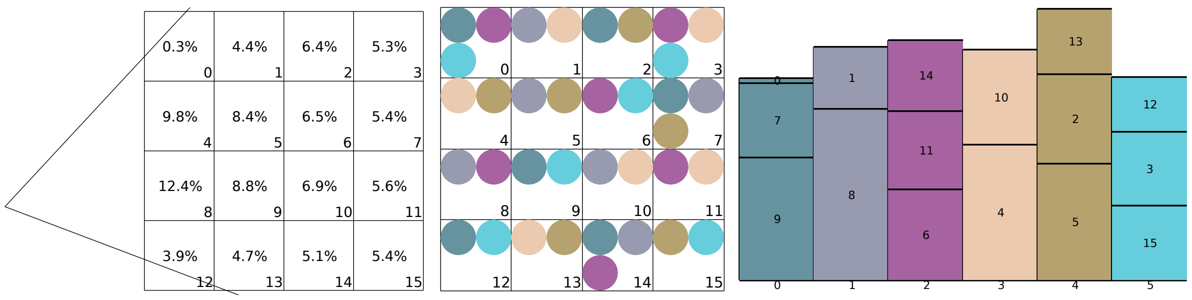
Eurographics Symposium on Parallel Graphics and Visualization (2011)
T. Kuhlen, R. Pajarola, and K. Zhou (Editors)

Load Balancing Utilizing Data Redundancy in Distributed Volume Rendering

S. Frey and T. Ertl

Visualisierungsinstitut der Universität Stuttgart, Germany



(a) Number of ray samples per brick relative to the total amount for a given view frustum (b) Distribution of bricks to devices (different colors) (c) Workload balanced over devices

Figure 1: *The cost of volume bricks highly depends on parameters that are commonly adjusted interactively at runtime, most notably the camera position and orientation (Fig. 1(a)). In order to prevent significant delays caused by data transfers, we distribute bricks to compute devices redundantly (Fig. 1(b)). This can be utilized by our scheduler to evenly distribute the load across all devices by shifting brick render tasks between devices without costly data transfers (Fig. 1(c)).*

Abstract

In interactive volume rendering, the cost for rendering a certain block of the volume strongly varies with dynamically changing parameters (most notably the camera position and orientation). In distributed environments – wherein each compute device renders one block – this potentially causes severe load-imbalance. Balancing the load usually induces costly data transfers causing critical rendering delays. In cases in which the sum of memory of all devices substantially exceeds the size of the data set, transfers can be reduced by storing data redundantly. We propose to partition the volume into many equally sized bricks and redundantly save them on different compute devices with the goal of being able to achieve evenly balanced load without any data transfers. The bricks assigned to a device are widely scattered throughout the volume. This minimizes the dependency on the view parameters, as the distribution of relatively cheap and expensive bricks stays roughly the same for most camera configurations. This again enables our fast and simple scheduler to evenly balance the load in almost any situation. In scenarios in which only very few bricks constitute the majority of the overall cost a brick can also be partitioned further and rendered by multiple devices.

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

1. Introduction

A great number of areas like medicine, material sciences, computational physics, and various other disciplines have to deal with large volumetric data sets. For visually analyzing this data, interactive visualization is demanded that does not compromise quality. Parallel volume rendering is one of the

DOI: 10.2312/EGPGV/EGPGV11/051-060

most efficient techniques to achieve this goal by distributing the rendering process over a cluster of machines. Particularly when dealing with large data sets, the volume data set is partitioned and distributed to render nodes. Splitting the data in object space this way avoids the need for costly out-of-core techniques. However, a major issue arises from the fact that the rendering costs for the volume block that is assigned to a computation device changes significantly with parameters like the camera position and orientation that are typically adjusted interactively by the user. In order to balance the load, the rendering of certain volume parts needs to be moved from one device to another. Common dynamic load-balancing schemes require time consuming data transfers to achieve evenly balanced computation load. Even if this only induces memory copies from main to device memory, it can still cause significant rendering delays.

At the same time, the amount of nodes in a standard cluster as well as the available graphics card memory are steadily increasing. For a wide range of areas and applications, the total available amount of graphics memory would allow to store the data set several times. However, common object-space distribution techniques usually save the whole data set in memory only once and maximally save data in boundary regions redundantly. This largely wastes potential flexibility for load-balancing.

We propose to split the volume in many small volume blocks called bricks and distribute them redundantly to compute devices. In each frame, for every brick a device is chosen for rendering that holds the respective brick in its memory. A good distribution of bricks already allows for balanced execution times and due to the fact that typically multiple bricks are rendered per device, there is no need for the bricks to be equally expensive.

In particular, we make the following contributions:

- Concept of redundantly distributing many small, equally sized bricks instead of allocating one brick per device.
- Load-balancing technique exploiting the brick redundancy to maximum effect.
- Brick distribution algorithm to allow for good load-balancing under all circumstances
- Evaluation of the influence of camera position and orientation on the brick rendering cost

The remainder of this paper is structured as follows. Sec. 2 discusses related work in distributed volume rendering and scheduling. Sec. 3 gives an overview on our approach, while Sec. 4 and Sec. 5 discuss in detail its two phases, namely the a priori initialization and the per frame load-balancing. We show the effectiveness of our approach in Sec. 6.

2. Related Work

Distributed Volume Rendering

Distributed volume rendering has been investigated for a long period of time and a magnitude of publications can be found on this issue. Most of the existing systems fit either into the sort-first or sort-last category according to Molnar et al.'s classification [MCEF94]. Our approach is in the sort-last category, i.e. the data is split between the nodes, and each node renders its own portion. Compositing then takes depth information into account to form a final image from each node's rendering. Sort-last volume rendering techniques are able to handle very large datasets as demonstrated by Wylie et al. [WPLM01] by statically distributing these datasets among the nodes. The predominant hierarchical compositing schemes that are used in sort-last architectures aiming at rendering large data sets are the Direct Send approach by Neumann [Neu93] and the Binary-Swap algorithm [MPHK93]. Palmer et al. [PTT97] discussed how to efficiently exploit all levels of the deep memory hierarchy of a cluster system. Using the clusters that compute the simulation also for volume rendering has also been investigated [PYRM08]. While the first techniques for parallel volume rendering employed slice-based rendering [MHE01], more recent systems use GPU-based raycasting [KW03] with a single rendering pass [SSKE05]. A simple back-to-front raycaster in the CUDA SDK [NVI08] demonstrates the implementation with a modern GPGPU language.

There has been a lot of work in recent years on data structures that can be used to address dynamic load balancing issues in distributed volume rendering systems. For instance, Wang et al. [WGS04] proposed a hierarchical space-filling curve for that purpose. Lee et al. [LSH05] employ a hybrid/BSP tree subdivision. Müller et al. [MSE06] and Marchesin et al. [MMD06] amongst others employ a kd-tree in order to dynamically reorganise the data distribution in a cluster. Marchisin et al. [MMD06] also showed that when zooming on parts of the data sets, load imbalance becomes a challenging issue. In order to achieve good load balancing they dynamically distribute the data (i.e. they resize the volume bricks) among the rendering nodes according to the load of the previous frame.

Similar to our approach, Peterka et al. [PRY*08] generate more volume bricks than there are devices. However, they assign every brick to only device in initialization using a round-robin scheme and no dynamic balancing takes place during rendering.

Frank and Kaufman [FK09] use data dependency information to automate and improve load balanced volume distribution and ray-task scheduling. A directed acyclic graph of bricks is employed and a cost function is evaluated to create a load balanced network distribution. The output is a render-node assignment which minimizes the total run time.

Scheduling

In the context of this work, scheduling refers to the way how work packages (i.e. rendering a part of the volume) are assigned to compute devices. Basic research in scheduling

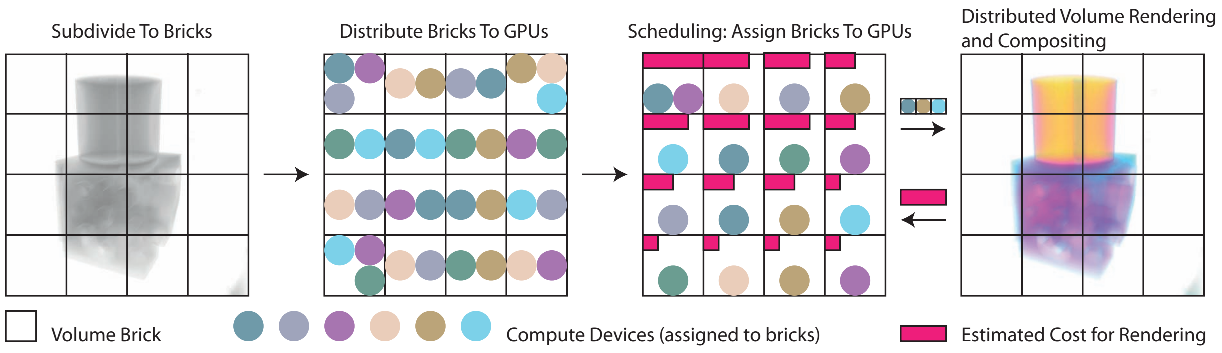

Subdivide To Bricks
Distribute Bricks To GPUs
Scheduling: Assign Bricks To GPUs
Distributed Volume Rendering and Compositing
Volume Brick
Compute Devices (assigned to bricks)
Estimated Cost for Rendering

Figure 2: *Overview of the basic steps of our approach. The volume subdivision into bricks and the brick-device distribution are executed before the interactive volume rendering pass. In that pass, the scheduler determines for every frame which device should be used to render a specific (sub-)brick prior to the actual raycasting and compositing.*

for parallel applications has been done by Diekmann [Die98] amongst others. In particular, he emphasized the amount of required communication as an important quality criteria for a schedule. Müller et al. [MFS*09] presented CUDASA, a general CUDA development environment supporting cluster environments. It features a GPU-accelerated scheduling mechanism that is aware of data locality. Frey and Ertl [FE10] proposed a framework for developing parallel applications for single host or ad-hoc compute network environments. It features a scheduler that is based on the critical path method that determines prior to the actual computation which implementation to execute on which device to minimize the overall runtime by considering device speed, availability and transfer cost.

Teresco et al. [TFF05] worked on a distributed system in which every CPU requests tasks from the scheduler which are sized according to the device's measured performance score. Resource-aware distributed scheduling strategies for large-scale grid/cluster systems were also proposed by Viswanathan et al. [VVR07]. Zhou et al. [ZHR*09] introduced a multi-GPU scheduling technique based on work stealing to support scalable rendering. In particular, they emphasize the importance of avoiding data transfers whenever possible. In order to allow a seamless integration of load-balancing techniques into an application, generic object-oriented load-balancing libraries and frameworks have also been developed [DHB*00] [SZ02].

3. Overview

The basic procedure of our approach is illustrated in Fig. 2. First of all, the volume is subdivided into equally sized bricks. The brick size is influenced by several factors like device memory or device architecture (e.g. for graphics cards there must be enough work to do in parallel for good device occupancy), the number of devices etc. . Subsequently, the bricks are distributed redundantly to the available devices. These steps are denoted as initialization steps in the following and discussed in detail in Sec. 4.

During the interactive volume rendering procedure, a schedule is created for every device before actually rendering a frame. The basic input for the scheduling procedure are the render time estimates for every brick and the brick-device distribution. The estimated rendering costs are simply taken from the previous frame, assuming good frame coherence concerning render time costs. In Sec. 6 we will show that this assumption is valid. The scheduling process is run locally on every host node with the same input data yielding the same results. This avoids transferring the schedule to all nodes, but naturally requires the timing results to be broadcasted to all nodes after rendering. However, transmitting the timing results can be done in parallel to the rendering and compositing computations. The scheduling and load-balancing procedure is explained in detail in Sec. 5.

4. Initialization

The initialization phase encompasses the steps that are executed once prior to the interactive volume raycasting and compositing phase. The initialization phase basically consists of the determination of how bricks are distributed across compute devices as well as the according transfers of the bricks to the devices.

Good brick distribution is important to enable good load-balancing properties. Optimally, each device has bricks whose render costs are on all levels from cheap to expensive for all camera positions and orientations. Badly distributed blocks can severely hinder balancing, because it forces few devices to deal with a large share of the overall load. This is discussed in more detail later in Sec. 5 by means of Fig. 4.

In order to achieve a good brick distribution, we uniformly distribute the bricks assigned to a device across the whole volume. This leads to a wide load diversity for all camera

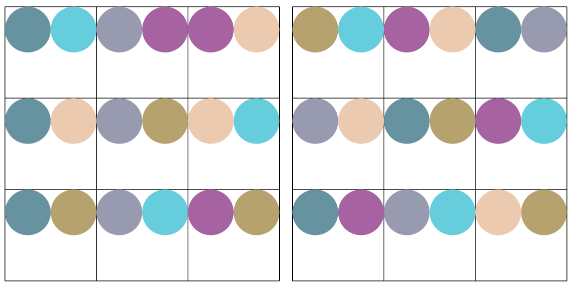

(a) Initial Random Distribution (b) Optimized Distribution

Figure 3: *Initialization of bricks (squares) with devices (colored circles). The optimized version is reorganized such that the distance of the bricks belonging to one device increases.*

parameter settings in the rendering process. We define the criteria for a good brick distribution as follows:

(a) Each device holds the maximum amount of bricks depending on its memory capacity.
(b) All bricks should be distributed about the same number of times whenever possible.
(c) Any two devices do not share a large amount of associated bricks.
(d) The minimum distance of the bricks assigned to a device is as big as possible.

The process to approach such a distribution is split into two steps as illustrated in Fig. 3. First, the bricks are distributed randomly, taking care of conditions (a), (b) and (c). Second, the brick distribution is optimized regarding condition (d) by swapping bricks between devices.. We will elaborate on these steps in detail in the following.

Brick Distribution Initialization

The goal of this step is to determine a fair initial distribution of bricks to devices. In this context fair means that initially every brick has the same chances to be assigned to any device and that it is attempted to distribute bricks equally often.

First of all, the list of the devices and the list of bricks is shuffled randomly. Then, a brick and a device index are used to iterate over the respective lists. In every iteration step, it is attempted to add the current brick to the current device. If it was successful (i.e. the brick has not already been assigned to the device previously), both indices are incremented. In the case of an index reaching the end of a list, the respective list is shuffled and the index is set to the beginning of that list.

If a brick cannot be added to device, the index of the brick stays the same and only the device index is incremented. When the device index reaches its original index again (i.e. the brick could not be assigned to any device), the brick is deleted from the list. Note that while a brick is in such a pending state, the device vector is not shuffled.

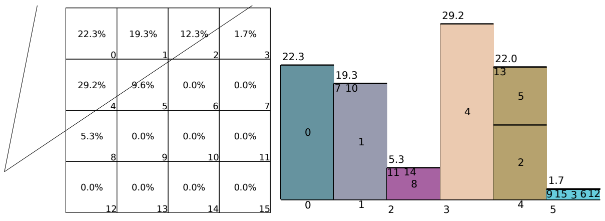
```
shuffle(devices)
shuffle(bricks)
b=0
d=0
while(!devices.empty() && !bricks.empty() ) {
  // brick is inserted if it is not present already
  inserted = deviceBricks[devices[d]].insert(bricks[b])
  if(inserted) {
    b++
    if(devices[d].size() == devices[d].capacity()) {
      // device is full -> delete it from list
      devices.erase(d)
      d--
    }
  }
  else {
    // brick insertion pending if d != -1
    if(d == attemptingInsertSince) {
      // delete brick,
      // it cannot be inserted anywhere anymore
      bricks.erase(b)
    }
    else if(d == -1)
    {
      // start a new attempt to add the brick
      // to the next possible device
      attemptingInsertSince = d
    }
  }
  d++
  if(d == devices.size()) {
    d=0
    if(attemptingInsertSince == -1) {
      // shuffle devices if there is
      // no pending brick insertion attempt
      shuffle(devices)
    }
  }
  if(b == bricks.size()) {
    b=0
    shuffle(bricks)
  }
}
```

Listing 1: *Pseudo-code for initializing bricks with devices.*

Devices are erased from the device list when they reach their maximum brick reception capacity. The initialization is complete when either the brick list or the device list is empty. The procedure is outlined in more detail by means of pseudo-code in Listing 1.

Brick Distribution Optimization

In the second step, devices swap bricks to optimize the brick distribution. The basic procedure of this step bears


(a) No Job Splitting

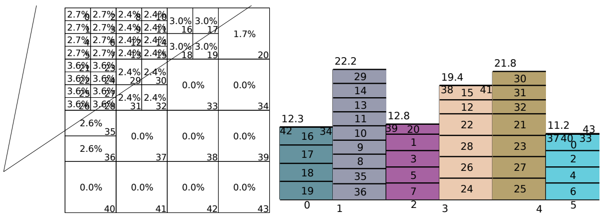

(b) With Job Splitting

Figure 4: *Splitting jobs improves the balancing of load across devices with certain camera configurations. Note that the assignment of bricks to devices is the same as in Fig. 1.*

some similarity with the point swapping procedure employed by Balzer et al. [BSD09] for optimizing point clusters. For each pair of devices, the most beneficial pair of bricks to swap between them is determined and subsequently exchanged if the status quo is improved by that. A swap is only valid when there are no brick duplicates on a device as a result.

The quality of the distribution of bricks $B(d)$ for a device d is measured by the quality function q:

$$q(d) = \sum_{b0 \in B(d)} \sum_{b1 \in B(d)} \sqrt{|b0 - b1|} \quad (1)$$

Using the square root of all brick pairs as quality measure is useful for our purpose as – in opposite to squared distances – many small uniform distances lead to a much higher value than only one far away brick. This means that the maximization of the measure q leads to a wide and largely uniform brick distribution. The optimization step ends when all possible device pairs have been looked at without inducing a swap.

5. Load-Balancing

After the initialization step, every device has a number of bricks assigned to it that it can render potentially. For every frame, it is determined which device to use for rendering a certain brick. This process is split into two steps: the subdivision of the bricks to render jobs and the scheduling of the jobs such that the overall execution time is minimized. The job generation process bears some similarity to traditional load balancing in parallel volume rendering as it has been discussed in Sec. 2. However, in contrast to these techniques, the amount of partitions is not bound to the amount of devices. In the end, this makes the (brick) partitioning easier because it is sufficient to distribute the load of the brick only approximately to jobs and then balance the load by smartly assigning multiple jobs to devices.

5.1. Job Generation

Originally, every brick rendering task translates into one job. A good balancing of load between devices can be achieved

```
while(!stable) {
  stable = true
  for d0 in devices {
    for d1 in devices {
      baseQuality = quality(bricksAssignedTo(d0))
                  + quality(bricksAssignedTo(d1))
      bestQuality = baseQuality

      // swap all possible brick pairs and
      // measure quality
      for b0 in bricksAssignedTo(d0) {
        for b1 in bricksAssignedTo(d1) {
          tmp0 = bricksAssignedTo(d0)
          tmp0.erase(b0)
          if(!tmp0.insert(b1))
            // b1 is already in assigned to d0
            continue
          tmp1 = bricksAssignedTo(d1)
          tmp1.erase(b1)
          if(!tmp1.insert(b0))
            // b0 is already in assigned to d1
            continue
          // determine most beneficial brick swap
          if(quality(tmp0) + quality(tmp1) > baseQuality) {
            bestQuality = quality(tmp0) + quality(tmp1)
            bestSwap = (b0,b1)
          }
        }
      }
      // swap most beneficial pair of bricks
      // if its better than the status quo
      if(bestQuality > baseQuality) {
        bricksAssignedTo(d0).erase(bestSwap[0])
        bricksAssignedTo(d1).erase(bestSwap[1])
        bricksAssignedTo(d0).insert(bestSwap[1])
        bricksAssignedTo(d1).insert(bestSwap[0])
        stable = false
      }
    }
  }
}
```

Listing 2: *Pseudo-code for optimizing brick assignments.*

that way when the major rendering load is distributed across many bricks (like in Fig. 1 for instance). However, when the rendering of only one or very few bricks constitutes the major share of the overall cost, the bricks need to be split to allow for equal load distribution (Fig. 4).

This is accomplished by looping over all jobs from the previous frame and splitting them recursively until each job's estimated rendering cost exceeds a certain value a:

$$a = \max\left(\frac{\sum_{j \in Jobs} c(j)}{|D| \cdot b^2}, r\right) \tag{2}$$

$c(j)$	The anticipated cost of a job.
$\lvert D\rvert$	The number of devices.
b	The maximal amount of redundant copies of any brick.
r	Lower bound for job render time.

The lower bound r for a is determined experimentally – we used 5 ms for the measurements in the context of this work. It is motivated from the fact that parallel devices in general and GPUs in particular (our targeted compute device in this work) need a certain amount of work (i.e. degree of parallelism) to work efficiently. In extreme cases, low GPU occupancy could lead to a scenario in which each of two newly generated jobs takes as long as the original job would have. The maximum amount of redundant copies b is used to express the number of options the scheduler has to choose a device for a certain job. In particular if there are many options for a scheduler, numerous smaller instead of few big jobs are useful because this enables the scheduler to distribute the load more finely. In contrast to that, if a job can only be assigned to one specific device, it makes no sense to split it and cause unnecessary execution overhead.

Before rendering the very first frame, each brick is initialized with its own binary tree, only consisting of the anchor node at first. The leaves of this tree represent the jobs that are associated with the respective brick. The job cost estimates are set to the same, arbitrary value due to the lack of real rendering timing results initially. Starting from the second frame, every leaf of the tree is assigned the rendering time estimated for it. In our case, this is simply the rendering cost that its associated job had in the previous frame. In Section 6, we will show that this assumption is valid at least for our application scenario. Based on the estimated costs for its leaves, the tree of each brick is modified per frame in two consecutive phases: the splitting and the merging phase.

In the splitting phase, if the cost estimate for a leaf exceeds a, the volume block associated with it is split in half and two child nodes are created representing the two new volume blocks. Each of the children is assigned half of the estimated cost of its parent node. Splitting is performed axis aligned such that the resulting jobs or leaves are equally big (i.e. their associated volume blocks have the same size). The split axis is chosen such that the length of the diagonal of the volume blocks attached to the jobs is minimal. This achieves a good trade-off between resulting image size that needs to be transferred for scheduling (smaller image-space footprint is better) and the amount of casted rays for good occupancy (larger image-space footprint is better). When all edge lengths are equal, we split along the axis that is most aligned with the view direction.

When the camera position or focus changes, rendering costs change as well and previously split jobs might need to be merged again in order to avoid unnecessary overhead. To this end we iterate over all sibling pairs of jobs (they were split from the same job in a previous frame) and determine whether the sum of their cost estimates still meets the splitting criteria. If this is not the case, they are merged back again to one job, which means that they are removed from the tree and their total weight is assigned to the parent node. This procedure is executed recursively for all leaf nodes until the sums of all leaf siblings exceed a.

5.2. Job Scheduling

The task of a scheduler is to decide in every frame what jobs a device needs to render such that every brick is rendered exactly once and the maximum load occurring on any device in minimal. Formally, this is an optimization problem that is closely related to the class of packing problems. It can be defined as follows:

Given: A set of devices D, a set of jobs J and a function $c : D \times J \rightarrow \mathbb{R} \cup \infty$.

Find: A surjective assignment function $a : D \rightarrow J$ such that $\max_{d \in D}(\sum_{j \in a(d)} c(d, j))$ is minimal.

Note that $c(d, j) = \infty$ if the brick belonging to the job $j \in J$ is not present on device $d \in D$. Alternatively, depending on the application, it can be useful to use a sufficiently large constant C instead (i.e. such that C is safely larger than any possible device fill level). This would trigger the additional assignment of bricks in cases in which a brick has not been assigned to any device initially.

As our scheduler needs to be run for every frame and thus needs to be fast, we do not attempt to solve this complex problem optimally. Instead, we opted for a quick and simple approach that still delivers good results. It bears some similarities to the best-fit decreasing heuristic that can be used for the bin packing problem and basically works as follows:

1. Sort all jobs in order of their weight in descending order
2. Iterate through the job list and assign each job to a device such that the overall estimated execution time of all jobs of a device across all devices is minimal.

The overall complexity of the algorithm is $O(n \log n + nd)$. While sorting is in $O(n \log n)$ with n being the number of jobs, the job assignment procedure is in $O(nd)$, with d denoting the number of devices assigned to a brick. A complete example on the output produced by the scheduler for a specific input is provided in Fig. 1 and Fig. 4.

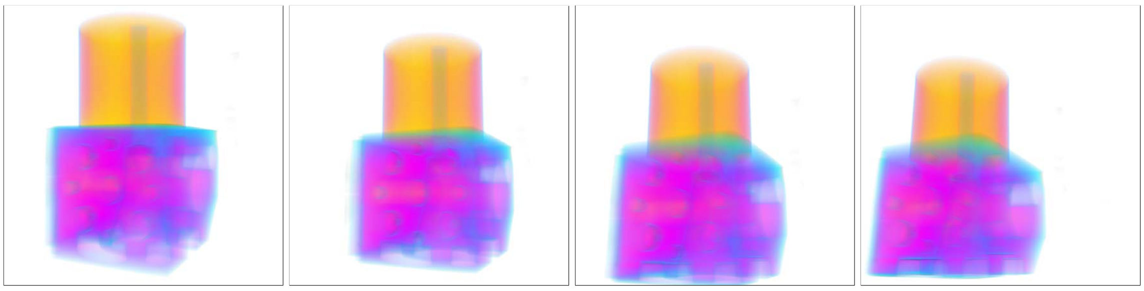


Figure 6: *Renderings from the first four camera and focus positions in our camera path.*

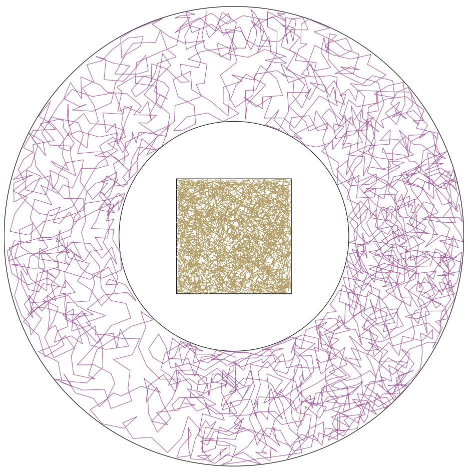


Figure 5: *Camera (purple) and the focal point (khaki) movement in a 2D example with 2000 time steps.*

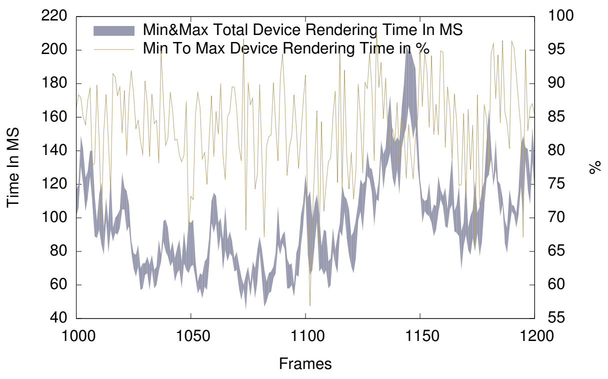


Figure 7: *Change of the rendering times on the camera path per frame. The gray curve shows both the average minimal and maximal render time of a device along the camera path. The khaki curve explicitly depicts the min-max ratio.*

6. Results

For the evaluation of our approach, we employed a plain vanilla volume raycaster and a simple compositer written in CUDA. We used a cluster featuring eight nodes connected via Gigabit ethernet. Each node is equipped with a NVIDIA GTX285 (featuring 1 GB of graphics memory) and a Quad-Core AMD Opteron Processor with 2.3 Ghz. We further used a 1024^3 data set volume data set with 16 bit accuracy, whose size is 2 GB accordingly and rendered 1024^2 images.

Regarding the choice of the brick size, there is a trade-off between load-balancing flexibility and overhead for rendering, compositing as well as data transfers. In our scenario, a brick size of 352^3 causes only marginal overhead compared to common sort-last rendering while at the same time the bricks can be distributed well enough to enable good load-balancing. This brick size leads to a total amount of $3^3 = 27$ bricks and enables each device to save 10 bricks.

Camera Path

The camera and its focal point move along a predefined camera path in a series of 5000 frames (see Fig. 5 for a 2D example). The direction and the step size were chosen randomly from one time step to another with some restrictions. The camera must not exceed a certain distance to the center of the volume or enter the volume and the focal point must not leave the volume. Furthermore, the maximal step size length is determined as a fraction of the diagonal of the volume.

In order to compute a step, a candidate set of 5 possible steps is generated randomly. Which step to take from the current position to the next position is chosen from the candidate set such that the distance of the next position to any other previous position is maximal, whereas steps violating the restrictions that were described above are simply ignored. If there are no valid steps, a new candidate set is generated. This enforces that eventually all possible areas for the camera and focus positions are sampled densely enough to deliver a well-balanced overall result. The first four rendered frames on the camera path are displayed in Fig. 6. The influence of the different camera configurations is depicted in Fig. 7. It can be seen that with each step the rendering cost

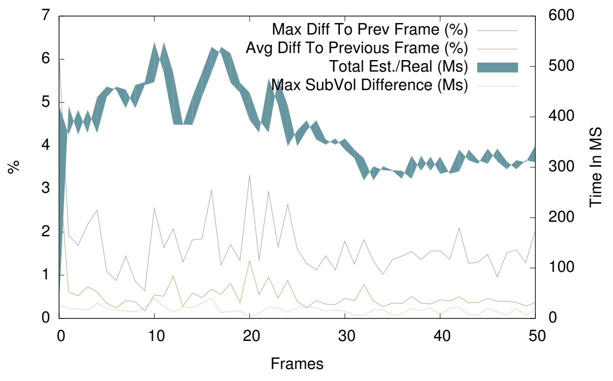


Figure 8: *Comparison between estimated render time (render time from previous frame) and real execution time for the first 50 frames of our camera path. The gray and the khaki curve express the difference relative to the previous values on a percentage basis. While the khaki curve compares the total execution time of all jobs, the gray curve displays the maximum difference between any two jobs. The cyan curve gives both the total estimated and total measured rendering time while the orange curve depicts the maximal time difference between two jobs.*

might change significantly. The magnitude of the changes might even be less favorable in terms of frame coherency than the average interactive volume rendering session. Still, as we will show in the following, the render times of the previous frame are a good indication for the current frame.

Brick Rendering Predictions

It is critical for our application that there are good predictions on how expensive jobs are relative to each other for the upcoming frame. To that end we simply use the render time from the previous frame. Figure 8 shows that the assumption of coherent render times is valid for our scenario as the relative differences are within the range of a few percent. Obviously, if there were really large camera leaps between two frames, the render times from the previous frame would not be very expressive. However, these leaps usually do not occur in systems with stable, interactive frame rates.

Scaling

The scaling behaviour of our approach with an increasing amount of devices is plotted in Fig. 9. It shows that the maximal render time per device profits significantly from an increase in device count. Its scaling factor is even slightly larger than one. For instance, the longest time any device needs to render its bricks is 122.9 ms with four nodes and 55.5 ms with eight nodes. The reason for that is that with more devices more memory becomes available and the level of redundancy rises. The increasing brick redundancy is also

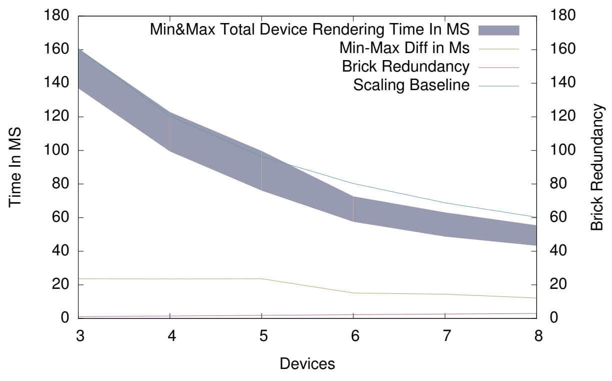


Figure 9: *Scaling with the amount of compute devices. The gray curve shows both the average minimal and maximal render time of a device along the camera path. The khaki curve explicitly depicts the difference. The purple curve shows the increasing redundancy factor, meaning that bricks are distributed to more devices (see Eq. 3). The cyan curve shows the baseline scaling curve based on the performance with three nodes.*

plotted in Fig. 9. The redundancy factor is defined as

$$\frac{\sum_{d \in D} b(d)}{|B|} \tag{3}$$

with D being the set of devices, $|B|$ the number of bricks and $b(d)$ the amount of bricks stored on a device $d \in D$. More redundancy enables the scheduler to find a more optimal device-brick assignment. It also helps to decrease the maximal load imbalance (i.e. the difference between the maximum and minimum time taken for rendering by a device).

Brick Distribution Variations

Our approach consists of a set of optimizations that can be enabled or disabled to study their benefit. It can be seen from Table 1 and Fig. 10 that the improvement of using a certain optimization heavily depends on the amount of devices that are involved in the computation. The keywords in the figure and the table stand for the following optimization variants:

Standard The normal approach with every feature presented in the paper.

No Redundancy Each brick is assigned one device only.

Brick Cluster Bricks belonging to a device are not distributed across the volume but concentrated in one area. This is computed by using the inverse quality function in the optimization step of the initial brick distribution (see Sec. 4). In combination with *No Redundancy*, this approximates the common one-brick-per-device strategy.

No Job Split Splitting of jobs into smaller jobs is disabled. This means that a brick translates into exactly one job.

The table shows the aforementioned effect that the technique for the standard variant scales a little better than lin-

Variants	8 Dev.	4 Dev.
Standard	55.5	122.9
Brick Cluster	58.6	140.8
No Redundancy	74.6	122.6
Brick Cluster & No Redundancy	85.2	141.5
No Job Split	75.2	123.2
Brick Cluster & No Job Split	68.2	142.2

Table 1: *Averaged maximum render times in milliseconds for different variants of our approach along our camera path for four and eight devices respectively. For a different brick size of* 256^3 *and eight nodes, the "Standard" rendering time was measured to be relatively slow with 73 ms, but it performed almost equally well compared to rendering with* 352^3 *in the "No Redundancy" case.*

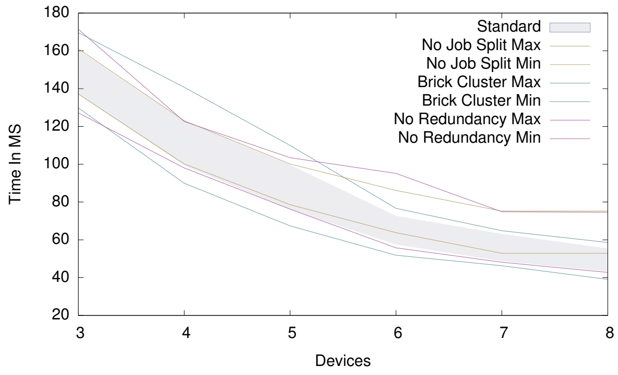


Figure 10: *Comparison of the normal version to variants with one optimization switched off respectively.*

early due to an increased level of redundancy. Furthermore, it can be seen that the negative effect of brick clustering is worse with few than with many nodes. The reason for that is again that due to the lower amount of redundancy, the scheduler has less possibilities to soften the negative effects. This becomes clear when considering the variant with clustered bricks and no redundancy. While the impact is only minor for setups with few devices (as there is no high level of redundancy to begin with), the required rendering time for 8 nodes is significantly higher. This is also true to a smaller extent when considering the no redundancy case only. Additionally, due to the higher amount of scheduling possibilities, disabling job splitting also has a much higher impact with large number of devices. Remember that the reason behind job splitting is to allow for a more fine distribution of jobs belonging to an expensive bricks. However, when a brick is only distributed once or twice, there is not much room for a scheduler to widely distribute the expensive job.

Using a smaller brick size of 256^3 and thus increasing the amount of bricks to 64 leads to a significant slow-down for our standard technique due the induced overhead. However, when disabling brick redundancy, both brick sizes lead to similar performance because the larger amount of bricks allows for a better static load distribution.

General System Timings

The overall execution time of the whole volume rendering process is largely dominated by the volumetric raycasting performance. Compositing at least 27 images (depending on the amount of job splits) takes approximately 10 ms in total with our naive compositor written in CUDA. Note that sending the render image resulting from one job to the compositing node is largely done in parallel to the execution of another job and thus does not have a significant performance impact in our testing scenario. Distributing the job render times is a n-to-n operation (every node needs to send its job render time to all other nodes), but its size is only a few bytes and it can be done in parallel with compositing. Accordingly, this does not contribute significantly to the overall execution time either, at least not in our small test cluster system. Finally, our simple scheduler delivers very high performance and takes significantly less than a millisecond to run in our scenario, even with a large number of jobs and devices.

7. Conclusion

We proposed a parallel volume rendering approach to utilize data redundancy in order to achieve good load-balancing with no data transfers required. In particular, we introduced a volume brick distribution procedure as well as a job generation and scheduling technique which are efficient and easy to implement. We showed the effectiveness of our approach in a small cluster environment, in which we also evaluated the dependence of brick rendering cost on camera parameters. The generality of our approach makes it flexible enough to be combined with basically any volume rendering acceleration technique that is suitable for parallel volume rendering.

The approaches presented in this paper are also able to handle heterogeneous environments efficiently, like different graphics cards that vary in speed and memory. We also expect the system to scale pretty well with larger cluster systems. For future work, we plan to investigate these aspects in detail. Furthermore, this work almost exclusively focuses on the volume raycasting part and largely neglects the performance impact induced by compositing. In particular we expect this impact to grow for larger scale cluster systems, which is why we would like to take compositing timing effects into consideration in more detail in our evaluation using state-of-the-art techniques from that area (e.g. by Makhinya et al. [MEP10]). Finally, we aim to integrate a more elaborate scheduler, that is able to compute a more optimal solution, but still preserves the good complexity and execution time properties of our current scheduler.

References

[BSD09] Balzer M., Schlömer T., Deussen O.: Capacity-constrained point distributions: A variant of Lloyd's method. *ACM Transactions on Graphics (Proceedings of SIGGRAPH) 28*, 3 (2009), 86:1–8. 5

[DHB*00] Devine K., Hendrickson B., Boman E., John M. S., Vaughan C.: Design of dynamic load-balancing tools for parallel applications. In *Proc. Intl. Conf. on Supercomputing* (Santa Fe, New Mexico, 2000), pp. 110–118. 3

[Die98] Diekmann R.: *Load Balancing Strategies for Data Parallel Applications*. PhD thesis, UniversitÃď't Paderborn, 1998. 3

[FE10] Frey S., Ertl T.: PaTraCo: A Framework Enabling the Transparent and Efficient Programming of Heterogeneous Compute Networks. Ahrens J., Debattista K., Pajarola R., (Eds.), Eurographics Association, pp. 131–140. 3

[FK09] Frank S., Kaufman A.: Dependency graph approach to load balancing distributed volume visualization. *The Visual Computer 25*, 4 (2009), 325–337. 2

[KW03] Krüger J., Westermann R.: Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of IEEE Visualization '03* (2003), pp. 287–292. 2

[LSH05] Lee W.-J., Srini V., Han T.-D.: Adaptive and Scalable Load Balancing Scheme for Sort-Last Parallel Volume Rendering on GPU Clusters. In *International Workshop on Volume Graphics* (2005). 2

[MCEF94] Molnar S., Cox M., Ellsworth D., Fuchs H.: A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications 14*, 4 (1994), 23–32. 2

[MEP10] Makhinya M., Eilemann S., Pajarola R.: Fast Compositing for Cluster-Parallel Rendering. Ahrens J., Debattista K., Pajarola R., (Eds.), Eurographics Association, pp. 111–120. 9

[MFS*09] Müller C., Frey S., Strengert M., Dachsbacher C., Ertl T.: A compute unified system architecture for graphics clusters incorporating data locality. *IEEE Transactions on Visualization and Computer Graphics 15*, 4 (2009), 605–617. 3

[MHE01] Magallòn M., Hopf M., Ertl T.: Parallel volume rendering using PC graphics hardware. In *Pacific Conference on Computer Graphics and Applications* (2001), pp. 384–389. 2

[MMD06] Marchesin S., Mongenet C., Dischler J.-M.: Dynamic Load Balancing for Parallel Volume Rendering. In *Eurographics Symposium on Parallel Graphics and Visualization* (2006), Eurographics Association, pp. 51–58. 2

[MPHK93] Ma K. L., Painter J. S., Hansen C. D., Krogh M. F.: A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering. In *PRS '93: Proceedings of the 1993 Symposium on Parallel Rendering* (1993), pp. 15–22. 2

[MSE06] Müller C., Strengert M., Ertl T.: Optimized Volume Raycasting for Graphics-Hardware-based Cluster Systems. In *Eurographics Symposium on Parallel Graphics and Visualization* (2006), Eurographics Association, pp. 59–66. 2

[Neu93] Neumann U.: Parallel Volume-Rendering Algorithm Performance on Mesh-Connected Multicomputers. In *PRS '93: Proceedings of the 1993 Symposium on Parallel Rendering* (1993), pp. 97–104. 2

[NVI08] NVIDIA: NVIDIA CUDA SDK code samples. http://developer.nvidia.com/object/cuda.html, 2008. 2

[PRY*08] Peterka T., Ross R., Yu H., Ma K.-L., Kendall W., Huang J.: Assessing improvements in the parallel volume rendering pipeline at large scale. In *Proceedings of SC 08 Ultrascale Visualization Workshop* (Austin TX, 2008). 2

[PTT97] Palmer M. E., Taylor S., Totty B.: Exploiting deep parallel memory hierarchies for ray casting volume rendering. In *Proceedings of the IEEE symposium on Parallel rendering* (New York, NY, USA, 1997), PRS '97, ACM, pp. 15–ff. 2

[PYRM08] Peterka T., Yu H., Ross R., Ma K.-L.: Parallel volume rendering on the ibm blue gene/p. In *Proc. Eurographics Parallel Graphics and Visualization Symposium 2008* (2008), pp. 73–80. 2

[SSKE05] Stegmaier S., Strengert M., Klein T., Ertl T.: A Simple and Flexible Volume Rendering Framework for Graphics-Hardware–based Raycasting. In *Proceedings of the International Workshop on Volume Graphics '05* (2005), pp. 187–195. 2

[SZ02] Stankovic N., Zhang K.: A distributed parallel programming framework. *IEEE Trans. Softw. Eng. 28*, 5 (2002), 478–493. 3

[TFF05] Teresco J. D., Faik J., Flaherty J. E.: Resource-aware scientific computation on a heterogeneous cluster. *Computing in Science and Engineering 7*, 2 (2005), 40–50. 3

[VVR07] Viswanathan S., Veeravalli B., Robertazzi T. G.: Resource-aware distributed scheduling strategies for large-scale computational cluster/grid systems. *IEEE Trans. Parallel Distrib. Syst. 18*, 10 (2007), 1450–1461. 3

[WGS04] Wang C., Gao J., Shen H.-W.: Parallel Multiresolution Volume Rendering of Large Data Sets with Error-Guided Load Balancing. In *Eurographics Symposium on Parallel Graphics and Visualization* (2004), pp. 23–30. 2

[WPLM01] Wylie B., Pavlakos C., Lewis V., Moreland K.: Scalable rendering on pc clusters. *IEEE Comput. Graph. Appl. 21* (July 2001), 62–70. 2

[ZHR*09] Zhou K., Hou Q., Ren Z., Gong M., Sun X., Guo B.: Renderants: interactive reyes rendering on gpus. In *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers* (New York, NY, USA, 2009), ACM, pp. 155:1–11. 3