gray) into which the final image is split.*

This approach will only work for depth-orderable decomposition. Many simulations use block-structured AMR grids which are depth-orderable. If, for example, unstructured grids are used, concave regions could be generated, through domain decomposition, where sorting by depth and then compositing the various subdomains would result in incorrect images. In this paper, our focus is on block structured grids with the block-structured decomposition already imposed by the simulation.

3.1. Algorithm

For our algorithm, we have set aside one process that is not involved in compositing and rendering to act as a scheduler. The scheduler builds a chain for each region, and the compositing processes contact the scheduler to determine with

Algorithm 1: Initialize Scheduler

```
Collect the depth and extents for each process
Sort the processes based on depth
Construct a chain based on depth
for each region do
    Use the computed depth chain as the starting point
    Compute and store the extents for that region
    for each process in the chain do
        Compute the extents of the process
        if extents of process does not overlap the chain's then
            Delete the process from the chain
            Adjust the to and from neighbor for the deleted process
    if length of chain < number of regions r then
        Set the last process as a collector
    else
        Set every r process to be a collector
Create a buffer for final receive
Launch asynchronous MPI receive for final image
```

which processes they should exchange images. The chain for each region is constructed as indicated in algorithm 1.

Based on the depth information from each process, a depth-sorted chain, as shown in figure 4, is constructed that is used as the initial chain for each region. For each region, processes that do not contribute to that region are removed from the chain, which creates spatial awareness for each region and reduces the length of each chain. In software, each chain is implemented using a hash map, unordered_map in C++, so that access time is always $O(1)$, and each node of the chain stores the neighbors to and from it. The last step is to create a buffer to receive the composited image for each region. This step ensures that when the final image regions are sent to the display node, they are not written to temporary buffer but directly to the final image.

The scheduler is then started and awaits communication from the compositing processes. Algorithm 2 shows the algorithm for the scheduler. If the scheduler is receiving information from a process for the first time, it also receives the extents of the rendered image. The chain for each region is initialized based on the expected rendered extents from each process, but depending on the transfer function, some regions might not have been rendered for a process. Based on the rendered extents, therefore, some nodes are removed from region chains if they do not have any information for that region. If that process p was marked as a collector process for a specific region, its neighbor is made a collector process and the process p is deleted to minimize unnecessary transfer of data to that process.

Next, the scheduler performs dynamic scheduling by deciding which processes should communicate with each

Algorithm 2: Scheduler

```
while !done do
    Wait for communication from rendering processes
    if first communication from a process then
        Receive rendered extents from the process
        for each region do
            Determine extents of the region
            if extents of a process does not overlap the chain's then
                Remove the process from the chain
                Adjust neighbors to and from for deleted process
    Mark the process as active in the chains where they exists
    for each active chain do
        if only one process in chain then
            Mark process to send information to display node
            Erase chain
        else
            Find neighbor for incoming node
            if neighbor found then
                Determine if sender or receiver
                Mark receiver as busy
                Delete sender from chain
                Save sender and receiver information
    for each active chain do
        if size is 1 AND process is ready then
            Process will send data to display node
    for each process marked for communication do
        Send information
    if all chains are empty then
        Exit Scheduler
```

other. In each region for which the received process is active, the received node in that chain is marked as ready and the chain is checked to see if there is any neighbor process marked as ready. If a valid neighbor is found and it is a collector process, the non-collector will send its data to the collector process. Otherwise, the node having the smaller image will send data to the node with the larger image to minimize communication time. In each case, the sender node is marked for deletion and the receiver is marked as busy. The last step of the algorithm is to check if any chains are now empty or have only one remaining ready process. If there is only one ready process, it is made to send its information to the display node and the chain is erased. The next step is to send all the information at once to each node that has work to do. A process might need to send data to a node x for a specific region and receive data from the same node x for another region. All the communication to a node from the scheduler is done in one step.

Algorithm 3: Compositor

```
Get the extents of the image rendered by the process
Count the number of active regions covered by the
image (countActiveRegions)
while !done do
    if first time then
        Send extents to the scheduler
    else
        Tell scheduler that it is ready
    Wait for the scheduler to respond
    for each process to communicate with do
        if Only process in chain then
            Send data to display node
            countActiveRegions -= 1
        else
            if Send then
                Async send to neighbor
                countActiveRegions -= 1
            else
                Receive image
                if last round then
                    Create opaque image
                    Create alpha buffer
                    Blend current image with the
                    background
                    Blend in opaque buffer
                    Send to display node
                    countActiveRegions -= 1
                else
                    Blend with image on node
    if no active regions left then
        Exit loop
```

Each compositing node runs the Compositor algorithm shown in algorithm 3. The first time a process communicates with the scheduler once it is done rendering, it sends its extents to the scheduler. As mentioned before, based on the transfer function, a process will not always render all data it has loaded, and as spatial awareness is a key component of our algorithm, we want to update the region chains to reflect the state of the rendering. Also, each process will receive in one message all the other processes with which it needs to communicate to keep communication in the system to a minimum. Information for each communication will contain the neighbor with which to communicate, the region, blending direction, and MPI tag. Also, each send from a process is in the form of an asynchronous send to maximize overlapping communication with computation.

3.2. Choosing number of regions

For the scaling run, we have set the number of regions to be 16. This number was determined after a series of initial test runs where we experimented with 1, 2, 4, 8, 16, and 32 regions for 4,096 x 4,096 sized images. When few regions are used, a slow node impacts few regions, but since each region occupies a substantial portion of the image, compositing ends up being slow. For example, if we use only two regions for an 8K x 8K image, and there is one slow node in the upper region, half of the compositing is delayed by one node. If too many regions are used, one slow node will impact many small regions, but since there are many regions, the overall impact of a slow region will be less. However, many regions will result in lots of communication with many exchanges, which we want to avoid. Sixteen regions provided a good balance between avoiding too much communication and one node having too much of an impact on the whole compositing process.

4. Testing and Results

The test platform used is the Edison Cray XC30 supercomputer at NERSC. Edison uses the Cray Aries high-speed interconnect with Dragonfly topology that has an MPI bandwidth of about 8 GB/sec and latency in the range of 0.25 to 3.7 usec. It has 5,576 compute nodes, each of which has two 2.4 GHz 12-core Ivy Bridge processors with 64 GB of memory per node. We scaled up to 2,048 nodes of the 5,576 nodes of Edison.

The test datasets that we used are an artificial box and artificial sphere test dataset and a combustion dataset shown in figure 6. The combustion dataset has 106,457,688 cells, stored as doubles, and is split into 5,996 blocks for a total size of 0.9 GB. It is part of combustion dataset that has 30 scalar values per timestep and about 500 timesteps. Fuel is injected into the combustion chamber through a number of tubes located at the bottom of the dataset. Combustion starts above these tubes and rises to the top of the combustion chamber, hitting the ceiling and the walls. When visualizing this dataset, much more work has to be done in the upper regions of the dataset, thereby creating an imbalance in the rendering workload. The artificial datasets are simpler: each rendering process is assigned one block of uniform scalar data per node. The box dataset is similar to what was used by Moreland et al. [MWP01], and we also introduced a sphere dataset whose diameter is equal to the length of the cube.

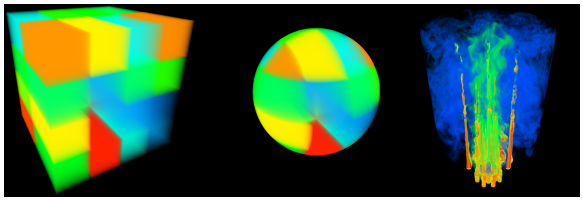

Figure 6: *The datasets: box (left), sphere (middle), and combustion (right).*

The algorithm we compared against is the TOD-Tree algorithm of Grosset et al. [GPC*15]. Grosset et al. have shown that TOD-Tree generally performs better than Radix-k and both TOD-Tree, and our algorithm uses threads and auto-vectorization compared to the ICET library [Mor11], which does not use threads.

4.1. Scheduler Cost

Building and running the scheduler is fast: the time it took to construct the region chains and using MPI Gather to collect the depth and extents information from each node, for 2,048 nodes, was measured to be on average 0.5 millisecond. The time it took the scheduler to respond to a compositing node if neighbors were available was on average 0.2 millisecond. With a latency of at most 3.7 millionth of a second, the cost

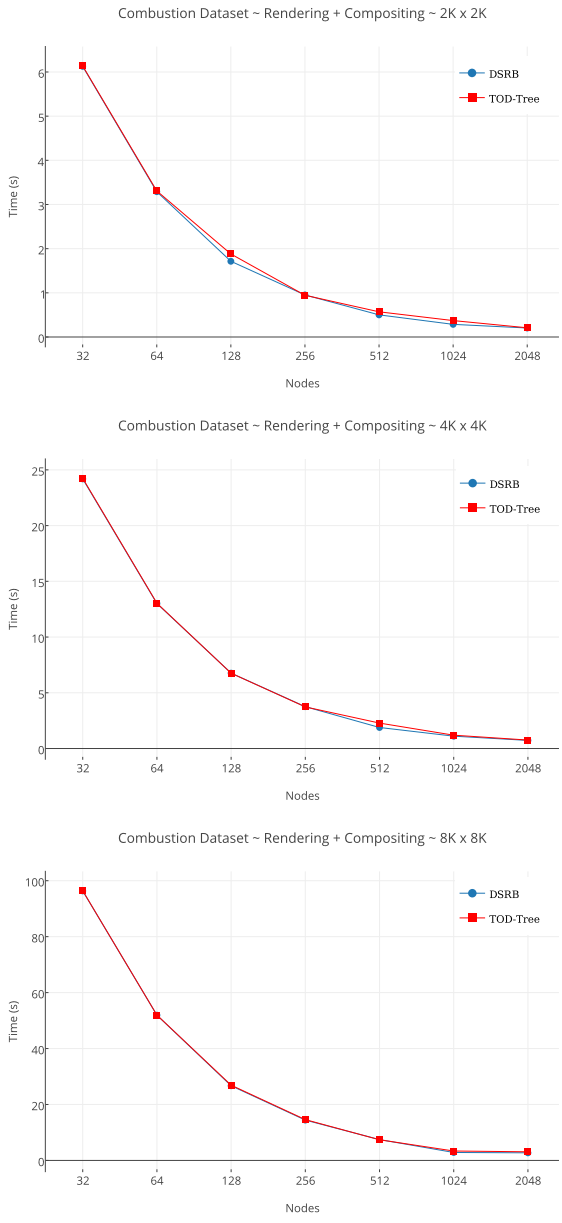

Figure 7: *Scaling of the combustion dataset on Edison - showing rendering and compositing.*

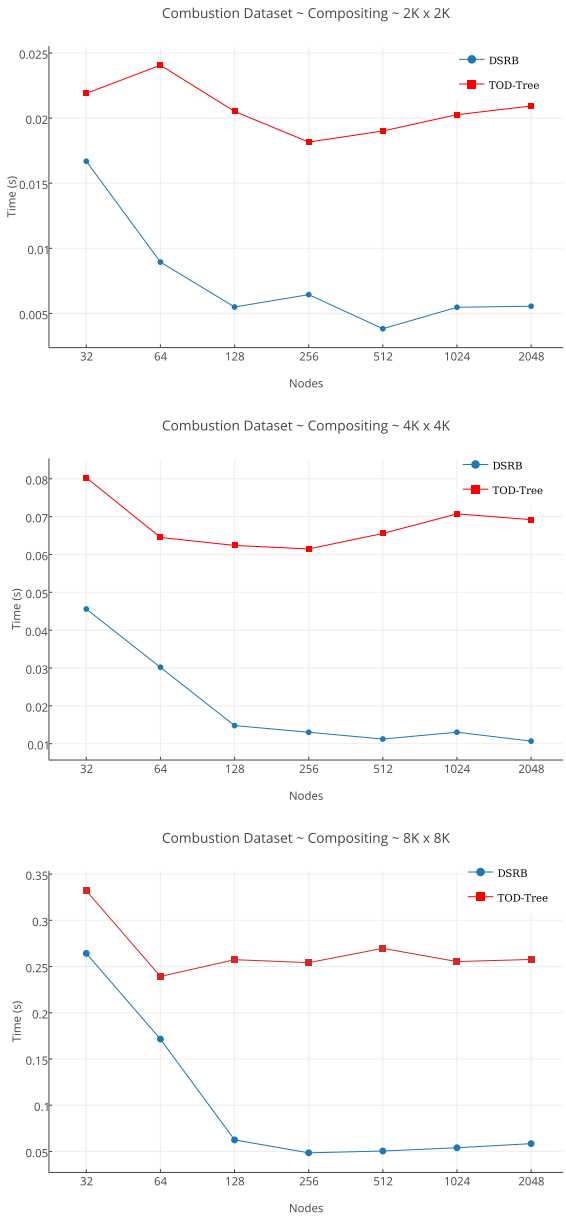

Figure 8: *Scaling of the combustion dataset on Edison - showing compositing only.*

of communicating with the scheduler is minimal compared to the cost of exchanging data among nodes.

4.2. Scaling Studies

For each of the three datasets, and for each of the three image sizes used (2,048 x 2,048, 4,096 x 4,096, and 8,192 x 8,192 pixels), we performed 10 runs after an initial warm-up run, and the results are the average of these runs after some outliers have been eliminated.

Figure 7 shows the total time it takes to render and composite the combustion dataset for up to 2048 nodes on Edison. As expected, as the number of nodes increases, the total time it takes to render the dataset decreases. The focus of this paper is image compositing and so, for the remainder of this section, we focus on compositing.

Depending on the amount of rendering work each node has to do, compositing will start at different times on each node. The compositing time that needs to be minimized is the time interval between when the slowest rendering job finishes and the final image is ready on the display node; the orange region in figure 2. Any compositing done in the interval of time between the fastest rendering node and the slowest rendering node does not slow down the entire compositing process. Therefore, the compositing time that we measured and plotted in figures 8 and 9 is the time interval between the slowest rendering job and the image being ready on the display node.

Figure 8 shows the compositing time for the combustion dataset for 2,048 x 2,048 (2Kx2K), 4,096 x 4,096 (4Kx4K), and 8,192 x 8,192 (8Kx8K) sized images for TOD-Tree and our Dynamically Scheduled Region-Based (DSRB) algorithm. When there are few nodes, each node renders a larger region and so influences many regions of the chain. We therefore do not gain much from overlapping rendering with compositing since compositing, in most regions, is stalled by waiting for other nodes. As the number of nodes increases and the contribution of each node to regions decreases, the overlapping of compositing and rendering allows us to perform better than the TOD-Tree, which does not have any spatial or temporal awareness of the image being rendered from each node. We also see that there is more variation for the 2K x 2K image compared to the 4K x 4K image and 8K x 8K image since it is more communication bound. The 8K x 8K image has the least variation as it is more computation bound.

The Dynamically Scheduled Region-Based compositing algorithm also performs faster than TOD-Tree on the artificial dataset. The difference in compositing times between the sphere and box is minimal in most cases. However, since there is less data for the sphere dataset, it takes less time to render compared to the box dataset and so has less "free compositing time" compared to the box dataset. This is translated in the chains by the box having a faster compositing time since what we are showing as compositing time is the time interval between the slowest rendering and final image being ready. For TOD-Tree, the sphere is generally faster since there is overall less data to process. As with the combustion dataset, compositing gets faster as the number of nodes increases. Here again, when more nodes are used, each node has a smaller share of the entire image, and a slow node impacts fewer regions, resulting in faster compositing.

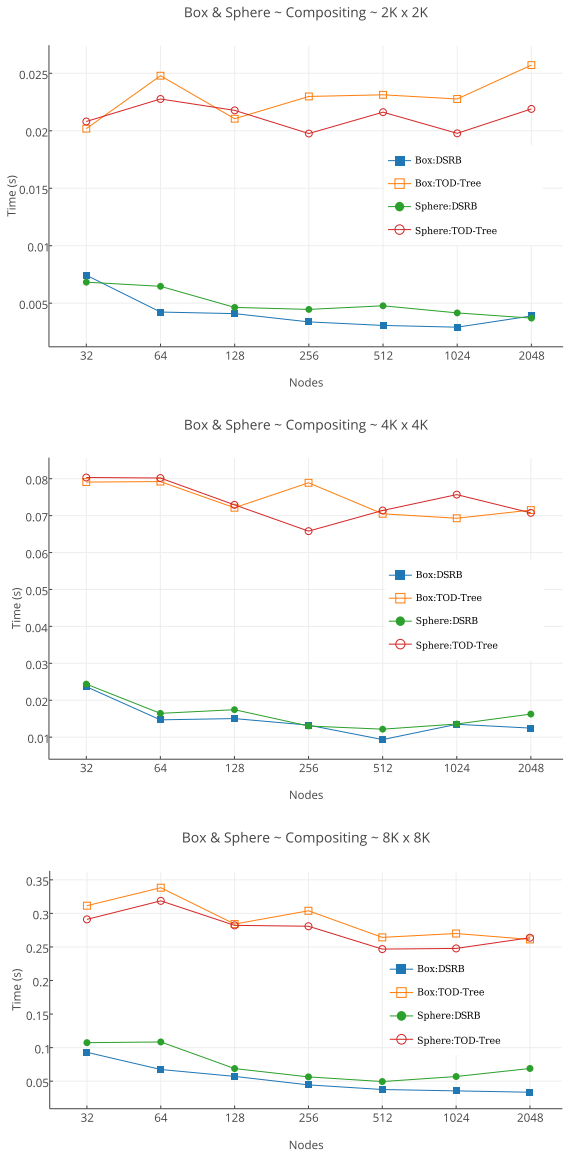

Figure 9: *Scaling of the artificial box and sphere datasets on Edison - showing compositing only.*

5. Conclusion and Future Work

In this paper, we have introduced an image compositing algorithm that has both spatial and temporal awareness of compositing. Spatial awareness ensures that no compositing processes will ever receive data for a region to which it does not contribute, thereby minimizing communication. Temporal awareness ensures that processes do not try to communicate with processes that are still rendering, thereby minimizing delays. Combining spatial and temporal awareness streamlines compositing by allowing several regions of an image to be fully composited fairly quickly. Compositing is delayed only for data-intensive regions of an image. This gives us a substantial gain compared to TOD-Tree, which lacks spatial and temporal awareness. The DSRB algorithm can also beneficial in situ visualization scenarios where the domain decomposition is dictated by the simulation.

As future work, we would like to run the scheduler as a thread on one of the compositing nodes instead of on a separate node. Also, we would like to try to find a way to estimate the time it takes to render on each node and see how this approach can be used to reduce communication. More complex visualization workloads involving polygons, glyphs, and mixed non-volumetric data may require a more sophisticated scheduler, perhaps employing directed graphs instead of chains. Also, we would like to run the DSRB algorithm on a GPU-accelerated supercomputer where the rendering times are likely to be shorter than on a CPU-only accelerated supercomputer.

One of the limitations of the DSRB algorithm is where the load is perfectly balanced. Having the extra communication with the scheduler will decrease the performance of DSRB algorithm. Also, as the number of nodes we use to render increases, there will be a point at which each node will finish rendering, even with lighting and imbalance in workload, nearly at the same time, and the differences in rendering completion time will become negligible. We would like to run experiments on large supercomputers to determine when this will happen for various data and image sizes. This will help establish the architecture dependent crossover point at which we should switch over to algorithms, such as TOD-Tree and Radix-k, which minimize communication. Another limitation is the depth-orderable requirement described in Section 3. While DSRB performs well for block-structured decompositions, including block-structured AMR grids, unstructured grids are not guaranteed to be depth-orderable due to the potential of concave regions. This could be overcome with some limited data replication and would be interesting future research.

6. Acknowledgments

The authors would like to thank the National Energy Research Scientific Computing Center (NERSC) for providing access to the Edison supercomputer and the support staff at NERSC for helping resolve compilation issues.

This research was partially supported by the Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002375, the DOE SciDAC Institute of Scalable Data Management Analysis and Visualization DOE DE-SC0007446, NASA NSSC-NNX16AB93A and NSF ACI-1339881, NSF IIS-1162013.

References

[EMP09] EILEMANN S., MAKHINYA M., PAJAROLA R.: Equalizer: A Scalable Parallel Rendering Framework. *IEEE Transactions on Visualization and Computer Graphics 15*, 3 (May 2009), 436–452. URL: http://dx.doi.org/10.1109/TVCG.2008.104, doi:10.1109/TVCG.2008.104. 3

[EP07] EILEMANN S., PAJAROLA R.: Direct send compositing for parallel sort-last rendering. In *Proceedings of the 7th Eurographics Conference on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2007), EGPGV '07, Eurographics Association, pp. 29–36. URL: http://dx.doi.org/10.2312/EGPGV/EGPGV07/029-036, doi:10.2312/EGPGV/EGPGV07/029-036. 3

[FCS*10] FOGAL T., CHILDS H., SHANKAR S., KRÜGER J., BERGERON R. D., HATCHER P.: Large data visualization on distributed memory multi-gpu clusters. In *Proceedings of the Conference on High Performance Graphics* (Aire-la-Ville, Switzerland, Switzerland, 2010), HPG '10, Eurographics Association, pp. 57–66. URL: http://dl.acm.org/citation.cfm?id=1921479.1921489. 2, 3

[FE11] FREY S., ERTL T.: Load Balancing Utilizing Data Redundancy in Distributed Volume Rendering. In *Eurographics Symposium on Parallel Graphics and Visualization* (2011), Kuhlen T., Pajarola R., Zhou K., (Eds.), The Eurographics Association. doi:10.2312/EGPGV/EGPGV11/051-060. 3

[FSZ*10] FANG W., SUN G., ZHENG P., HE T., CHEN G.: *Network and Parallel Computing: IFIP International Conference, NPC 2010, Zhengzhou, China, September 13-15, 2010. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, ch. Efficient Pipelining Parallel Methods for Image Compositing in Sort-Last Rendering, pp. 289–298. URL: http://dx.doi.org/10.1007/978-3-642-15672-4_25, doi:10.1007/978-3-642-15672-4_25. 3

[GPC*15] GROSSET A. V. P., PRASAD M., CHRISTENSEN C., KNOLL A., HANSEN C.: Tod-tree: Task-overlapped direct send tree image compositing for hybrid mpi parallelism. In *Proceedings of the 15th Eurographics Symposium on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2015), PGV '15, Eurographics Association, pp. 67–76. URL: http://dx.doi.org/10.2312/pgv.20151157, doi:10.2312/pgv.20151157. 1, 3, 7

[HBC10] HOWISON M., BETHEL E. W., CHILDS H.: MPI-hybrid Parallelism for Volume Rendering on Large, Multi-core Systems. In *Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2010), EG PGV'10, Eurographics Association, pp. 1–10. URL: http://dx.doi.org/10.2312/EGPGV/EGPGV10/001-010, doi:10.2312/EGPGV/EGPGV10/001-010. 2

[HBC12] HOWISON M., BETHEL E., CHILDS H.: Hybrid Parallelism for Volume Rendering on Large-, Multi-, and Many-Core Systems. *Visualization and Computer Graphics, IEEE Transactions on 18*, 1 (Jan 2012), 17–29. doi:10.1109/TVCG.2011.24. 3

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., FRANK R., AHERN S., KIRCHNER P. D., KLOSOWSKI J. T.: Chromium: A Stream-processing Framework for Interactive Rendering on Clusters. *ACM Trans. Graph. 21*, 3 (July 2002), 693–702. URL: http://doi.acm.org/10.1145/566654.566639, doi:10.1145/566654.566639. 3

[Hsu93] HSU W. M.: Segmented Ray Casting for Data Parallel Volume Rendering. In *Proceedings of the 1993 Symposium on Parallel Rendering* (New York, NY, USA, 1993), PRS '93, ACM, pp. 7–14. URL: http://doi.acm.org/10.1145/166181.166182, doi:10.1145/166181.166182. 2

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *Computer Graphics and Applications, IEEE 14*, 4 (1994), 23–32. doi:10.1109/38.291528. 1

[MKPH11] MORELAND K., KENDALL W., PETERKA T., HUANG J.: An Image Compositing Solution at Scale. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), SC '11, ACM, pp. 25:1–25:10. URL: http://doi.acm.org/10.1145/2063384.2063417, doi:10.1145/2063384.2063417. 3

[MMD06] MARCHESIN S., MONGENET C., DISCHLER J.-M.: Dynamic Load Balancing for Parallel Volume Rendering. In *Eurographics Symposium on Parallel Graphics and Visualization* (2006), Heirich A., Raffin B., dos Santos L. P., (Eds.), The Eurographics Association. doi:10.2312/EGPGV/EGPGV06/043-050. 2, 3

[Mor11] MORELAND K.: *IceT Users' Guide and Reference*. Tech. rep., Sandia National Lab, January 2011. 3, 7

[MPHK93] MA K.-L., PAINTER J., HANSEN C., KROGH M.: A data distributed, parallel algorithm for ray-traced volume rendering. In *Parallel Rendering Symposium, 1993* (1993), pp. 15–22, 105. doi:10.1109/PRS.1993.586080. 1, 2

[MSE07] MÜLLER C., STRENGERT M., ERTL T.: Adaptive load balancing for raycasting of non-uniformly bricked volumes. *Parallel Comput. 33*, 6 (June 2007), 406–419. URL: http://dx.doi.org/10.1016/j.parco.2006.12.002, doi:10.1016/j.parco.2006.12.002. 3

[MWMS07] MOLONEY B., WEISKOPF D., MÖLLER T., STRENGERT M.: Scalable sort-first parallel direct volume rendering with dynamic load balancing. In *Proceedings of the 7th Eurographics Conference on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2007), EGPGV '07, Eurographics Association, pp. 45–52. URL: http://dx.doi.org/10.2312/EGPGV/EGPGV07/045-052, doi:10.2312/EGPGV/EGPGV07/045-052. 3

[MWP01] MORELAND K., WYLIE B. N., PAVLAKOS C. J.: Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (2001), Breen D. E., Heirich A., Koning A. H. J., (Eds.), IEEE, pp. 85–92. URL: http://dblp.uni-trier.de/db/conf/pvg/pvg2001.html#MorelandWP01. 6

[Neu94] NEUMANN U.: Communication Costs for Parallel Volume-Rendering Algorithms. *IEEE Comput. Graph. Appl. 14*, 4 (July 1994), 49–58. URL: http://dx.doi.org/10.1109/38.291531, doi:10.1109/38.291531. 2

[PGR*09] PETERKA T., GOODELL D., ROSS R., SHEN H.-W., THAKUR R.: A Configurable Algorithm for Parallel Image-compositing Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), SC '09, ACM, pp. 4:1–4:10. URL: http://doi.acm.org/10.1145/1654059.1654064, doi:10.1145/1654059.1654064. 1, 3

[SML*03] STOMPEL A., MA K.-L., LUM E. B., AHRENS J., PATCHETT J.: Slic: Scheduled linear image compositing for parallel volume rendering. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (Washington, DC, USA, 2003), PVG '03, IEEE Computer Society, pp. 6–. URL: http://dx.doi.org/10.1109/PVGS.2003.1249040, doi:10.1109/PVGS.2003.1249040. 3

[SMW*04] STRENGERT M., MAGALLÃŞN M., WEISKOPF D., GUTHE S., ERTL T.: Hierarchical Visualization and Compression of Large Volume Datasets Using GPU Clusters. In *Eurographics Workshop on Parallel Graphics and Visualization* (2004), Bartz D., Raffin B., Shen H.-W., (Eds.), The Eurographics Association. doi:10.2312/EGPGV/EGPGV04/041-048. 3

[YWG*10] YU H., WANG C., GROUT R. W., CHEN J. H., MA K.-L.: In Situ Visualization for Large-Scale Combustion Simulations. *IEEE Comput. Graph. Appl. 30*, 3 (May 2010), 45–57. URL: http://dx.doi.org/10.1109/MCG.2010.55, doi:10.1109/MCG.2010.55. 2

[YWM08] YU H., WANG C., MA K.-L.: Massively Parallel Volume Rendering Using 2-3 Swap Image Compositing. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Piscataway, NJ, USA, 2008), SC '08, IEEE Press, pp. 48:1–48:11. URL: http://dl.acm.org/citation.cfm?id=1413370.1413419. 3