PAPER *Special Section on Parallel and Distributed Computing and Networking*

Memory Efficient Load Balancing for Distributed Large-Scale Volume Rendering Using a Two-Layered Group Structure

Marcus WALLDEN[†a)], **Stefano MARKIDIS**[††], *Nonmembers*, **Masao OKITA**[†], *and* **Fumihiko INO**[†], *Members*

SUMMARY We propose a novel compositing pipeline and a dynamic load balancing technique for volume rendering which utilizes a two-layered group structure to achieve effective and scalable load balancing. The technique enables each process to render data from non-contiguous regions of the volume with minimal impact on the total render time. We demonstrate the effectiveness of the proposed technique by performing a set of experiments on a modern GPU cluster. The experiments show that using the technique results in up to a 35.7% lower worst-case memory usage as compared to a dynamic *k*-d tree load balancing technique, whilst simultaneously achieving similar or higher render performance. The proposed technique was also able to lower the amount of transferred data during the load balancing stage by up to 72.2%. The technique has the potential to be used in many scenarios where other dynamic load balancing techniques have proved to be inadequate, such as during large-scale visualization.

key words: *large-scale visualization, distributed computing, load balancing, GPU*

1. Introduction

The capabilities of modern supercomputers enable the simulation and visualization of large-sized data sets with high precision and detail. Generated data sets, often multivariate or spanning multiple time steps, can consist of terabytes of data. Using a sorting scheme called sort-last [1], the data can be partitioned and distributed among available processes. The distributed data volumes can then be visualized in parallel by utilizing ray-casting based volume rendering [2]. Partial images from all processes then need to be composed based on their position and distance from the camera in the volume [3].

The render time can vary between processes based on many factors, e.g., used optimization techniques or the characteristics of the data set. If there are any substantial render time imbalances, dynamic load balancing techniques can be used to effectively reduce the total render time during in-situ visualization or post-hoc exploration. However, dynamically redistributing data can result in large memory imbalances between processes. For large data sets some processes might run out of memory, making many dynamic load balancing techniques unsuitable for large-scale visualization [4], [5].

Commonly used dynamic load balancing techniques are based on tree structures, e.g., a *k*-d tree [6]. In the *k*-d tree structure, the original volume is represented by the root of the tree, as illustrated in Fig. 1. For each new depth in the tree, the volume is split in two on either the x, y or z-axis. The two resulting volume *blocks* are then separately held in two child branches. Each process that participates in the rendering stage is given ownership of a branch (and all of its child branches) in one of the levels of the tree. For example, if 2, 4 or 8 processes are used, each process is given ownership of a branch on depth 1, 2 or 3, respectively. Data can be load balanced between children of the same branch in the tree, as shown in Fig. 1. The load balanced data consists of a slice of blocks that border both branches, ensuring that each process still only holds a contiguous and convex partition of data in object space after the load balancing has been completed [7]. Utilizing this structure and ensuring that each process only renders contiguous data results in two positive aspects:

1. **A simple composition order for partial images rendered by each process.** The *k*-d tree structure enables processes to compose all partial images generated by its blocks during the rendering stage, without any external communication. As such, only a single partial image from each process need to be composed in an inter-process compositing stage. In this stage the remaining partial images are composed to create an image of the full volume.
2. **A low scheduling complexity.** The strict *k*-d tree load balancing scheme limits between which processes load

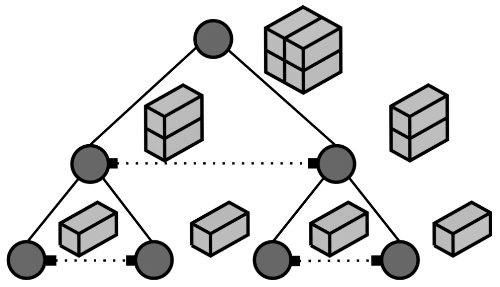


Fig. 1 *k*-d tree data distribution and load balancing. Circles represent branches in the tree, whereas blocks are represented by cuboids, each of which results in a partial image when rendered. Branches that can load balance are connected via dotted lines.

Manuscript received January 7, 2019.
Manuscript revised May 10, 2019.
Manuscript publicized September 9, 2019.

[†]The authors are with the Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871 Japan.
[††]The author is with the School of Electrical Engineering and Computer Science, KTH Royal University of Technology, Stockholm, 114 28 Sweden.
a) E-mail: ma-wa@ist.osaka-u.ac.jp

DOI: 10.1587/transinf.2019PAP0003

Copyright © 2019 The Institute of Electronics, Information and Communication Engineers

balancing can take place. This limitation significantly simplifies the load balancing algorithm.

If data needs to be transferred between two processes that have ownership of branches on opposite sides of the tree structure, it is impossible to transfer the data directly between them. Instead, load balancing has to be performed multiple times between the upper branches of the tree, meaning that many processes have to participate in the load balancing stage. The upper branches of the tree are responsible for larger regions of the volume; the amount of data that is transferred increases by 100% in each level. This does not only result in many redundant data transfers, it also means that using the *k*-d tree structure can lead to a significant memory load imbalance [8]. This behavior should scale in relation to the number of processes, meaning that it could be of a greater concern during large-scale visualization. An example of a *k*-d tree memory imbalance is shown in Fig. 2, where the main computational load is focused on the upper quadrant of the volume. Equalizing the render times also results in one process holding a substantial part of the volume in memory.

The worst-case memory usage of a single process when using the *k*-d tree structure is $O(v)$, where v is the number of voxels in the volume. The risk that a high data imbalance occurs limits the use of *k*-d tree based dynamic load balancing in large-scale applications, where even small data imbalances can result in some processes running out of memory [5].

There is a need for a dynamic load balancing technique that does not adhere to the existing limitations of hierarchical tree structures. We propose a technique for distributed volume rendering of rectilinear grids by which processes can render data from non-contiguous regions of the volume, contrary to *k*-d tree techniques. By having a non-hierarchical, less restrictive structure, it would be possible to prioritize load balancing blocks with high render times. This would lead to a lower worst-case memory usage and less redundant data transfers. However, rendering data from non-contiguous regions is not without its drawbacks. In a naive implementation, it would not be possible to compose partial images on each process during the rendering stage. Partial images, each representing a single block, would instead have to be composed during the inter-process compositing stage, greatly increasing the total compositing time.

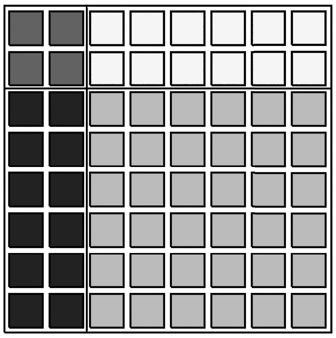


Fig. 2 2D representation of a *k*-d tree block distribution between four processes, in a worst-case scenario where the main computational load is focused on the top left quadrant of the volume. Volume blocks are represented as squares, whereas the color distinguishes different processes.

The contribution of this paper is a novel compositing pipeline and load balancing technique which utilizes a scalable non-hierarchical group structure to effectively allow a single process to render data from non-contiguous regions of the volume. Through this technique we also enable the use of custom load balancing schemes, meaning that the algorithm effectively can be tailored according to the needs and constraints of the researcher. The main goal of the two-layered *group* technique is to resolve existing limitations of tree-based hierarchical structures, thus reducing the worst-case process memory usage, without negatively affecting the total render time. A secondary goal is to reduce the amount of redundant data transfers, which unnecessarily burdens I/O functionality. We demonstrate the effectiveness of the group technique as compared to a *k*-d tree technique as well as a static distribution by conducting a series of experiments on a GPU cluster using up to 32 processes, each of which has a dedicated GPU.

The structure of this paper is organized as follows. Related work is presented in Sect. 2. The compositing pipeline and the load balancing technique are described in Sect. 3. The technique is then evaluated in Sect. 4. Lastly, our conclusions are presented in Sect. 5.

2. Related Work

k-d trees and similar tree structures have been used extensively in many related works to achieve dynamic load balancing [7]–[11]. The render time of the previous frame is often used as a load balancing heuristic [7], [9]. Commonly, uniform-sized blocks are stored in the *k*-d tree [7], [11]. Other works have also explored using non-uniform blocks in order to achieve finer granularity [9]. However, this would require extensive preprocessing and that the volume is static. Others have utilized machine learning and performance modeling as a load balancing heuristic, though still using a *k*-d tree structure [11].

Zhang *et al.* [8] proposed a constrained *k*-d tree structure to achieve dynamic load balancing for parallel particle tracing. They strove to achieve a balanced particle load by redistributing particles among processes based on a *k*-d tree structure. However, they recognized that particles can be condensed in a small region of the volume, thus requiring some processes to hold large sections of the volume in memory. In order to sidestep this issue they introduced constrains on the *k*-d tree data partitioning, thus limiting the number of voxels held by each process. Although this approach ensures that processes can hold their respective regions of the volume in memory, it fails to guarantee an even distribution of particles due to the used constraints. Furthermore, the authors note that the only way to ensure an optimal distribution is to have the full volume in memory on each process [8].

Utilizing dynamic load balancing techniques tends to result in an uneven data distribution among processes. In large-scale applications the data sets could consist of multiple terabytes of data, meaning that even small-scale data transfers can be time consuming and result in some processes exceeding their available amount of memory. As such, many large-scale visualization projects have utilized static techniques [5], [12] or have limited load balancing to equalizing the data distribution, rather than explicitly lowering the total render time [10], [13].

Dorier *et al.* [13] developed a technique which in-situ can identify important data of a simulation and reduce less important data based on a time limit constraint. As such, the amount of blocks in full resolution vary between each frame, leading to an interesting load balancing challenge. Rendering and compositing were both performed by using Catalyst, Paraview's in-situ library [14]. Blocks were randomly distributed among processes each time step to achieve an even data distribution and to lower the total render time. In total 16,000 blocks were randomly distributed, leading to 16,000 partial images that needed to be composed. Since blocks on each process were from random regions of the volume there was no way to compose the images locally during the rendering stage, leading to a resource- and time-demanding inter-process compositing stage.

3. Two-Layered Dynamic Load Balancing Technique

To lower the memory usage and redundant data transfers as compared to *k*-d tree techniques, we propose a load balancing technique with a non-hierarchical structure by which processes can render blocks from non-contiguous regions of the volume. Rendering non-contiguous blocks can lead to a complicated irregular compositing order, as noted in Sect. 2. We introduce a two-layered group structure and a compositing pipeline to lower the complexity of the compositing stage. In this section we provide an overview of the technique, followed by detailed information about all included functionality: how the two-layered group structure is formed, how efficient load balancing is accomplished and how the compositing pipeline simplifies the compositing stage. Lastly, the memory usage of the group technique is analyzed.

3.1 Overview of the Technique

We coin the terms *full sets*, a number of sets which contain the initial static collection of contiguous blocks delegated to a process, and *working sets*, the sets of contiguous blocks being rendered by a process during a specific frame. All processes are responsible for two operations: (1) rendering all blocks present in its working sets and (2) compositing all partial images of blocks in its full sets.

Figure 3 depicts the execution flow of the presented technique, whereas Fig. 4 shows an example where a block is load balanced to another process. As illustrated in Fig. 4 (b), the rendered partial images from load balanced blocks are returned asynchronously to the original owner during the rendering stage. Each process can as such compose partial images from blocks in its own full sets (Fig. 4 (c)), leading to a correct compositing order even if processes are rendering blocks from different regions of the volume. Utilizing this compositing pipeline means that only a single partial image from each process need to be composed during a final inter-process compositing step (Fig. 4 (d)). To summarize, two distinct compositing steps are required in the group technique: one to compose partial images of blocks in each process' full sets and one to compose the resulting image from each process.

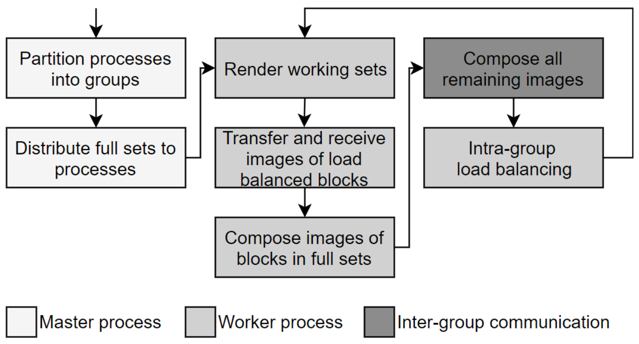

Partition processes into groups
Distribute full sets to processes
Render working sets
Transfer and receive images of load balanced blocks
Compose images of blocks in full sets
Compose all remaining images
Intra-group load balancing
Master process
Worker process
Inter-group communication

Fig. 3 Program execution flow of the two-layered group technique. Initially, processes are partitioned into groups and the full sets are distributed to processes. The rendering stage and an initial compositing step are then performed by each process. The remaining images are composed in an inter-group fashion in a second compositing step. Lastly, load balancing is performed within each group before the next frame can be rendered.

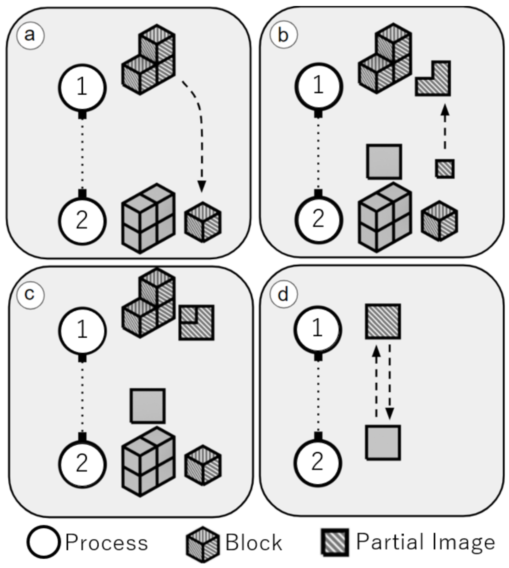


Fig. 4 Example of how a load balanced block is rendered and composed. (a) A block is load balanced between two processes in the same group. The load balanced block makes up a new working set on the receiving process. (b) Each process renders their working sets, starting with blocks belonging to other processes. Images of working sets not present in a process' full sets are returned to the original owner asynchronously during the rendering stage. (c) Each process composes images from blocks in their full sets. (d) Inter-group image compositing takes place to compose the final image.

3.2 Two-Layered Group Structure

Instinctively there are two scalability-related concerns coupled to the group technique:

- Finding an adequate load balance for a large number of processes is time consuming.
- The introduced first compositing step can be time consuming if many processes are involved.

These factors can result in excessive communication and time consuming computations if many processes are utilized. To improve the scalability of the technique, processes are distributed into one or more distinct and autonomous groups and limited to load balancing with processes within the same group. Load balancing and the first compositing step (Fig. 4 (a)–(c)) have in such a scenario no inter-group dependence, meaning they can be performed in parallel within each group. By limiting the amount of processes that can interact we lower the communication and algorithm complexity, thus effectively eliminating many scalability-related concerns.

We define a group as a non-empty static set of processes, whereas each process is a member of a single group. Processes are partitioned into groups in a round-robin fashion at the start of the program, which ensures that the blocks held by the processes in each group are not concentrated in the same region of the volume. By scaling the number of groups relative to the number of processes we can ensure that the amount of processes in each group remains constant. An example group structure is displayed in Fig. 5.

3.3 Intra-Group Load Balancing

Load balancing can be performed between any pair of processes within the same group. Although this approach is more flexible than *k*-d tree techniques, it also means that the load balancing scheme is NP-Hard if no limitations are set. To lower the load balancing time complexity we utilize a greedy load balancing algorithm to determine between which processes load balancing takes place and what data is transferred. An example of the data distribution using the group technique in the scenario presented in Fig. 2 is shown in Fig. 6.

We utilize $f = 4$ full sets on each process, created by splitting the initial contiguous collection of blocks in half on the y- and z-axes. Slices of blocks can be load balanced from both the positive and negative direction on the x-axis, as illustrated in Fig. 7. However, we limit load balancing to a single process at a time in each direction on the x-axis for each full set. For example, if process 1 load balances a slice of blocks from the positive direction on the x-axis of the first full set to process 2, no other process can receive blocks from the set's positive direction until process 2 has returned all load balanced blocks to process 1. This means that a process is simultaneously only able to delegate blocks to $2f$ other processes. Limiting load balancing to a single process in each direction for each full set has one key benefit: if a pair of processes consecutively performs load balancing, all transferred blocks are from a contiguous region in object space. They can as such be put in the same working set on the receiving process, resulting in a single partial image which asynchronously can be transferred back before

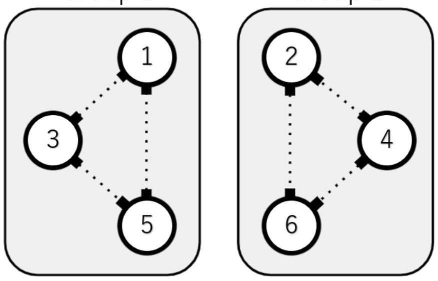


Fig. 5 An example structure containing two groups. Processes are partitioned into groups in a round-robin fashion. Processes within the same group can freely perform load balancing amongst each other, as illustrated by the dotted lines.

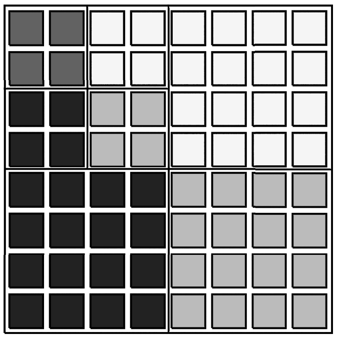


Fig. 6 2D representation of a block distribution between four processes using the two-layered group technique, in a worst-case scenario where the main computational load is focused on the top left quadrant of the volume. Volume blocks are represented as squares, whereas the color distinguishes different processes.

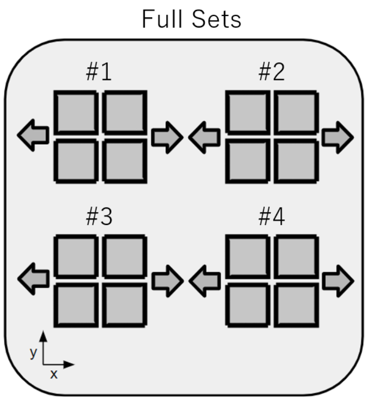


Fig. 7 Slices of blocks can be load balanced from both directions of the x-axis. However, load balancing for each full set can only be performed with a single process in each direction. Having four full sets on each process means that blocks can be delegated to eight other processes simultaneously.

Algorithm 1: Intra-group load balancing algorithm executed after each rendered frame

```
Input :
    $L$: List of processes that have a lower-than-average render time.    ▷ List is sorted based on lowest render time
    $H$: List of processes that have a higher-than-average render time.    ▷ List is sorted based on highest render time
    $S = \{s_1, \cdots, s_n\}$: Dictionary of type {process, set of processes from which the process has been delegated blocks}.
    $E = \{e_1, \cdots, e_n\}$: Dictionary of type {process, set of processes that the process has delegated blocks to}.
    $T = \{t_1, \cdots, t_n\}$: List containing the total render time of each process.

1  $B = \{b_1, \cdots, b_n\}$    ▷ List containing all processes that have not performed load balancing this frame
2  $\forall t \in L \cap B$ : if $e_t <> \phi$ then
3      set $p$ where $p \in e_t \cap B$ such that $t_p \geq t_i, \forall i \in e_t \cap B$    ▷ $p$ has the highest render time in $e_t \cap B$
4      recall load balanced slice from $p$ to $t$
5      $B = B \backslash \{t, p\}$    ▷ Exclude $t$ and $p$ from other load balancing events
6      if |load balanced blocks from t to p|= 0 then
7          $e_t = e_t \backslash \{p\}, s_p = s_p \backslash \{t\}$
8  $\forall t \in H \cap B$ : if $s_t <> \phi$ then
9      set $p$ where $p \in s_t \cap B$ such that $t_p \leq t_i, \forall i \in s_t \cap B$    ▷ $p$ has the lowest render time in $s_t \cap B$
10     recall load balanced slice from $t$ to $p$
11     $B = B \backslash \{t, p\}$    ▷ Exclude $t$ and $p$ from other load balancing events
12     if |load balanced blocks from p to t|= 0 then
13         $s_t = s_t \backslash \{p\}, e_p = e_p \backslash \{t\}$
14 $\forall t \in L \cap B$ : if $H \cap s_t <> \phi$ then
15     set $p$ where $p \in H \cap s_t \cap B$ such that $t_p \geq t_i, \forall i \in H \cap s_t \cap B$    ▷ $p$ has the highest render time in $H \cap s_t \cap B$
16     load balance slice from $p$ to $t$
17     $B = B \backslash \{t, p\}$    ▷ Exclude $t$ and $p$ from other load balancing events
18 $\forall t \in L \cap B$ : if $H <> \phi$ then
19     set $p$ where $p \in H \cap B$ such that $t_p \geq t_i, \forall i \in H \cap B$    ▷ $p$ has the highest render time in $H \cap B$
20     load balance slice from $p$ to $t$
21     $B = B \backslash \{t, p\}$    ▷ Exclude $t$ and $p$ from other load balancing events
22     $s_t = s_t \cup \{p\}, e_p = e_p \cup \{t\}$
```

the end of the rendering stage.

After each frame the average render times are calculated in each group. Processes of which the render time deviates from the average are sorted into one of two lists depending on if the render time is lower or higher. Historical data transfers and current process render times are then used to dictate which processes in the two lists perform load balancing. This functionality helps reduce unnecessary spread of blocks, resulting in fewer image transfers. The pseudo code of the load balancing algorithm is shown in Algorithm 1. Once a process has performed load balancing it is excluded from subsequent load balancing events during the same frame to limit the amount of data that can be transferred before the next rendering stage.

The goal of the load balancing algorithm is to equalize the render time among all processes. However, it also strives to minimize the spread of blocks to lower the amount of communication and data transfers during the compositing stage. For this purpose, in the first operation of Algorithm 1, each process that has a lower-than-average render time recalls a previously load balanced slice of blocks, if possible. Similarly, in the second operation each process with a higher-than-average render time attempts to return a load balanced slice of blocks to another process. These operations ensure that a process never delegates blocks to other processes whilst simultaneously rendering blocks it does not own. If such operations are not possible, each process with a lower-than-average render time attempts to perform load balancing with processes from which it already has been delegated blocks. Transferred blocks can be put in an already existing working set, meaning that there is no extra overhead during the first compositing step. However, for this operation to be possible it requires that at least one such process has a higher-than-average render time. Finally, if none of the previous operations are possible, each process with a lower-than-average render time is delegated a slice of blocks from a process with a higher-than-average render time, which then has to be put into a new working set. Processes with lower-than-average and higher-than-average render times are iterated starting with the lowest and highest render times, respectively. As such, the process with the lowest render time performs load balancing with the process that has the highest render time.

All four operations have a clear block transfer order. In the first and second operations, slices of blocks are returned in a LIFO (last in first out) order between all pairs of blocks to ensure that working sets only render contiguous data. In the third operation, a slice of blocks is taken from the same full set and direction as previously transferred blocks between the two processes. For the fourth operation, the slice of blocks is taken from the full set on the sending process with the highest render time that currently has delegated blocks to fewer than two other processes. A slice of blocks is then load balanced from the positive direction of the x-axis, or the negative direction in case another process already has been delegated blocks from the positive direction.

3.4 Image Compositing Pipeline

As described in Sect. 3.1, two compositing steps are required when using the group technique. In the first step all processes compose images of blocks in their full sets. This operation is strictly performed within each group, and involves partial images from all load balanced sets being transferred back to the owning process. This procedure is similar to direct send compositing [15], which normally would induce a communication complexity of $O(n^2)$, where n is the number of processes [16]. However, since the process only is performed within each group the worst-case communication complexity will be less than $O(n^2)$ as long as the number of groups is scaled in proportion to the number of processes. Furthermore, this compositing step can be asynchronously performed during the rendering stage, resulting in minimal time overhead.

In the second compositing step all remaining partial images are composed to form the final image which represents the full volume. It is as such performed in an intergroup fashion, and is identical to a k-d tree's or static technique's compositing stage. Compositing methods such as binary swap [3] or DSH [17] could be used in the second compositing step due to their low communication complexities.

We maintain the same resolution for all images generated and used in the two compositing steps. However, we note that utilizing various compression strategies [18] or variable image sizes could lower the compositing time; especially in the first compositing step where the partial images in many cases only portray a small subset of the volume.

3.5 Process Memory Usage

Given a volume of v voxels, if the group technique is used each process has to keep its full sets in memory, resulting in a memory usage of v/n, where n is the number of processes. Furthermore, in the worst case scenario, a specific process' full sets consist of blocks with near-zero render times. The process is then delegated blocks so that its render time matches that of the rest of the processes in the group. The absolute worst case memory usage is as such v/g, where g is the number of groups. However, as described in Sect. 3.3, the load balancing algorithm prioritizes load balancing blocks with high render times, ensuring that the process memory usage will remain lower than $2v/n$ other than in extreme scenarios.

4. Experimental Evaluation

In this section we evaluate the group technique by comparing it to a k-d tree technique as well as a static distribution. Load balancing, data transfers and compositing can potentially be performed asynchronously during the rendering stage depending on the used rendering pipeline. As such we chose to evaluate the process render time performance, the process memory usage, the amount of transferred data as well as the effect of utilizing multiple groups separately to provide a broader overview that is not tied to a specific rendering pipeline. We also provide a separate overview of the computation times for all stages of the pipeline.

4.1 Experiment Description

We performed a series of tests on a GPU cluster using 8, 16 and 32 processes. Each node was equipped with an Intel Xeon E5-1650 v4 CPU, 128GB of memory and two Nvidia GeForce GTX 1080 GPUs. Nodes were interconnected via EDR InfiniBand and used GCC version 7.3.0, CUDA version 9.2 [20] and Open MPI version 3.1.0 [21]. Up to two MPI processes were run on each node, each of which were allocated a dedicated GPU.

We chose to rotate the camera 360 degrees around the y-axis to measure the render time and memory usage of each process at different viewing angles. The image resolution was set to 1024^2, which commonly is used for this type of testing [22]. The used test case and image resolution should represent an average-case scenario for the examined load balancing techniques. To test our technique in a wide range of scenarios we used three different data sets, each of which have their own unique characteristics.

The first data set is a computed tomography (CT) scan of a porcine heart [19], shown in Fig. 8 (a). The second data set is of a Richtmyer-Meshkov instability simulation [19], displayed in Fig. 8 (b). The third data set is a CT scan of a Spathorhynchus fossorium [19], shown in Fig. 8 (c).

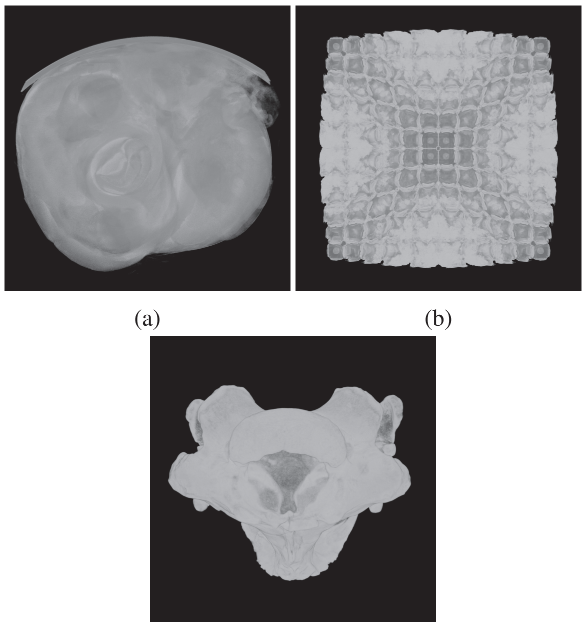


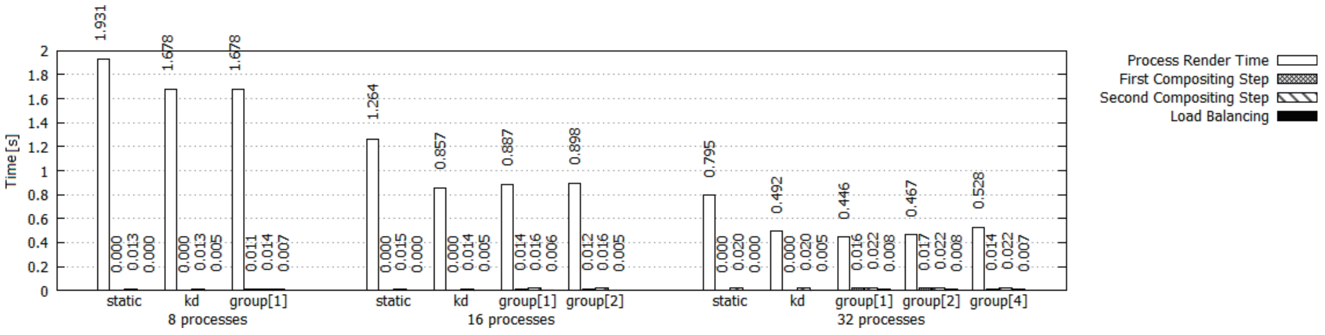
Fig. 8 Data sets used for evaluation: (a) a CT scan of a porcine heart, (b) a simulation of the Richtmyer-Meshkov instability and (c) a CT scan of a Spathorhynchus fossorium [19].

The three data sets consist of 2048 × 2048 × 2612 voxels (43.8GB), 2048 × 2048 × 1920 voxels (32.2GB) and 1024 × 1024 × 750 voxels (6.3GB), respectively. The third data set is substantially smaller than the other two and can be visualized on a single machine using modern hardware. As such, the computation times of the compositing and load balancing stages will constitute a higher percentage of the total render time as compared to the other two data sets. However, it is still of interest to evaluate the achieved load balance and the amount of transferred data.

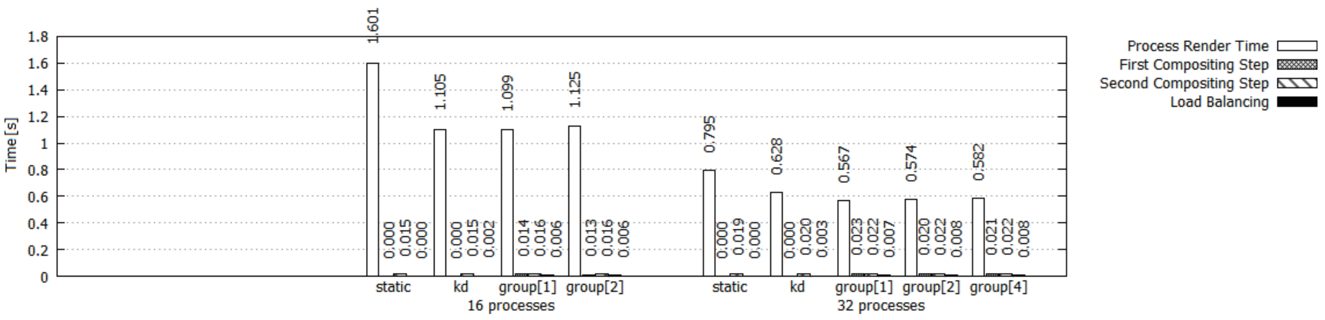
Performance variations due to different block sizes have been investigated in related work [7], which found that blocks of 64^3 voxels provided the best balance between fine-grained load balancing and extra overhead. These block dimensions are still used in some modern applications [13]. Based on this information we chose to partition the data sets into same-sized blocks consisting of around 64^3 voxels: 64 × 64 × 82, 64 × 64 × 60 and 64 × 64 × 47 voxels for the three respective data sets. As such, the two first data sets were partitioned into 32,768 blocks whereas the third data set was partitioned into 4096 blocks.

The *k*-d tree technique used for testing follows the definition in related work [7]. The initial block distribution produced by the *k*-d tree is used as the initial block distribution for the three examined techniques.

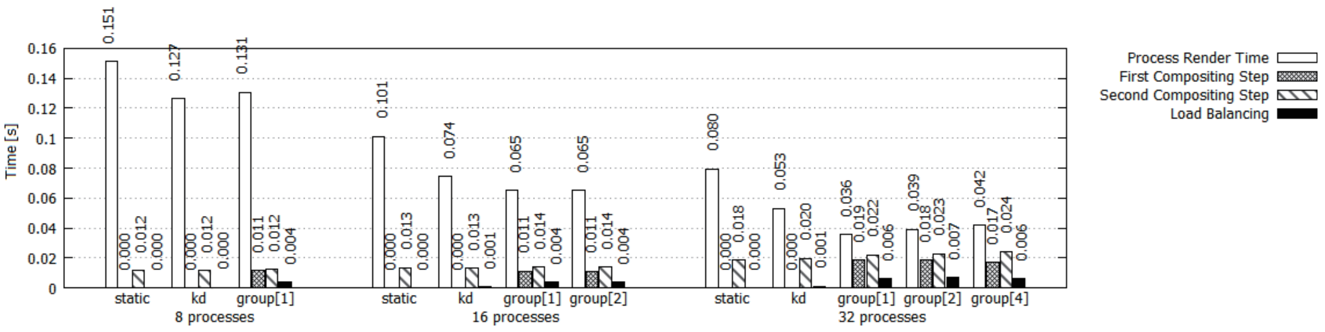
We developed a custom distributed volume rendering application to use during testing, which seamlessly can support the structures required by the *k*-d tree and group techniques. Volume data was stored in the float format, which is used internally by the developed rendering application. Ren-



(a)



(b)



(c)

Fig. 9 Overview of the computation and communication times of the various stages in the rendering pipeline, including the process render time, the two compositing steps and load balancing. The displayed values are the average of all frames using the (a) porcine heart, (b) Richtmyer-Meshkov instability and (c) Spathorhynchus fossorium data sets.

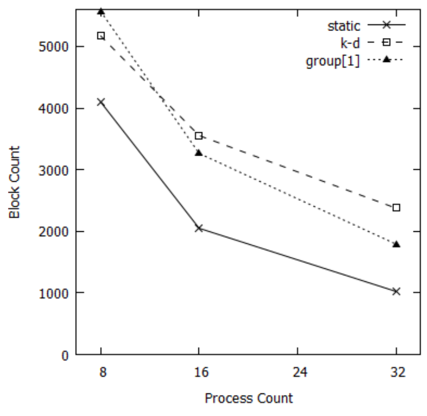
dering was carried out exclusively on GPUs using CUDA, whereas inter-process communication was performed using Open MPI. We used the binary swap [3] strategy in the IceT compositing framework [23] to compose images in the final compositing step of all evaluated techniques. Rudimentary empty space skipping [24] was used to avoid rendering empty blocks.

4.2 Performance Benefits of the Two-Layered Group Technique

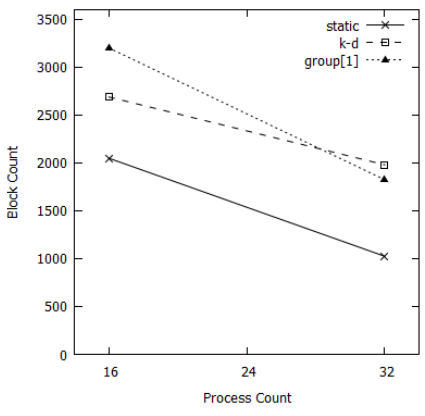
We performed each test multiple consecutive times for the three evaluated data sets. The performance difference between each run was negligible; constantly being less than 1%. An overview of the computation times of all stages of the pipeline is displayed in Fig. 9. *Process Render Time* represents the render time of the slowest process, excluding compositing, data transfers and load balancing. *First Compositing* and *Second Compositing* represent the total time required to perform the first and second compositing steps, respectively. *Load Balancing* represents the time required to load balance blocks, including data transfers.

The memory usage was measured by tracking the highest amount of blocks held in memory by a single process for each test, shown in Fig. 10. Using eight processes did not result in any substantial differences between the two dynamic techniques. For example, when using the porcine heart data set the highest recorded amount of blocks was 5568 for the group technique and 5168 for the *k*-d tree technique, i.e. 35.9% and 26.2% higher than using a static distribution, respectively. Using the Richtmyer-Meshkov instability data set in an eight-process configuration resulted in both dynamic techniques running out of memory, and is as such not included in the test results.

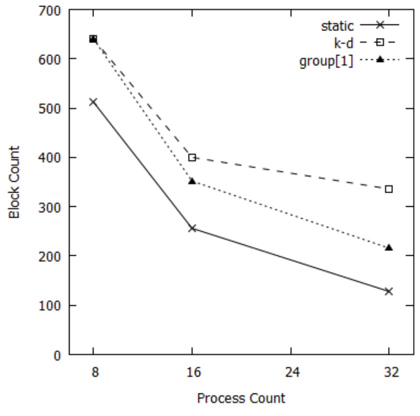
Increasing the number of processes to 32 resulted in the *k*-d tree technique having the highest memory usage in all tests. For the porcine heart data set the *k*-d tree and group techniques reached a memory usage of 2376 and 1792



(a)

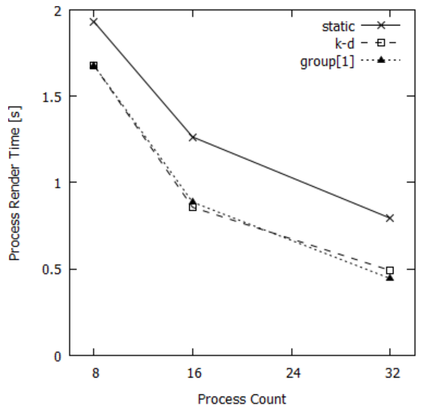


(b)

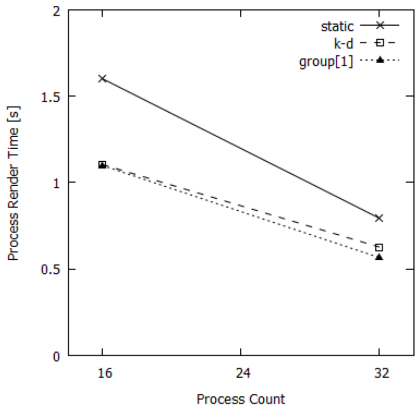


(c)

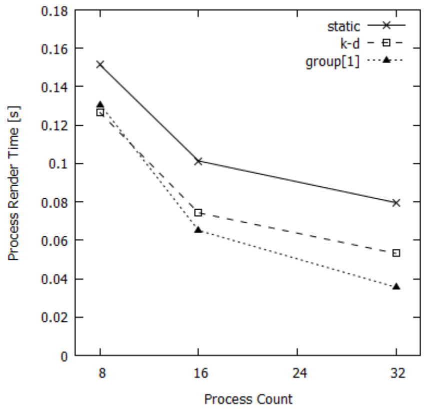
Fig. 10 The highest amount of blocks held in memory by a single process using the (a) porcine heart, (b) Richtmyer-Meshkov instability and (c) Spathorhynchus fossorium data sets on 8, 16 and 32 processes. Only one group is used in the case of the group technique.



(a)



(b)



(c)

Fig. 11 Average process render times using the (a) porcine heart, (b) Richtmyer-Meshkov instability and (c) Spathorhynchus fossorium data sets on 8, 16 and 32 processes. Only one group is used in the case of the group technique.

blocks, respectively; 132.0% and 75.0% higher than using a static distribution. The biggest difference between the two dynamic techniques was observed for the Spathorhynchus fossorium data set, where the *k*-d tree and group techniques held 162.5% and 68.8% more blocks in memory than the static distribution, respectively. The group technique consistently achieved a lower memory usage than the *k*-d tree technique as the number of processes increased. As such, we conclude that the group technique has a lower memory usage complexity.

Figure 11 shows the average process render times for the three data sets. Both evaluated dynamic techniques achieved lower process render times than the static distribution in all tests, clearly demonstrating the benefits of utilizing dynamic load balancing during large-scale visualization. However, as the amount of processes increased the group technique was able to achieve a lower render time than the *k*-d tree technique.

Using 8 or 16 processes resulted in similar process render times between the two dynamic techniques; the biggest difference was observed when using the Spathorhynchus fossorium data set, where the group technique was 12.3% faster. Increasing the process count to 32 resulted in the group technique consistently achieving the lowest render time; between 33.1% (Fig. 11 (c)) and 9.5% (Fig. 11 (a)) lower than the *k*-d tree technique.

The *k*-d tree technique transferred more blocks than the group technique in all test cases, as seen in Fig. 12. The gap widened as the number of processes increased, which validates our claim that the *k*-d tree technique induces an abundant amount of redundant data transfers. Using 32 processes resulted in the *k*-d tree technique transferring 227.1% (Fig. 12 (a)), 52.9% (Fig. 12 (b)) and 260.0% (Fig. 12 (c)) more data than the group technique for the three data sets. As an example, for the porcine heart data set 12,981 blocks were transferred when using the *k*-d tree technique. That amounts to 39.6% of the whole volume. All data sets used for evaluation were static, meaning that most of the load

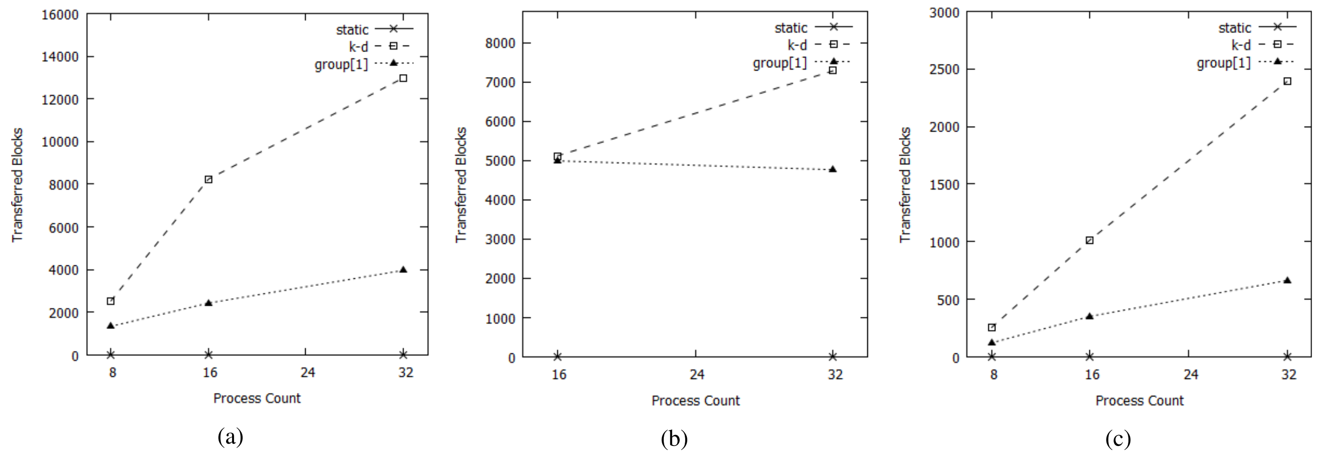


Fig. 12 The total number of transferred blocks using the (a) porcine heart, (b) Richtmyer-Meshkov instability and (c) Spathorhynchus fossorium data sets on 8, 16 and 32 processes. Only one group is used in the case of the group technique.

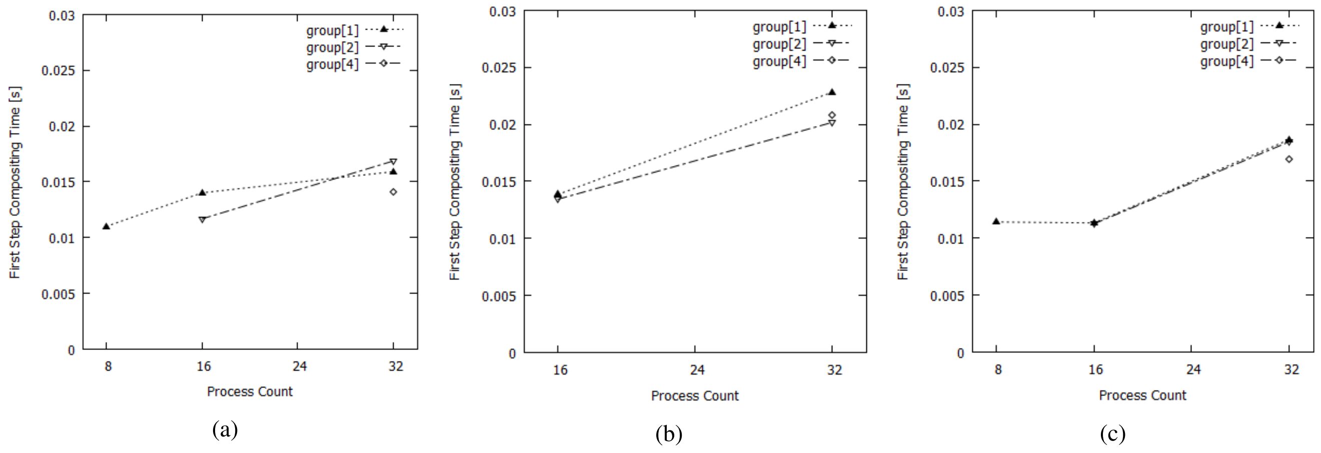


Fig. 13 The first-step compositing times for the group technique using the (a) porcine heart, (b) Richtmyer-Meshkov instability and (c) Spathorhynchus fossorium data sets on 8, 16 and 32 processes. In each test the group technique is evaluated using as few as eight processes per group.

balancing occurred during the first few frames of the visualization to equalize the initial load imbalance. During in-situ visualization the volume can change considerably at any time in the simulation, meaning that using a *k*-d tree technique could significantly affect I/O functionality. Furthermore, transferring too much data during the same frame could prove to be time consuming.

4.3 Utilizing Multiple Groups

To evaluate the scalability of the group technique's first compositing step we performed tests using down to eight processes per group, shown in Fig. 13. We observe that the first-step compositing times are similar for all three data sets and that they do not increase linearly in relation to the amount of processes. The highest increase was observed when going from 16 to 32 processes using the Spathorhynchus fossorium data set (Fig. 13 (c)). The compositing time increased from 11.3 to 18.7 ms when using one group; a 64.5% increase.

The recorded fist-step compositing times are sufficiently low to be performed asynchronously during the rendering stage, thus not resulting in any time overhead. Increasing the amount of groups generally lead to a lower compositing time. Although seemingly not required in a 32-process configuration, we believe that utilizing multiple groups can lead to a large performance increase if more processes are involved.

Decreasing the number of processes also limits between which processes load balancing can take place, resulting in higher render times. Figure 9 also includes the process render times for the group technique when using multiple groups; down to eight processes per group. Utilizing multiple groups sometimes results in a higher render time due to having too few processes in each group, which increases the chance of a high inter-group load imbalance.

5. Conclusion

We have presented a dynamic load balancing technique for large-scale volume rendering by which processes can render data from non-contiguous regions of the volume. By utilizing a two-layered group structure and a novel compositing pipeline we are efficiently able to resolve many scalability-related concerns that normally would arise with this type of design.

The effectiveness of the two-layered group technique was displayed by comparing it to a *k*-d tree load balancing technique in a variety of scenarios. The group technique proved to have a lower worst-case process memory usage, while simultaneously achieving similar or higher render performance. In addition, using the group technique significantly decreased the amount of redundant data transfers. We believe that the presented technique has the potential to be used in large-scale and memory-limited scenarios where *k*-d tree techniques currently do not suffice.

Next we plan to evaluate the technique in an in-situ scenario where the volume is not static. We would also like to evaluate the technique using more compute nodes to more accurately assess the benefits of utilizing multiple groups during large-scale visualization.

Acknowledgements

This study was supported in part by the Japan Society for the Promotion of Science KAKENHI Grant Numbers 15H01687 and 16H02801. We thank the anonymous reviewers for their insightful comments.

References

[1] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A sorting classification of parallel rendering," IEEE Comput. Graph. Appl., vol.14, no.4, pp.23–32, 1994.
[2] R.A. Drebin, L. Carpenter, and P. Hanrahan, "Volume rendering," ACM SIGGRAPH Comput. Graph., vol.22, no.4, pp.65–74, June 1988.
[3] K.-L. Ma, J.S. Painter, C.D. Hansen, and M.F. Krogh, "Parallel volume rendering using binary-swap compositing," IEEE Comput. Graph. Appl., vol.14, no.4, pp.59–68, 1994.
[4] A.C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. O'Leary, V. Vishwanath, B. Whitlock, and E.W. Bethel, "In situ methods, infrastructures, and applications on high performance computing platforms," Comput. Graph. Forum, vol.35, no.3, pp.577–597, 2016.
[5] A.H. Hassan, C.J. Fluke, and D.G. Barnes, "A distributed GPU-based framework for real-time 3D volume rendering of large astronomical data cubes," Publications of the Astronomical Society of Australia, vol.29, no.3, pp.340–351, 2012.
[6] J.L. Bentley, "Multidimensional binary search trees used for associative searching," Commun. ACM, vol.18, no.9, pp.509–517, 1975.
[7] S. Marchesin, C. Mongenet, and J. Dischler, "Dynamic load balancing for parallel volume rendering," Proc. 6th Eurographics Conference on Parallel Graphics and Visualization, pp.43–50, 2006.
[8] J. Zhang, H. Guo, F. Hong, X. Yuan, and T. Peterka, "Dynamic load balancing based on constrained K-D tree decomposition for parallel particle tracing," IEEE Trans. Vis. Comput. Graph., vol.24, no.1, pp.954–963, 2018.
[9] C. Müller, M. Strengert, and T. Ertl, "Adaptive load balancing for raycasting of non-uniformly bricked volumes," Parallel Computing, vol.33, no.6, pp.406–419, 2007.
[10] W.-J. Lee, V.P. Srini, W.-C. Park, S. Muraki, and T.-D. Han, "An effective load balancing scheme for 3D texture-based sort-last parallel volume rendering on GPU clusters," IEICE Trans. Inf. & Syst., vol.E91-D, no.3, pp.846–856, March 2008.
[11] V. Bruder, S. Frey, and T. Ertl, "Prediction-based load balancing and resolution tuning for interactive volume raycasting," Visual Informatics, vol.1, no.2, pp.106–117, 2017.
[12] R.B. Ross, T. Peterka, H.-W. Shen, Y. Hong, K.-L. Ma, H. Yu, and K. Moreland, "Visualization and parallel I/O at extreme scale," Journal of Physics: Conference Series, vol.125, no.1, 012099, 2008.
[13] M. Dorier, R. Sisneros, L.B. Gomez, T. Peterka, L. Orf, L. Rahmani, G. Antoniu, and L. Bougé, "Adaptive performance-constrained in situ visualization of atmospheric simulations," 2016 IEEE International Conference on Cluster Computing (CLUSTER), pp.269–278, 2016.
[14] N. Fabian, K. Moreland, D. Thompson, A.C. Bauer, P. Marion, B. Gevecik, M. Rasquin, and K.E. Jansen, "The ParaView coprocessing library: A scalable, general purpose in situ visualization library," 2011 IEEE Symposium on Large Data Analysis and Visualization, pp.89–96, Oct. 2011.
[15] U. Neumann, "Communication costs for parallel volume-rendering

algorithms," IEEE Comput. Graph. Appl., vol.14, no.4, pp.49–58, 1994.

[16] H. Yu, C. Wang, and K.-L. Ma, "Massively parallel volume rendering using 2–3 swap image compositing," SC '08: Proc. 2008 ACM/IEEE Conference on Supercomputing, pp.48:1–48:11, 2008.

[17] F. Ino, T. Sasaki, A. Takeuchi, and K. Hagihara, "A divided-screenwise hierarchical compositing for sort-last parallel volume rendering," Proc. International Parallel and Distributed Processing Symposium, 2003.

[18] J. Ahrens and J. Painter, "Efficient sort-last rendering using compression-based image compositing," Proc. 2nd Eurographics Workshop on Parallel Graphics and Visualization, pp.145–151, 1998.

[19] P. Klacansky, "Open SciVis Datasets," Oct. 2018.

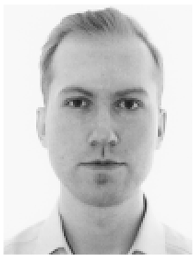
[20] NVIDIA, "About CUDA | NVIDIA developer," Oct. 2018.

[21] The Open MPI Project, "Open MPI: Open source high performance computing," Oct. 2018.

[22] M. Larsen, K. Moreland, C.R. Johnson, and H. Childs, "Optimizing multi-image sort-last parallel rendering," 2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV), pp.37–46, 2016.

[23] K. Moreland, W. Kendall, T. Peterka, and J. Huang, "An image compositing solution at scale," SC '11: Proc. 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pp.1–10, 2011.

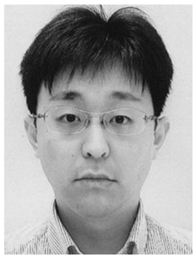
[24] W. Li, K. Mueller, and A. Kaufman, "Empty space skipping and occlusion clipping for texture-based volume rendering," Proc. 14th IEEE Visualization 2003 (VIS '03), pp.317–324, 2003.



Marcus Wallden received his B.S. degree in information technology and M.S. degree in computer science from KTH Royal University of Technology in Stockholm, Sweden. He is currently working towards a Ph.D. degree at the Graduate School of Information Science and Technology at Osaka University. His research interests include visualization techniques for scientific simulations and high performance computing.

Stefano Markidis is an Associate Professor in High Performance Computing and Docent in Computer Science at KTH Royal Institute of Technology. He holds a M.S. degree from Politecnico di Torino and a Ph.D. degree in Nuclear Engineering from the University of Illinois at Urbana-Champaign. Before joining KTH, he was a graduate research assistant at Los Alamos National Laboratory and Lawrence Berkeley National Laboratory and postdoc at KU Leuven. Stefano won two R&D100 awards in 2005 and 2017 as part of the CartaBlanca and SHIELDS projects. He is the author of more than one hundred peer-reviewed publications.

Masao Okita is an Assistant Professor in the Department of Computer Science, Graduate School of Information Science and Technology, Osaka University since 2009. He received the B.E. and M.E., and Ph.D. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 2001, 2003, and 2006, respectively. From 2006 to 2009, he worked for Acces Co., Ltd., as a section chief. His research interests include parallel processing, especially physiological simulation on supercomputers and GPU-accelerated machine learning.



Fumihiko Ino received the B.E., M.E., and Ph.D. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 1998, 2000, and 2004, respectively. He is currently a Professor in the Graduate School of Information Science and Technology at Osaka University. His research interests include parallel and distributed systems software development tools, and performance evaluation.