Accelerating On-the-Fly Visualization and Analysis of Large-Scale Scientific Simulations

Submitted to
Graduate School of Information Science and Technology
Osaka University

July 2021

Marcus Walldén

Published Papers

Journal Papers

1. Marcus Walldén, Masao Okita, Fumihiko Ino, Dimitris Drikakis, and Ioannis Kokkinakis, "Accelerating In-Transit Co-Processing of Scientific Simulations Using Region-Based Data-Driven Analysis," *Algorithms*, 14(5):154, May 2021.

2. Marcus Walldén, Stefano Markidis, Masao Okita, and Fumihiko Ino, "Memory Efficient Load Balancing for Distributed Large-Scale Volume Rendering Using a Two-Layered Group Structure," *IEICE Transactions on Information and Systems*, E102-D(12):2306–2316, Dec. 2019.

Technical Reports

3. Marcus Walldén, "Accelerating In-transit Co-processing for Scientific Simulations Using Region-based Data-driven Adaptive Compression," *Osaka University Cybermedia HPC Journal*, 10:11–14, Jan. 2021.

Poster Presentations

4. Marcus Walldén, "Effective Load Balancing for Distributed Large-Scale Volume Rendering Using a Two-Layered Group Structure," In *Proceedings of the 11th Symposium on Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures*, p.76, July 2019.

Other Publications

5. Ruiyun Zhu, Yuji Misaki, Marcus Walldén, and Fumihiko Ino, "Cache-Aware Volume Rendering Methods with Dynamic Data Reorganization," *Journal of Visualization*, 24(2):275–288, Feb. 2021.

6. Jingcheng Shen, Jie Mei, Marcus Walldén, and Fumihiko Ino. "Integrating GPU Support for FreeSurfer with OpenACC," In *Proceedings of the 6th IEEE International Conference on Computer and Communications (ICCC 2020)*, pages 1622–1628, Dec. 2020.

7. Ruiyun Zhu, Jingcheng Shen, Xiangtian Deng, Marcus Walldén, and Fumihiko Ino. "Training Strategies for CNN-based Models to Parse Complex Floor Plans," In *Proceedings of the 9th International Conference on Software and Computer Applications (ICSCA 2020)*, pages 11–16, Feb. 2020.

Unpublished Papers

Journal Papers

1. Marcus Walldén, Jingcheng Shen, Masao Okita, and Fumihiko Ino. “Accelerating Multi-Image Compositing using Dynamic Image Resolutions.” Manuscript submitted for publication, 2021.

Abstract

Data generated by large-scale scientific simulations is expected to increase by orders of magnitude in the future as we approach exascale computing. Input/output constraints of supercomputers have increased the use of *co-processing* approaches, i.e., visualizing and analyzing scientific simulations on the fly. Co-processing tasks consume valuable simulation time, thus affecting the fidelity and scale of phenomena that can be simulated in a given time frame. As the complexity and scale of both simulations and co-processing tasks are bound to increase in the future, there is a need for new techniques to accelerate challenging co-processing tasks.

In this dissertation, we identify and address three challenges facing the large-scale co-processing of simulation data. The first challenge is how to expeditiously determine what constitutes important data. Analysis and visualization tasks can be focused on the most essential data, thus accelerating the co-processing. We present a method that evaluates the importance of different regions of simulation data and a data-driven approach that uses this information to accelerate the in-transit co-processing of large-scale simulations. We use the importance metrics to simultaneously employ multiple compression methods on different data regions to accelerate the in-transit co-processing. Our approach strives to adaptively compress data on the fly and uses load balancing to counteract memory imbalances. We demonstrate the method's efficiency through a fluid mechanics application, a Richtmyer–Meshkov instability simulation, showing how to accelerate the in-transit co-processing of simulations. The results show that the proposed method can identify regions of interest expeditiously, even when using multiple metrics. Our approach achieved a speedup of 1.29× in a lossless scenario. The data decompression time was sped up by 2× compared to using a single compression method uniformly.

The second challenge concerns load balancing simulation data. In large computing clusters, dynamically load balancing data can lead to significant inter-process memory imbalances; as a result, data is typically statically distributed among processes. We propose a novel compositing pipeline and a dynamic load balancing technique for volume rendering that utilizes a two-layered 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 rendering time. We demonstrate the effectiveness of the proposed technique by performing a set of experiments on a computing 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 lower rendering times. 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 in large-scale visualization.

The third challenge relates to image batch visualization. In many cases, rather than saving large quantities of simulation data, thousands to millions of images of different variables and viewpoints can be rendered on the fly and saved to permanent storage. This image generation process can be accelerated by rendering and compositing images in batches. Specifically, images can be combined into larger *multi-images*, which results in less synchronization and communication overhead during the image compositing stage. We present a technique to accelerate such batch processing, called dynamic image resolutions. The dynamic image resolutions technique maps regions of blank pixels in each image and uses this information to dynamically restructure the multi-images to reduce the total image size with no loss of detail. An evaluation of the technique demonstrates a 2.02× speedup of the compositing stage as compared to traditional image compositing and a 1.82× speedup compared to existing multi-image techniques.

Our work successfully addresses three important challenges facing on-the-fly co-processing of large-scale simulation data. The first application shows how essential data can be identified, and how this information can be used to accelerate co-processing tasks of large-scale data sets. The second application significantly reduces memory imbalances resulting from dynamic load balancing, thus making load balancing feasible in large-scale environments. The third application shows that batch image visualization can be sped up significantly by taking advantage of the underlying image data.

Acknowledgements

I want to express my deepest gratitude to my supervisor, Professor Fumihiko Ino, for all his advice, support, and friendship throughout this journey. I would also like to thank Associate Professor Masao Okita for his encouragement and many insightful discussions.

Besides my supervisor, I want to sincerely thank the rest of my thesis committee, Professor Katsuro Inoue and Professor Toshimitsu Masuzawa.

I would like to thank my collaborators. Associate Professor Stefano Markidis at KTH Royal Institute of Technology, for his guidance and motivating comments, Professor Dimitris Drikakis at the University of Nicosia, and Lecturer Ioannis Kokkinakis at the University of Strathclyde, for their insight and assistance in multidisciplinary research projects.

I would also like to thank the Sysmex Corporation for their support, especially Hiroko Nakatsuka and Yuta Nakano.

Finally, I am grateful for the daily support, discussions, and friendships with members of the Supercomputing Engineering Laboratory.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The processing capabilities of distributed compute clusters are increasing at a fast pace; however, limited input/output (I/O) bandwidth leads to bottlenecks in many applications [12, 13]. This is one of the most pressing matters for large-scale scientific simulations. Simulations can generate multiple tera- or petabytes of data, making it challenging to save all necessary data to permanent storage [14–17]. One popular approach to circumvent this limitation is to perform *co-processing* (i.e., to visualize and analyze simulation data on the fly). Co-processing tasks can be time consuming, thus limiting the available simulation time. In this dissertation, we identify and address three challenges facing on-the-fly co-processing of large-scale scientific simulations. Specifically, we accelerate three time-consuming co-processing tasks, thus freeing up time to perform more advanced and detailed simulations. We first introduce co-processing approaches and the visualization pipeline in Section 1.1. We then present the challenges and our solutions in Sections 1.2–1.4.

1.1 On-the-Fly Co-Processing of Simulation Data

Computer simulations are used in a wide variety of fields to study how phenomena interact and develop in certain environments. Simulations are especially prevalent for applications that otherwise would be expensive or difficult to test, such as fluid mechanics simulations [18, 19]. Scientific simulations are typically performed in three-dimensional (3D) space (x, y, and z-axis) and span multiple variables and time steps. Large-scale simulations have extensive compute and memory requirements, requiring the use of supercomputers or computer clusters. The simulation domain can be partitioned into multiple contiguous and convex *blocks*; contiguous and convex data regions that combined make up the simulation data or 3D data set. Such blocks can be distributed and processed in parallel by multiple processes. Researchers can perform various analyses or visualizations on the data to extract information about the simulation.

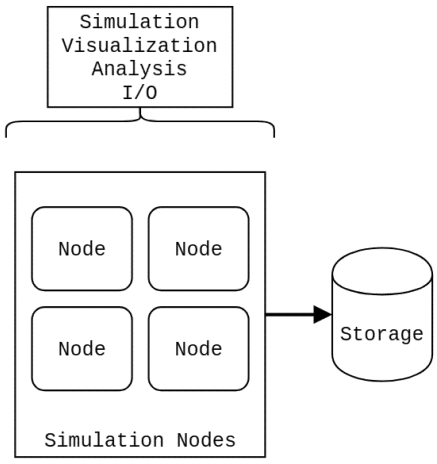


Figure 1.1: In-situ co-processing. Simulation, visualization, analysis, and I/O operations are performed on the same set of processes.

Co-processing enables researchers to analyze and visualize all generated simulation data on the fly, as it is created. The often small-sized output from co-processing tasks (e.g., images or analysis results) can then be saved to permanent storage instead of the larger simulation data sets. As a result, time-consuming I/O operations can be kept at a minimum, whilst information about the simulation can be kept for post-hoc analysis. In addition, more resources and time can be dedicated to the simulation, allowing researchers to run larger simulations and process more data. Typically, two types of on-the-fly co-processing approaches are considered: *in-situ* and *in-transit*.

In-situ co-processing (illustrated in Fig. 1.1) is typically performed on the same compute nodes as the simulation and can achieve good performance, partly because of the locality of the data. The main drawback of in-situ co-processing is that time-consuming analysis and visualization tasks consume valuable simulation time. On the other hand, in-transit co-processing (shown in Fig. 1.2) is performed on a separate group of compute nodes. Utilizing a different group of nodes means that co-processing can be performed asynchronously during the simulation stage, resulting in a less strict time limitation compared to an in-situ approach. Furthermore, nodes can be equipped with different hardware to accelerate the co-processing tasks. The main drawback of in-transit co-processing is the time-consuming data transfers needed to relocate necessary simulation data. Neither co-processing approach is objectively better than the other in all cases [20]; in many situations, in-situ and in-transit co-processing can be used in combination to take advantage of their respective benefits [16, 21].

Visualization is a common co-processing task to understand and store information about the studied phenomenon. In the context of scientific simulations, it is the process of creating one or more images of the simulations' underlying data variables. In a distributed

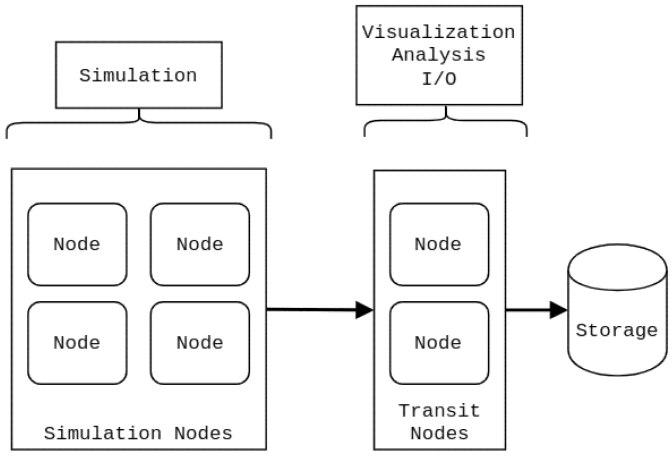


Figure 1.2: In-transit co-processing. The simulation is performed on one set of nodes, whereas visualization, analysis, and I/O operations are performed on a different set.

setting, visualization consists of two stages: rendering and compositing. In the rendering stage, one or more images are generated on each compute process based on the partial simulation data present on each process. As the images only contain information on part of the simulation domain, they need to be combined into a single, complete image. This work is done in the compositing stage.

1.2 Determining Essential Simulation Data

Important regions of simulated phenomena are often limited to a subset of the simulation domain. By identifying important data on the fly, regions not of interest can be down-prioritized by analysis, visualization, and I/O processes, which accelerates such operations. Another use case is to perform more fine-grained simulation steps on regions identified as important, which is faster than uniformly performing the same steps on the whole domain [8, 9].

The first challenge (Chapter 2) concerns how to determine the importance of different data regions. What constitutes important data depends on multiple factors, such as the simulation type and the focus of the study. As a result, there is no definitive approach to analyze the importance of data that can be used in all situations. The importance of a data region can be tied to multiple different metrics, and it must be possible to make multiple decisions about the data based on such metrics. A flexible pipeline that can adapt to these different situations is needed.

To accurately determine the importance of data, we present a method to efficiently calculate the importance of regions of simulation data, as well as a data-driven approach

that uses the proposed method in-situ to accelerate the in-transit co-processing of a fluid simulation. Our hybrid in-situ in-transit approach strives to adaptively compress regions of data on the fly, using multiple different compression methods based on the underlying data. In addition, we use load balancing to counteract memory imbalances.

1.3 Dynamic Load Balancing of 3D Data Sets in Large-Scale Cluster Environments

Differences in computing times between processes can significantly affect the rendering and compositing times when visualizing multiple images (e.g., when visualizing images in batches). Dynamic load balancing techniques are often used to alleviate these differences in compute time [22–26]. Load balancing techniques typically structure the underlying data in a tree structure, such as a k-d tree [27]; however, balancing the compute times using this type of tree structure can instead result in substantial memory imbalances [26].

The second challenge (Chapter 3) concerns large-scale dynamic load balancing of 3D data sets. Post-hoc exploration of certain time steps or using tools like cinema [28] to visualize simulation data on the fly results in rendering multiple images from the same data sets. Memory imbalances resulting from dynamic load balancing can, when using large-scale simulations, lead to some processes running out of memory. As a result, dynamic load balancing is seldom used in large-scale settings to balance compute times.

To limit memory imbalances caused by load balancing 3D data sets, we propose a novel compositing pipeline and a dynamic load balancing technique for volume rendering that utilizes a two-layered group structure to achieve memory-efficient and scalable load balancing. The technique can be used in both in-situ and in-transit settings and enables each compute process to render data from non-contiguous regions of the volume with minimal impact on the total rendering time.

1.4 Accelerating Data Visualization Using Multi-Images with Dynamic Image Resolutions

In the context of scientific simulations, multiple images are often generated to represent different viewing angles, variables, and time steps. Certain tools [28, 29] automate this process by generating thousands to millions of images at regular intervals around the studied phenomenon in 3D space. These in-situ visualization tools can reduce the stored data size by orders of magnitude, while still enabling researchers to explore the data post-hoc.

Consider several images, each representing a specific simulation variable or unique camera position. In a distributed setting, each process holds a small region of the simulation

domain in memory [30, 31]. As a result, images must be composited among the compute processes after being rendered. Typically, each image is rendered and composited in sequence. However, this type of image generation pipeline induces a substantial amount of overhead, primarily as a result of synchronization and communication between different processes. Such overhead can be reduced by rendering and compositing images in batches, as has been explored in related work [32]. Combining batches of images into *multi-images* proved to further accelerate the compositing process.

The third challenge (Chapter 4) is related to multi-image batch visualization. Multi-image techniques [32] have proved to substantially accelerate the batch processing of large quantities of images. However, rendering and compositing potentially up to millions of images remains a very time-consuming process. As a result, there is a need to further accelerate image batch processing techniques such as the multi-image technique. The amount of data generated by scientific simulations is bound to increase by orders of magnitude in the future as we approach exascale computing, making this type of co-processing essential to analyze simulations post-hoc.

We present a technique called dynamic image resolutions to accelerate on-the-fly multi-image batch visualization of distributed simulation data in large-scale computing environments. The dynamic image resolutions technique maps regions of blank pixels in each image. This information is then used to dynamically restructure the multi-images to reduce the total image size with no loss of detail. Reducing the image size significantly accelerates image compositing and the overall visualization process.

1.5 Overview of the Dissertation

The rest of this dissertation is organized as follows. Chapter 2 reports our work on the first challenge: determining essential simulation data, and how that information can be used to accelerate in-transit co-processing. Our work to address the second challenge, limiting memory imbalances caused by load balancing in large-scale environments, is presented in Chapter 3. Our solution to the third challenge is presented in Chapter 4: accelerating batch visualization by using multi-images with our novel dynamic image resolutions technique. Finally, Chapter 5 concludes the dissertation and presents directions for future work.

Chapter 2

Accelerating In-Transit Co-Processing for Simulations Using Data-Driven Analysis

2.1 Introduction

The processing capabilities of modern supercomputers are improving at a tremendous pace. However, I/O bandwidth has advanced at a much slower rate, leading to bottlenecks in many applications [12, 13]. This is one of the most pressing matters for large-scale scientific simulations. Simulations can result in multiple tera- or petabytes of generated data, making it challenging to save all necessary data to permanent storage due to limited storage capacity or time-consuming I/O operations [14–17]. In-situ or in-transit visualization and analysis are often used for each time step while the data is generated.

In-situ co-processing is typically performed on the same compute nodes as the simulation and can achieve high performance, partly because of the locality of the data. However, in-situ co-processing takes up valuable simulation time. Many researchers are reluctant to dedicate computing resources used by the simulation to other compute-intensive tasks [13, 15], meaning that the time available for in-situ co-processing often is short compared to that of the simulation. One option to reduce the co-processing time is to lower the frequency at which time steps are analyzed. Another is to limit the available time to perform co-processing for each analyzed time step. Both these options are undesirable because they limit the amount of data available to the researcher after the simulation has been completed. Furthermore, a significant amount of the available memory is used by the simulation in an in-situ scenario, meaning that only a tiny amount can be allocated for visualization or analysis purposes [33].

In contrast, in-transit co-processing [20, 34] is generally performed on a separate group of compute nodes, which we refer to as *transit nodes*. Utilizing a different group of nodes

means that co-processing can be performed asynchronously during the simulation stage, resulting in a less strict time limitation than an in-situ approach. Furthermore, the transit nodes can be equipped with different hardware to accelerate the co-processing tasks. The main drawback of in-transit co-processing is that the relevant simulation data needs to be transferred to the transit nodes. Moreover, it is not guaranteed that the simulation nodes can hold an extra copy of the data in memory. It is often impossible to transfer the data asynchronously while simulating the next time step.

The data transfers required to perform in-transit co-processing can be accelerated by reducing or compressing the simulation data. However, using a reduction or a lossy compression method could lower the detail and accuracy of the whole data set, including regions of interest. It could instead be beneficial to selectively compress areas of data based on their contribution to the simulated phenomenon. Reducing or compressing areas that are not of interest would result in less time-consuming data transfers without any significant loss to the data quality.

To determine the frequency at which data is analyzed, the time available for co-processing, and what compression method(s) to use to accelerate the data transfers, there is a need to identify the importance of the underlying simulation data. Such information can be used in a wide variety of cases, including guiding the simulation, determining which time steps of the simulation to analyze or save to permanent storage, removing or reducing unimportant data, and finding interesting camera locations for co-processing purposes. Some of these cases have been explored in related work [4–11]. However, these methods have either been limited in the number of used importance metrics [4, 6–11] or been unable to combine multiple metrics to create advanced importance metrics and to make more advanced decisions [5].

We propose an in-situ method that efficiently can identify the importance of subsets of simulation data, which consists of multivariate and temporal data sets in the form of structured rectilinear grids. User-defined importance metrics and filters are used to determine the importance of blocks. The calculated importance can then be used for a variety of purposes to accelerate or guide the simulation, such as identifying important regions, down-sample, reducing, compressing, or simply removing parts of the data based on user-defined constraints and the underlying hardware. What sets the proposed method apart from related work is its ability to calculate importance when using multiple analyses efficiently. It can adaptively make any number of decisions based on the importance, compared to only a binary decision, which is typical in other methods.

We also propose an approach that uses the proposed in-situ method to identify how to best combine the usage of multiple different compression methods based on the importance of the underlying data, which can be seen as a use case of the proposed method. In the case of lossy compression, loss of detail in regions of interest can be kept at a minimum by reducing unimportant areas. Using this data-driven approach, we strive to reduce the data size and the in-transit data transfer time to accelerate in-transit co-processing. The main contributions of this work are as follows:

1. **An efficient method to determine the block importance of large-scale simulations.** Calculating the importance of various regions of data was a time-consuming task in many related works. As such, only a few importance metrics tend to be used. The main goal of our proposed method is to provide researchers with a customizable and easy-to-use way to efficiently determine the importance of subsets of large-scale simulation data, even when using multiple complex analyses to determine such importance. To reach this goal, we have developed a scheme that uses a separate buffer to transform the data to a more suitable structure for analysis and strives to schedule importance analyses in a fashion that improves the cache hit rates. We also introduce the usage of an adaptive *condition window*; a custom range within which multiple condition parameters can vary. Using a condition window allows parameters to change based on the current constraints of the environment. For example, if a data transfer is too time-consuming, the parameters can be adapted to compress or remove more data in the coming time step.

2. **A flexible approach to accelerate in-transit co-processing.** We observe that the effectiveness of a certain compression algorithm often depends on the underlying compressed data. By identifying the importance and the most suitable compression for each region of generated simulation data in-situ, we strive to minimize the data size and the in-transit data transfer time by combining the use of multiple compression methods integrated into a pipeline. The approach explored in this chapter puts no additional restraints on the contents of the co-processing stage. It can be used in tandem with various visualization software, e.g., Paraview [35], VisIt [36], or OSPRay [37].

3. **A case study of how the in-transit co-processing of a Richtmyer–Meshkov instability (RMI) simulation can be accelerated using the proposed approach.** The RMI simulation is performed using CNS3D, a state-of-the-art program for numerical fluid simulations.

The structure of this chapter is organized as follows. Related work is discussed in Section 2.2. The proposed approach is presented in Section 2.3. In Section 2.4, we evaluate the performance of the proposed approach. Lastly, our conclusions are presented in Section 2.5.

2.2 Related Work

Determining and analyzing regions of interest in 3D data sets has been an important topic in many research fields, and has as such been explored in many related works [4–11]. A commonly used technique is Adaptive Mesh Refinement (AMR) [7–9]. Using AMR, the simulated region is divided into multiple sub-regions, commonly by using a tree structure

such as an octree [38]. After the simulation has been completed, important regions in the tree structure are subdivided to create even smaller sub-regions, on which more compute-intensive operations can be used to, for example, increase the accuracy or resolution of the simulation. This process can be performed recursively based on the researcher's specifications. The analysis required to calculate block-based importance metrics results in a fairly uniform data access over all data points in the simulation region. As such, the recursive, top-down AMR approach could result in redundant calculations and low cache hit rates. This notion is confirmed in related work [6], where a bottom-up approach, i.e., calculating the importance of each individual block, could achieve significantly better performance.

Close to our work is a paper by Dorier et al. [4], in which calculated importance metrics were used to adaptively reduce unimportant blocks. The technique targets explicitly in-situ visualization and supports elementary data reduction and load balancing based on a random distribution. However, the technique only supports a single reduction strategy. Furthermore, their technique assumes that simulation data is stored as blocks, which is not the case for most simulations. For their technique to work in the general case, all simulation data first needs to be preprocessed and partitioned into blocks, which is time-consuming and increases the memory usage as two sets of the simulation data need to be maintained while performing the importance analysis.

Wang et al. [5] introduced an importance curve to store the derived importance of all data blocks for a time-varying data set. By analyzing the changing block importance between different time steps it would be possible to characterize temporal behaviors exhibited by simulation data. Such information could be used to make informed processing decisions, e.g., setting a time budget for the co-processing stage or saving data to permanent storage.

Nouanesengsy et al. [6] presented a prioritization method for 3D data sets inspired by AMR. Data sets were recursively partitioned into smaller regions based on user-defined importance metrics. A prioritization tree was constructed for the data set, which then could be used to identify interesting camera placements or to determine compression strategies for saving data to permanent storage.

Some data sampling and summarization methods [10, 11] use importance based on entropy metrics to prioritize the reduction of unimportant data. Here, reduced subsets of the simulation data are saved to permanent storage for post-hoc analysis; more than 99% of simulation data is removed in some applications [10]. Data sampling methods are not suited for applications that operate on lossless simulation data. Moreover, these methods do not support use cases that require custom metrics to define important data.

Table 2.1 shows a qualitative comparison to related work. Here, we specifically compare features related to the use case considered in this work. Method [5] is processed post-hoc, whereas methods [6–9] have no defined behavior of how to use the calculated importance values to compress unimportant data. Out of the methods used for comparison, only related work [4, 10, 11] explore using the data importance to prioritize compression

Table 2.1: Comparing features of related work [4–11] to this work.

	Method					
	This work	[4]	[5]	[6]	[7–9]	[10, 11]
Processing approach	In-transit	In-situ	Post-hoc	In-situ	In-situ	In-situ
Supports custom importance metrics	✓	✓		✓	✓	
Decisions based on multiple metrics	✓		✓			
Tracks time-varying data			✓	✓		
Used to compress unimportant data	✓	✓				✓
	lossless/lossy	lossy				lossy
Multiple compression methods	✓		—	—	—	

or reduction of unimportant data. However, in contrast to our work, none of the methods support lossless compression or simultaneously using multiple compression and reduction methods. In addition, there is no defined behavior of how to use the methods for in-transit co-processing. A quantitative comparison is difficult, as each work has a different feature set and use case. We, therefore, chose to evaluate our method independently, as was also done in related work [4–6].

Many visualization and analysis techniques have been proposed for in-situ and in-transit co-processing. Some aim to batch-render images of each time step of the simulation from multiple viewpoints, thus allowing researchers to interactively explore different regions of the simulation post-hoc [29, 39]. Other techniques focus on extracting features, metadata, or samples of the simulation [13, 40–42]. Using many different types of visualizations and analyses would generally be desired to extract as much information from the simulation as possible. However, each additional task would increase the execution time. The logical approach would be to use in-transit co-processing in such a scenario due to the often limited time available to perform such computations in-situ. To make in-transit co-processing a viable option, we need new techniques to achieve faster data transfers between simulation and transit nodes. Many researchers have developed methods to lower the data transfer time required to perform in-transit co-processing [16, 43–45]. Most works have focused on uniformly reducing or compressing data [16, 43, 44]. Although efficient, regions of interest within the data are reduced to the same extent as unimportant regions.

Lossy compression methods such as zfp [46] and an extension of SZ [47] strive to lower the compression error by analyzing the entropy of the data. Data sets are split into blocks of size $4 \times 4 \times 4$ and $6 \times 6 \times 6$, respectively. A function is then used to predict a suitable level of compression for each block, based on the estimated compression error.

These lossy methods have proved to achieve good performance [46, 47]. However, the fact that they are lossy makes them unusable in many use cases. We also note that these methods are unable to determine the importance of regions of data and that although the level of compression varies, they ultimately use the same lossy compression method on the whole data set. Both of these compression methods can be integrated into the proposed approach, as presented in Section 2.3, to be used on a subset of the available data blocks.

In summary, effectively performing in-transit co-processing of data generated by scientific simulations remains an important research topic in the field of high-performance computing. Many methods which uniformly reduce or compress data have been proposed. However, the characteristics of the simulation data can vary substantially depending on the current time step, meaning that the need for compression and analysis also can vary. Some techniques which analyze the characteristics of the data also exist. However, one of the most commonly used techniques, AMR, is often unable to efficiently calculate individual block importance because of the different data access pattern. Although alternatives [4–6, 10, 11] have been proposed, they are generally limited in scope or in their ability to perform multiple analyses to determine block importance or characteristics of different parts in the studied simulation data. Our work improves upon this related work by efficiently handling multiple importance analyses and by adaptively utilizing multiple different compression and reduction strategies.

2.3 Adaptive In-Transit Co-Processing

In this section, we present the proposed method and an approach to accelerate in-transit co-processing. To the best of our knowledge, this is the first work to accelerate in-transit processing by simultaneously using more than one compression method based on multiple importance analyses of the underlying simulation data. The workflow of the approach is shown in Fig. 2.1.

The approach consists of three distinct stages:

1. The in-situ stage (Sections 2.3.1– 2.3.4), which consists of the proposed method. The block importance is calculated, and which compression method to use is determined on a per-block basis.

2. The distribution stage (Section 2.3.5), where data is compressed, load balanced, and transferred over the network to the transit nodes.

3. The in-transit stage (Section 2.3.6), where compressed data is decompressed and restructured on the transit nodes.

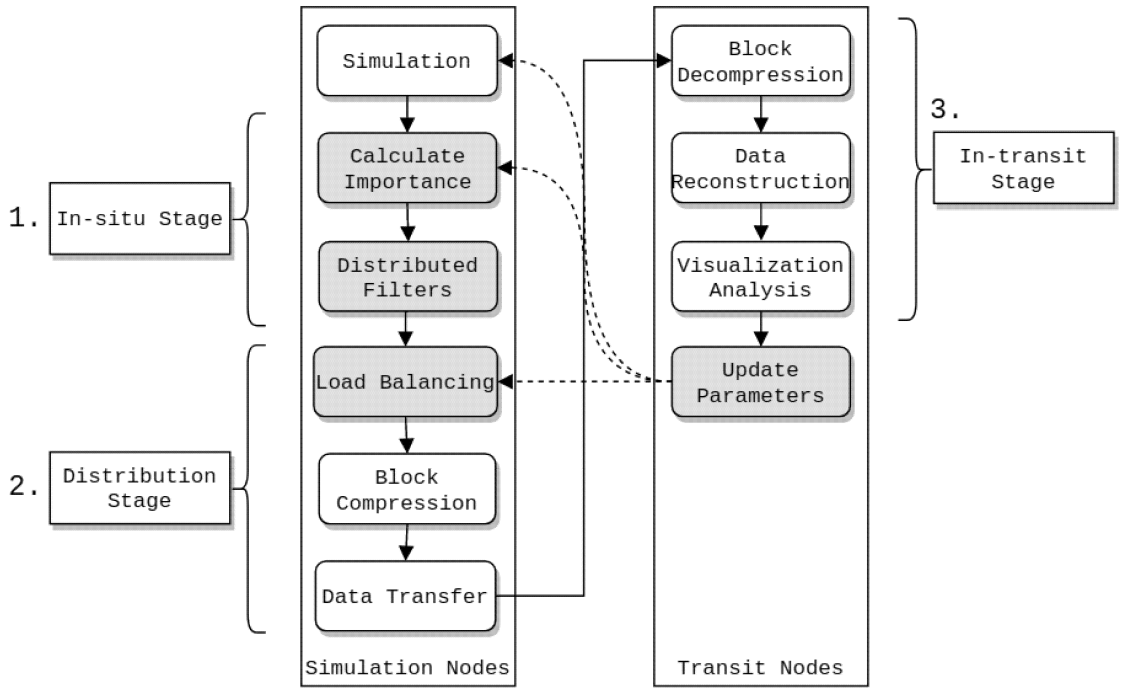


Figure 2.1: Typical in-transit workflow using the proposed approach. 3D data sets are partitioned into blocks, analyzed, and compressed in parallel on the simulation nodes. The blocks are then transferred to the transit nodes, where they are decompressed and reconstructed into the original data sets. The in-situ stage consists of the proposed method. The proposed approach requires additional operations (highlighted in gray) compared to a typical in-transit workflow that uses a single compression method.

2.3.1 Calculating Importance

Calculating block importance has been a time-consuming task in many related works, even for relatively small data sets. Because of this reason, the block importance in related work has generally been calculated using one or a few importance measurements. However, simulation data often contains many different types of regions of interest. Furthermore, the best compression algorithm might differ based on the entropy of the data in the different regions, meaning that there are many situations in which using multiple importance measurements would be preferable. Typically, the computation time would increase proportionally to the number of used importance measurements. However, in our solution, we strive to minimize this increase in computation time. In addition to performing importance calculations on a block's complete simulation data, we also support using a random sample of a block's data. That is to say, importance calculations can either be performed on the entire simulation data of a block or a smaller, random sampled subset

of a block's data. Using only a sample of the simulation data can result in a significant speedup. The calculated importance would not be entirely accurate, which might be an issue in specific applications. However, sampling could still be useful in most cases. For example, random sampling could be used to identify blocks that appear only to contain homogeneous space, i.e., blocks in which all values are identical. A non-sampled analysis of the data could then be performed on the subset identified by the sampled importance, which substantially could reduce the total computation time.

A key issue when calculating the importance is the time it takes to access data. The simulated region is typically allocated in contiguous memory space. However, a block makes up a 3D subset of the simulated area, which leads to low cache hit rates, especially for small block sizes. Using multiple importance metrics or metrics that utilize advanced data access patterns further complicates this issue. Some related work [4] has solved this issue by assuming the simulated region is allocated on a per-block basis. However, we feel that this is too limiting. Our solution is to dynamically allocate and deallocate a separate buffer for a block or a block sample when it is analyzed. All relevant importance analyses are then applied in sequence on a per-block basis, which leads to higher cache hit rates and a low memory overhead. Furthermore, this approach ensures that simulation data of each analyzed block only needs to be allocated and copied once, minimizing the computation time overhead introduced by this step. Operating on a separate buffer of a block's subset of the simulated area leads to higher cache hit rates, which means that multiple importance calculations can be performed at a lower computational cost.

A *pipeline* consists of a list of filters, which are executed in sequence. Filters contain *probes*, which are functions that analyze the data on a per-block basis. An overview of a pipeline's block importance analysis scheme is provided in Algorithm 1.

A filter contains zero or more probes, which are used to analyze each block in sequence. Each probe executed by a filter is assigned a custom weight (by default, 1). Given a block b, a filter f, and a set of probes for filter f, $\mathcal{P}_f$, the resulting importance of block b, i_b, after a filter has been applied is

$$i_b = \sum_{p \in \mathcal{P}_f} \frac{p(b)}{w_p}, \tag{2.1}$$

where w_p is the weight of probe p and $p(b)$ is the function by probe p to calculate the block importance of block b. A filter also contains an action, a condition, and a scope. An action specifies how a block should be handled during the data transfer stage. By default, a block has no specified action. However, a block's action can be changed by a filter to, for example, use a specific reduction strategy during the data transfer stage. Whether a block should assume a filter's action depends on if the block's importance fulfills the filter's condition. The condition returns either true or false depending on a block's importance; for example, a condition could specify $i_b > 0.8$, in which case all blocks with importance higher than 0.8 will assume the filter's action. The scope, a set of one or more actions,

Algorithm 1 Scheme used to calculate the importance of all blocks. The importance is used to determine the action of each block.

Input:
$\mathcal{B} = \{1, 2, \ldots, N\}$: set of all N blocks
$\mathcal{F}$: set of all active filters in pipeline
$\mathcal{W} = \{w_p\}$: set of weights, where w_p is the weight of probe p
Output:
$\mathcal{I} = \{i_1, i_2, \ldots, i_N\}$: set of importances, where i_b is the importance for block $b \in \mathcal{B}$
$\mathcal{A} = \{a_1, a_2, \ldots, a_N\}$: set of actions, where a_b is the action for block $b \in \mathcal{B}$

```
1:  for each block b ∈ B do
2:      i_b ← 0;                                  ▷ Importance initially set to zero for block b
3:      a_b ← NONE;                               ▷ Action initially set to NONE for block b
4:      for each filter f ∈ F do
5:          if a_b is in the scope of f then      ▷ Each scope is a set of zero or more actions
6:              for each probe p of filter f do   ▷ p ∈ { Mean, Range, SD, AVGSEQ, Distinct, Entropy }
7:                  if p analyzes a sample which is not allocated then
8:                      allocate sample of block b;
9:                  if p analyzes a block which is not allocated then
10:                     allocate block b;
11:                 i_b = i_b + p(b)/w_p;         ▷ p(b) calculates the importance for probe p on block b
12:             if i_b satisfies the condition of filter f then
13:                 a_b ← action of filter f;     ▷ a_b ∈ { NONE, Skip, RLE, LZ77, HOMO }
14:     Free the allocated memory of block b;
15: return I, A;
```

determines which blocks should be processed by a specific filter. For a filter to be applied, the block's action needs to match one of the scope's actions. This structure makes it possible to target specific subsets of blocks to perform further analysis and importance calculations.

Data is analyzed by using probes. Probes are short functions that analyze and output the importance of each block, based on the full content or a sample of a block. That is to say, the output of a probe calculating the mean value of a block is computed by iterating through each element in the block and then calculating the average value. In our testing, six different probes were used. The six probes are defined as follows:

- **Mean.** Calculates the mean of all values in a block.
- **Range.** Calculates the range of the values in a block.
- **SD.** Calculates the standard deviation of the values in a block.
- **AVGSEQ.** Calculates the average sequence length of identical values in a block. This probe is only used on non-sampled data, as sampled data would not retain enough information about the average sequence length.

- **Distinct.** Calculates the number of distinct values compared to the total number of values in a block.
- **Entropy.** Calculates the entropy of a block [4].

2.3.2 Block Actions

Block actions consist of three stages: initialization, pre-processing, and post-processing. Block actions can act as tags for a wide variety of acceleration purposes and can be seen as the action that has been taken for a specific block, based on its importance and the used filter conditions. However, we consider actions as tags for different types of compression methods. The initialization stage can be used to initialize resources and to specify the estimated compressed data size. Such information could be used to, for example, improve load balancing. The pre-processing and post-processing stages can be used to compress and decompress a block's data, respectively. Five different actions are used in our testing:

- **No Action (NONE).** No reduction or compression is performed. Blocks have this action set by default. This action is mainly useful for blocks where the time overhead introduced by compression outweighs the speedup of data transfers.
- **Skip.** Blocks with this action are never allocated or sent to the transit nodes. This action is useful if, for example, a region of the simulated grid is not of interest. Using the Skip action can, as such, substantially reduce the data transfer and co-processing times in some scenarios.
- **Run-Length Encoding (RLE).** Blocks with the RLE action are compressed using RLE. Both the compression and decompression of data can be completed in one pass.
- **LZ77.** Blocks with this action are compressed using the LZ77 compression algorithm [48]. Compared to the RLE compression method, LZ77 can be used to more effectively compress repeating sequences of data. This means that its usefulness differs depending on the simulation as well as on each block.
- **Homogeneous (HOMO).** Blocks that are Homogeneous are allocated as a single value. Similar to the Skip action, it can dramatically reduce the data transfer time. This action is useful for empty or homogeneous space, and can also be used to reduce regions that are not of interest. The main benefit that the Homogeneous action has over other types of compression and reduction algorithms is the compression time, which has a time complexity of $O(1)$.

We note that although these compression and reduction methods are used in the context of this chapter, the proposed approach can be used in combination with any existing

compression and reduction methods, including lossy methods like zfp [46], SZ [49], and its extension [47].

2.3.3 Adaptive Condition Window

The time required to perform importance calculations and data transfers largely depends on the filters and the target block actions, which are based on the specified filter conditions. Setting static filter conditions, i.e., each condition has a non-changing, constant value, could work in some situations. However, simulation data is rarely the same between any two time steps. As such, the execution time and memory usage could vary substantially. Instead, it could be preferable to adaptively change the parameters of the filter conditions based on some criterion. This criterion could be based on, for example, the execution time, memory usage, or the remaining allocated time on a compute cluster. We refer to this kind of filter condition as an *adaptive condition*.

In our approach, we have to consider multiple filters, all of which could include adaptive conditions. The key issue is how to adaptively modify the conditions without affecting the intended flow of the analysis. Our solution is to use an adaptive condition window, ω, by which multiple adaptive condition values can vary predictably. The condition window is a value that slides between 0.0 and 1.0, at 0.05 intervals. The condition window can slide one interval towards 0.0 or 1.0 after each time step. All filters have a defined range (upper bound u and lower bound l) for their condition value. The filter condition values are reevaluated after each executed time step based on input to the program, using the expression

$$l + (u - l) \cdot \omega. \tag{2.2}$$

The upper and lower bounds can be set to the same value, in which case the condition value of that filter is constant.

For example, a researcher could determine that as much important data as possible should be saved to permanent storage after each time step. However, there is a strict time limit for the length of the I/O operation. The researcher would first decide on probes and filters that accurately can identify important data for the specific use case. The l and u variables would then be set to 0.0, 1.0, respectively. Initially, ω is set to 0.0. However, after each time step, ω can increase by 0.05 based on if the I/O time exceeds the specified time limit. Over time, ω moves to the highest value possible such that the I/O operation does not exceed the time limit. As a result, the researcher can maximize the amount of data that can be stored for each time step. This behavior further scales to work with multiple probes and filters, meaning that many decisions can be made about different aspects of the simulation data.

The adaptive condition window ensures that the condition values of all filters can be modified in a controlled manner, thus retaining the intended analysis flow. One condition

window is used per pipeline, meaning that different condition windows can be used for different data sets if required.

2.3.4 Advantages of the Pipeline Structure

The proposed structure facilitates pipeline construction in practical applications. Specifically, the pipeline has three key advantages compared to alternative structures (e.g., a directed acyclic graph (DAG) or a decision tree approach [50] that calculates an importance metric on each branch node):

1. **Ease of use**. Using the structure of the proposed pipeline, each component (e.g., a filter or probe) is categorized and serves a clear purpose. In practice, it is easy to construct and understand the structure of the proposed pipeline. In contrast, some structures (e.g., decision trees) would not be intuitive without a visual interface.

2. **Gradual refinement**. A new action can be assigned to a block after each filter has been applied. As filters are applied in sequence, this behavior enables a gradual refinement of the importance analysis. More advanced or time-consuming analyses can be limited to the relevant subsets of the simulation data. Although this behavior can be mirrored by other structures (e.g., a DAG or tree structure), it would be more complicated to create.

3. **Reusability**. It is easy to reuse parts of a pipeline (e.g., filters or probes) in other applications as all parts of the pipeline are compartmentalized.

2.3.5 Data Distribution

Load balancing blocks of 3D data sets has been the focus of extensive research [4, 22–26, 31]. A load balancing technique can either create a static distribution of the data or dynamically change the distribution based on some variables. Typically, dynamic load balancing techniques strive to minimize either the difference in computation time or the difference in memory usage on each process.

Scientific simulations often generate large quantities of data, meaning that memory usage is of primary concern. Therefore, we currently consider a static and a dynamic different load balancing technique. The static technique is based on the initial distribution in a k-d tree [27], which distributes blocks to all processes on the transit nodes. However, some blocks may have been removed using the Skip action. As such, using this technique could result in a memory imbalance on the transit nodes. The dynamic technique partly resolves this issue by rebalancing the k-d tree. Let $\mathcal{B}_t$ be the list of blocks on transit process t, μ the average number of blocks per process, and n the number of transit processes. The

goal is then to minimize

$$\sum_{t=1}^{n} \sqrt{\frac{|\mathcal{B}_t| - \mu}{n}}, \tag{2.3}$$

i.e., the difference in allocated memory on each process on the transit nodes.

Blocks sent to a specific process on a transit node make up a contiguous and convex subset of the original 3D data set. They can, as such, be decompressed in parallel and combined to reconstruct a single region in 3D space. This step is essential to perform certain types of analyses and visualizations efficiently. For example, many volume rendering engines utilize internal block structures to perform empty space skipping [51] and optimization techniques such as early ray termination [52, 53]. Prematurely partitioning the 3D data set into multiple blocks can negatively affect the effectiveness of such techniques.

At the start of the distribution stage, we utilize *distributed filters* to apply filters that require information about blocks from multiple processes. For evaluation, only one distributed filter is used, Filter Borders. The Filter Borders distributed filter attempts to identify blocks of a specific action on the border of the 3D data set. If a slice of border blocks can be identified, their action is changed to the target action specified by the filter. For example, this distributed filter can be used to identify slices of homogeneous blocks along the border of the 3D volume, and then exclude them from the in-transit rendering process by changing the action to Skip.

2.3.6 In-Transit Co-Processing

Blocks transferred to processes on transit nodes might have been reduced or compressed during the in-situ stage. To perform any visualization or analysis of the data, the blocks first need to be decompressed. Blocks are decompressed based on the used compression algorithms. Because blocks on a specific process make up contiguous and convex regions in 3D space they can also easily be aggregated into a single block, which significantly can improve the performance of the co-processing process.

The content of the co-processing is not part of our proposed approach. Instead, the approach only handles data decompression and data reconstruction on the transit nodes. Analysis and visualization can be performed in a normal fashion according to the researcher's needs, without any need to integrate existing tools and software with our approach.

2.4 Experimental Evaluation

To evaluate the proposed method and the approach to accelerate in-transit co-processing, we have run a series of tests on two different compute clusters, cluster A and cluster B

Table 2.2: Test environments. Each cluster system has Infiniband EDR as its interconnection.

	Cluster A		Cluster B	
	Simulation node	Transit node	Simulation node	Transit node
CPU	Xeon	Xeon Silver	Xeon Gold	Xeon Gold
	E5-1650 v4	4110	6126 ×2	6126 ×2
	6 cores	8 cores	12 cores ×2	12 cores ×2
Memory (GB)	128	96	192	192
Node count	16	2	32	4
Processes	16	2	64	8
Software	GCC version 7.3.0		ICC	
	OpenMPI version 3.1.0		Intel MPI version 18.0.3	

(Octopus) [54], at different resolutions using up to 864 cores. Information about the test environments is detailed in Table 2.2.

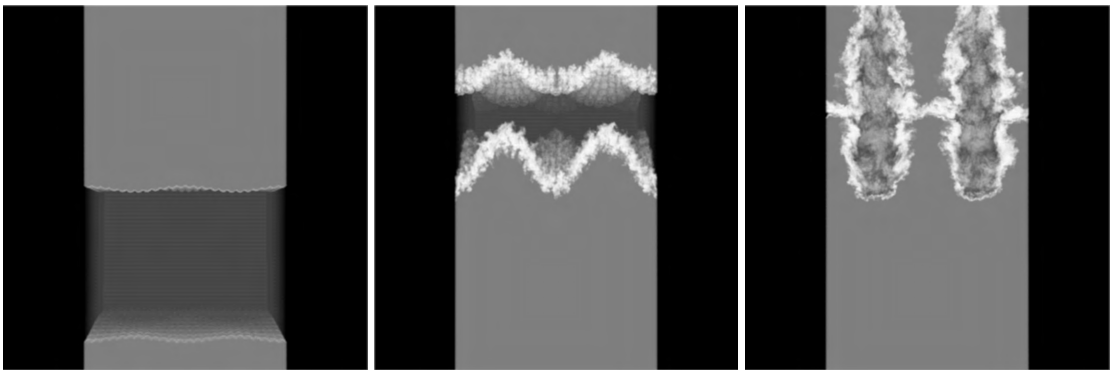
2.4.1 Experiment Description

We used the RMI simulation data to evaluate the performance of our proposed approach. The RMI test-case set-up, as well as numerical methods considered, are similar to previous work [18, 19]. However, here, the effect of the membrane mesh separating the two gases is modelled in the simulations according to well-defined modes combined with random perturbation components [1, 55].

To mimic a realistic co-processing scenario, we used up to three different variables in each analyzed time step: the mass fraction (MF) and two axes of the momentum (MY and MZ). The data of these variables were analyzed, compressed, and transferred to the transit nodes using the proposed approach. On the transit nodes, data decompression and data reconstruction were also performed. However, no additional co-processing was included as part of the testing.

A grid resolution of $1601 \times 401 \times 801$ was used on cluster A, resulting in 11.5 GB of data spread over three variables, each time step. Similarly, the grid resolution on cluster B was set to $2401 \times 601 \times 1201$, resulting in 38.7 GB of data for the three analyzed variables.

We chose to perform co-processing for time steps at a 5e-5 second interval (simulated time). In practice, this amounts to approximately 1 out of every 127 and 1 out of 193 time



(a) (b) (c)

Figure 2.2: Visualization of the MF data set generated by the RMI simulation at time (a) 0 ms, (b) 2 ms, and (c) 4 ms; blue represents pure air, red represents pure sulfur-hexafluoride (SF_6), while yellow represents the binary mixture comprising of the two miscible components.

steps, respectively. The simulations ran for 10,292 time steps on cluster A and 15,653 time steps on cluster B, out of which 81 were analyzed. That is to say, up to 0.91 TB (cluster A) and 3.06 TB (cluster B) of data were analyzed. It took a total of 117.5 hours on cluster A and 100.9 hours on cluster B to run the simulations.

The data was loaded into memory as 64-bit floats during the testing. Figure 2.2 shows a visualization of the MF data set, using the OSPRay rendering engine [37], version 1.7.3.

The proposed approach can utilize a combination of existing compression methods, and its performance depends on the performance of the used methods. We, therefore, chose to compare the approach to uniformly applying each compression method used by the approach on the full data set. Those methods consist of RLE, LZ77, and HOMO, as specified in Section 2.3.2.

2.4.2 Block Size

Analysis operations are heavily dependent on the size of each block. Appropriate block sizes for volume rendering have been investigated in related work [22], which found that blocks with a $64 \times 64 \times 64$ resolution achieved the best result in their use case. A similar block resolution has been used in research related to this work [4], although with no motivation. We analyzed five resolution to determine an appropriate block resolution for our use case: $50 \times 50 \times 50$, $50 \times 50 \times 100$, $50 \times 100 \times 100$, $100 \times 100 \times 100$, and $100 \times 100 \times 200$.

Tests were performed on cluster A using the MF data set and a Mean probe. As per

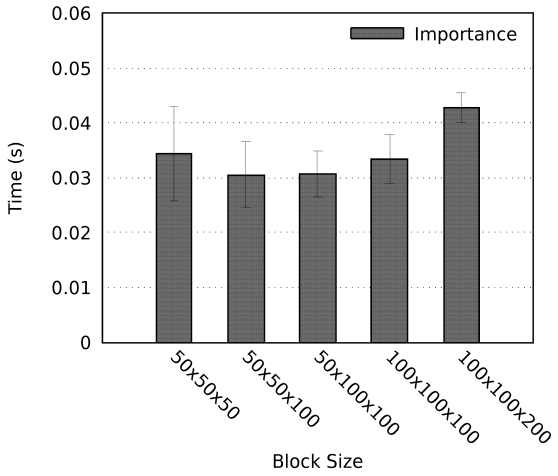


Figure 2.3: The average time required to perform the importance analysis for the MF data set on cluster A. The error bars display the standard deviation of the computation times.

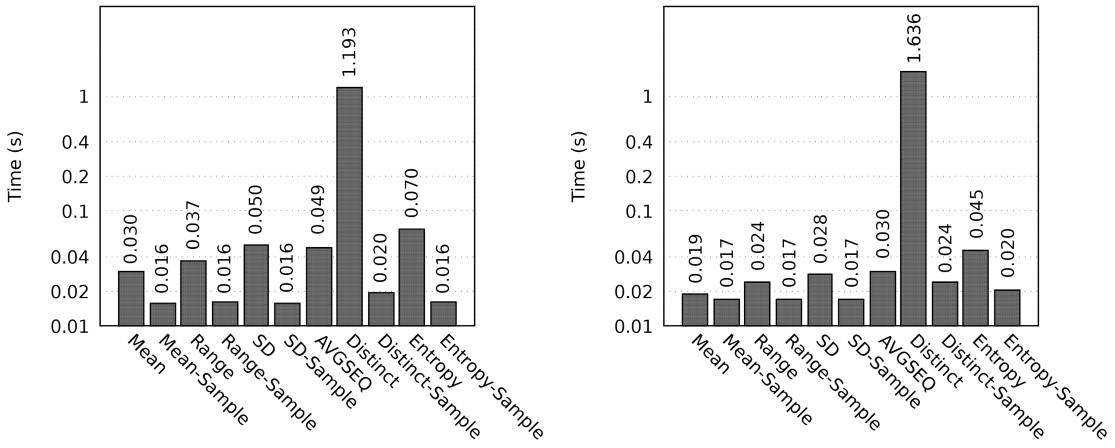
the results, shown in Fig. 2.3, a block size of $50 \times 50 \times 100$ or $50 \times 100 \times 100$ achieved the fastest computation time. We chose to use the block size of $50 \times 100 \times 100$ in all successive tests on cluster A presented in this chapter. The simulation resolution and the number of processes on cluster B differ from that of cluster A. As a result, it was not possible to use the same block size. Instead, we set the block size on cluster B to $75 \times 75 \times 75$, which was the closest alternative.

2.4.3 Performance of the Proposed Method

The computation time varied between different probes, as shown in Fig. 2.4. For sampled probes, we used a block sample size of 2390 and 2387 for the two respective resolutions. All sampled probes were consistently faster; the best example being the Distinct probe, which on average took 1.19 seconds to complete on cluster A. In comparison, the sampled Distinct probe only required 0.02 seconds of computation time (a speedup of 61.14). Similar results were obtained on cluster B.

We observe that the sampled probes generally were able to achieve more significant speedups on cluster A. This is primarily because of two reasons. First, simulation processes on cluster A had to process more data than the processes on cluster B. Second, simulation processes on cluster B could more efficiently calculate the importance due to having access to more cores. Calculating the importance of sampled blocks is limited by the time required to complete the initial data access, which is similar on both clusters.

As discussed in Section 2.3.1, one of the goals of the proposed method was to minimize the computation overhead of utilizing multiple probes. As seen in Fig. 2.5, the compu-

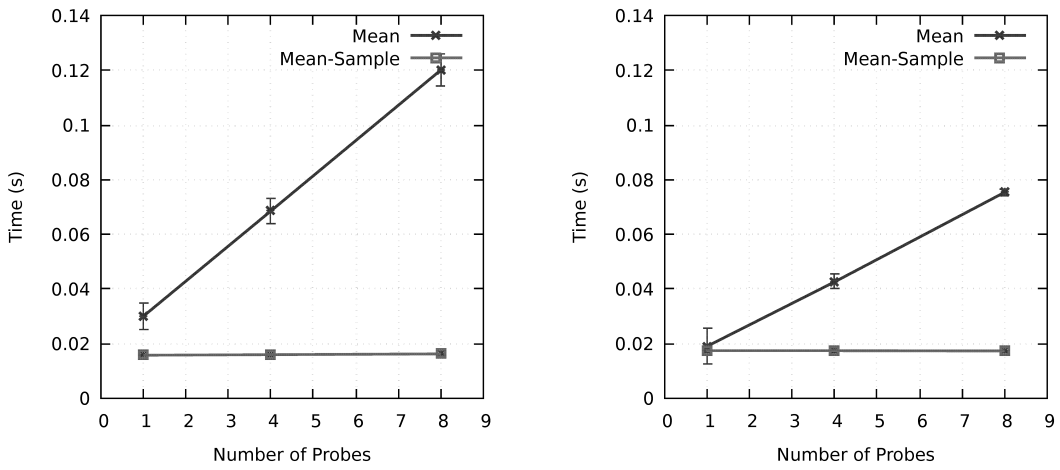


(a) (b)

Figure 2.4: The importance calculation times on (a) cluster A and (b) cluster B using the MF data set. The y-axis has a logarithmic scale.

tation time increased linearly when using multiple non-sampled Mean probes. However, increasing the number of probes from one to four and eight only increased the computation time on cluster A by 2.28× and 4×, respectively, and 2.24× and 3.96× on cluster B, as compared to the expected 4× and 8×. Similarly, the increase was only 1% and 3% for the sampled Mean probes on cluster A, whereas all results were within the margin of error on cluster B. By preprocessing the block data and using the block-based importance calculation process described in Section 2.3.1, the importance calculation time is not directly proportional to the number of used probes. This is especially the case for sampled probes, where the increase in computation time was negligible in our tests. The initial data reorganization required to achieve this performance results in some overhead, increasing the computation time for a probe with a simplistic data access pattern. However, as seen in Fig. 2.5, the performance is better when using multiple probes or probes with more advanced data access patterns as a result of the improved cache hit rate. The performance of our method is directly affected by the data access pattern of the used probe(s). Intuitively, probes with advanced data access patterns should benefit more from the higher cache hit rates. A one-pass algorithm, such as the Mean probe, should as such represent a worst-case scenario.

The accuracy of the sampled probes depends on the sample size as well as the used probe algorithm. Table 2.3 shows the absolute error, E, of the sampled probes using the MF data set on cluster A at time step 80. E is equivalent to $x_i - x$, where x_i and x are the measured and true values, respectively. Using the Mean or SD sampled probes resulted



(a) (b)

Figure 2.5: The average importance calculation times using up to eight Mean probes on (a) cluster A and (b) cluster B for the MF data set. The error bars display the standard deviation of the computation times.

in all blocks having an importance value within 0.01 of the correct importance value. However, The Range and Distinct sampled probes were not as accurate as the Mean, SD, and Entropy probes. In some scenarios, the calculated importance of these probes depends on a small subset of the data. For example, by changing a single data value in some blocks, the importance calculated by the Range probe can increase from 0 to 1.

In summary, the importance analysis scheme could successfully improve the performance when using multiple probes. Sampled probes could achieve a high accuracy and further accelerate the importance analysis computation.

2.4.4 Evaluating the Proposed Approach

In this section, we evaluate the performance of the approach used to accelerate in-transit co-processing.

Compression Performance

Intuitively, different probes should reflect different information about the underlying block data. It should then be possible to use this information to, for example, select the best compression technique for a specific use case. Figure 2.6 shows the relation between the RLE-, HOMO-, and LZ77-compressed block sizes and the importance calculated by the used probes. The Distinct probe provided an almost linear relationship between the

Table 2.3: Accuracy of the sampled probes. Percentage of all sampled block importance values which absolute error is less than 0.001, 0.005, 0.01, 0.05, and 0.1. Tests were run on cluster A, using the MF data set and time step 80. The AVGSEQ probe is not included, as it is only used on non-sampled data.

	<1e-3	<0.005	<0.01	<0.05	<0.1
Mean	71.9	98.5	100	100	100
Range	72.9	75.0	76.4	86.4	95.1
SD	71.9	99.4	100	100	100
Distinct	51.2	60.1	63.9	82.2	98.3
Entropy	50.2	56.4	61.3	85.4	99.7

importance and the compressed data size, as seen in the figure. Similarly, the AVGSEQ and Entropy probes had clear connections between their calculated importance and the compressed block sizes.

The compressed data size achieved by using RLE compression was generally smaller than when using LZ77. This trend was caused by the entropy of the used data set, which does not contain many repeating sequences of values. Similarly, the effectiveness of the RLE compression is linked to the many homogeneous regions in the investigated data set. However, as seen in Fig. 2.7 and Fig. 2.8, the performance of the various compression methods vary throughout the simulation. The initial data set, at time step 0, contains many homogeneous regions. Consequently, the initial compressed data size is minimal. As the simulation progresses, the homogeneous regions become smaller and disappear, which increases the compressed data size. The total compressed data size remains lower than that of the uncompressed data throughout the simulation. However, as seen for the RLE and LZ77 methods in Fig. 2.6, compressing some blocks actually increases the block size (an uncompressed block has a size of 3906 kB on cluster A). It is better to not use these compression methods when transferring such blocks. This further reinforces our notion of selectively choosing which compression method to use on a per-block basis for each time step.

Assessing the Execution Times of the Proposed Approach

Pipelines should be devised based on the needs of the researcher and the nature of the used simulation. It is not possible to create a pipeline that is optimal in all scenarios. Furthermore, which compression methods to use also depend on the specific use case. We designed two advanced pipelines: one lossless and one lossy that uses a condition window

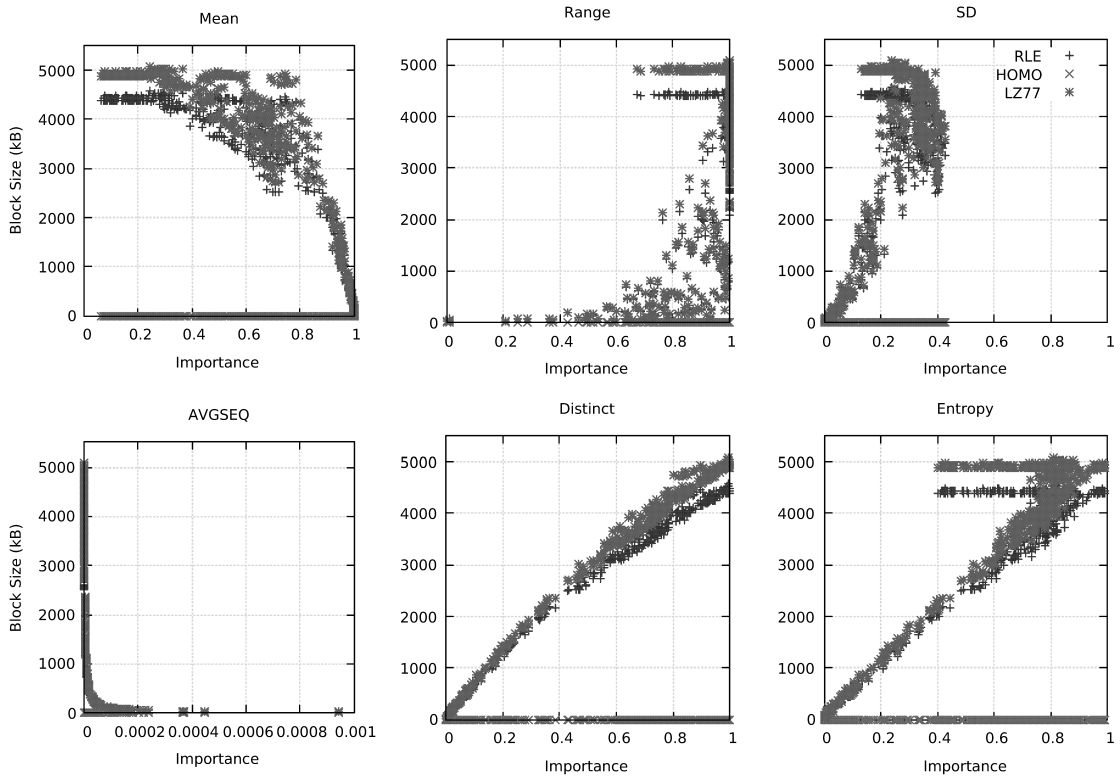
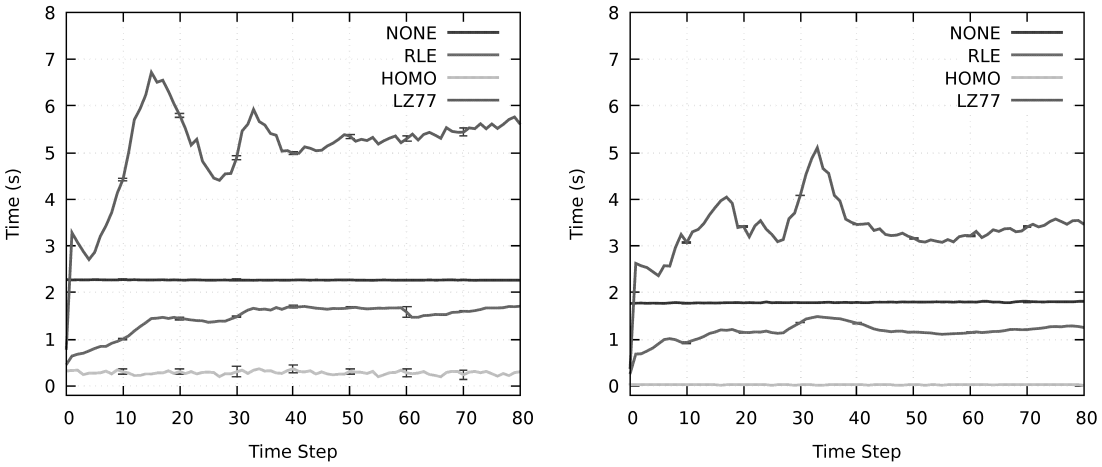


Figure 2.6: The compressed block sizes compared to the calculated importance for the MF data set using the Mean, Range, SD, AVGSEQ, Distinct, and Entropy probes. Tests were conducted on cluster A at time step 80.

to reduce the least important data adaptively. Both pipelines used a distributed filter to remove homogeneous border blocks (i.e., blocks which are not of any significance to the analysis of the simulation data). Furthermore, the pipelines were evaluated using both no load balancing (static) and a k-d tree load balancing technique (kd) to equalize the data distribution after the use of the distributed filter.

Pipeline 1 is structured as follows. Initially, all blocks are sampled using the Range probe. Blocks with an importance of 0 have their actions set to RLE. A non-sampled Range probe is then used on all blocks with action RLE. If the importance is 0, the action is set to Homogeneous. By initially using a sampled probe, the non-sampled probe only has to operate on a subset of all blocks, which reduces the computation time. Finally, the third filter uses a sampled Distinct probe to change the action of blocks with the NONE action and an importance ≤ 0.9 to RLE. The value 0.9 is based on previous experiments to evaluate the compressed block sizes, displayed in Fig. 2.6. Using the Distinct probe, having importance lower than approximately 0.9 resulted in a lower block size than the



(a) (b)

Figure 2.7: The total compression, load balancing and data transfer times on (a) cluster A and (b) cluster B, using the MF, MY, and MZ data sets. The error bars display the standard deviation of the computation times.

original non-compressed data. Intuitively, blocks with many distinct values might also be more time-consuming to compress and decompress, further increasing the data transfer time. We chose to use a sampled version of the Distinct probe to accelerate the in-situ computation time. Note that the resulting compression of Pipeline 1 is lossless.

The idea behind Pipeline 2 is to reduce the most unimportant data (in this case, the blocks containing the lowest amount of distinct values) on the fly to achieve a total compressed data size of between 1200 to 1600 MB (cluster A) and 4000 to 5400 MB (cluster B) for each data set (i.e., using all three data sets, up to 4.69 GB and 15.82 GB of data for each time step on clusters A and B, respectively). We use a range rather than a specific threshold to reduce the volatility of the compressed data size between each time step. The initial three filters of Pipeline 2 are identical to those of Pipeline 1. However, Pipeline 2 includes a fourth filter that utilizes a condition window. The filter uses a Distinct sampled probe and targets blocks with the RLE or NONE actions. The condition value range is set as [0, 1.0], with an initial value of 0. If the condition of the filter is fulfilled, the action is changed to Homogeneous.

The data size of homogeneous blocks using the Homogeneous compression method was almost the same as when using RLE compression. This can be seen in Fig. 2.9, where the compressed data size of Pipeline 1 is almost identical to when utilizing an RLE compression. However, Pipeline 2 achieved a significantly lower data size because of the additional adaptive (lossy) reduction filter. Noteworthy is that Pipeline 2 never reached

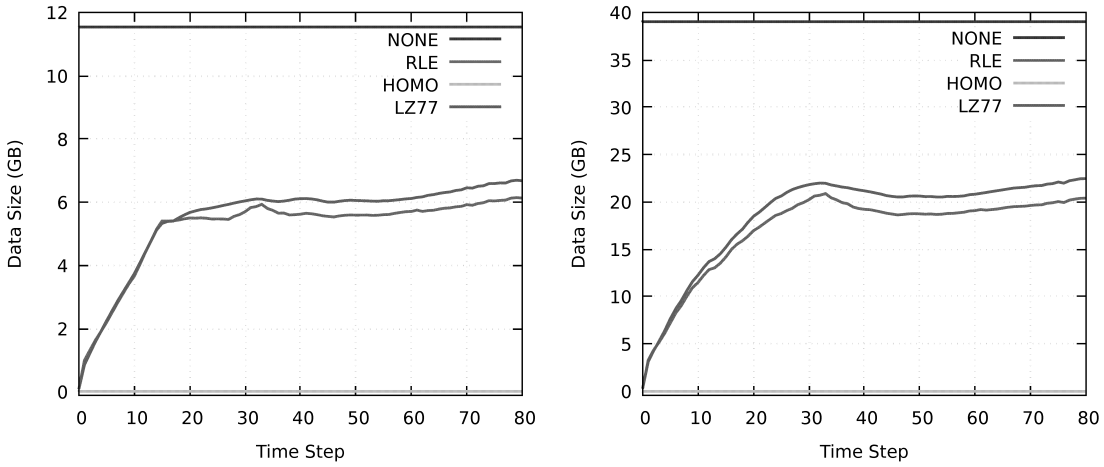


Figure 2.8: The size of the compressed data transferred during the distribution stage on (a) cluster A and (b) cluster B, using the MF, MY, and MZ data sets.

the upper limits of 4.69 GB and 15.82 GB. This is because the compressed data size of at least one data set remained below the specified upper data size threshold.

The impact of the proposed approach on the entire simulation time widely depends on the used simulation and co-processing. For example, the simulation time of a single time step, detailed in Section 2.4.1, and the in-transit co-processing time can vary by orders of magnitude depending on the simulation, the specified simulation parameters, and which types of analysis and visualization tasks are performed during the co-processing stage. Furthermore, co-processing could be performed for each simulated time step or for a small fraction of time steps. As such, we chose to focus the evaluation exclusively on the actual performance of the three main stages of the proposed approach, i.e., the co-processing stage only consists of data decompression and data reconstruction. The average execution times of each stage are shown in Fig. 2.10. The importance calculation of the RLE, HOMO, and LZ77 tests used for comparison consisted of a single Mean probe.

Out of the lossless compression methods used for comparison, RLE achieved the best performance. For the total execution time, the RLE compression method achieved a speedup of 1.26× (cluster A) and 1.3× (cluster B) compared to using no compression. The lossless Pipeline 1 achieved an even more significant speedup; 1.62× on cluster A and 1.55× on cluster B. Compared to RLE, Pipeline 1 could speed up the execution time by up to 1.29× and 1.19× on the two respective clusters. Similarly, Pipeline 2, which reduces unimportant data, achieved total speedups of 1.69× and 1.52× compared to the RLE method. Table 2.4 summarizes the average speedups of the total execution time for

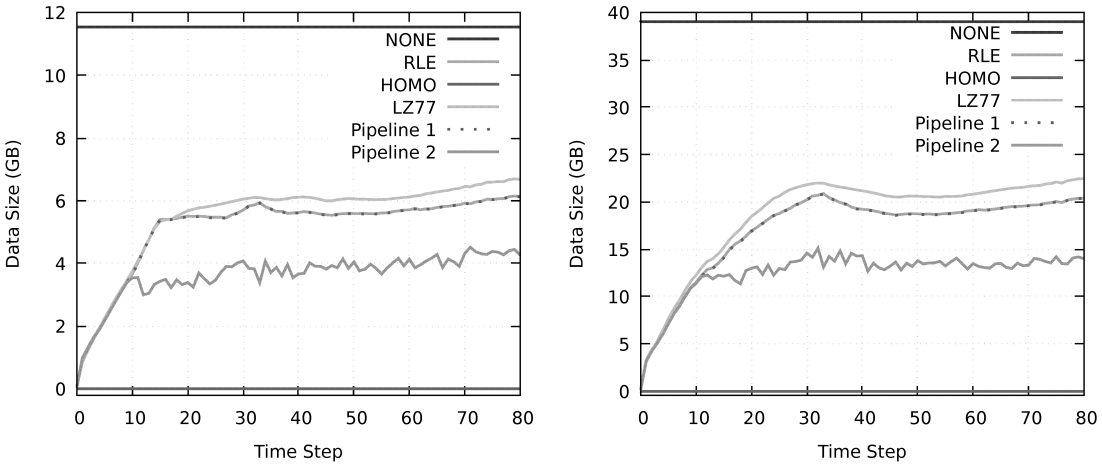


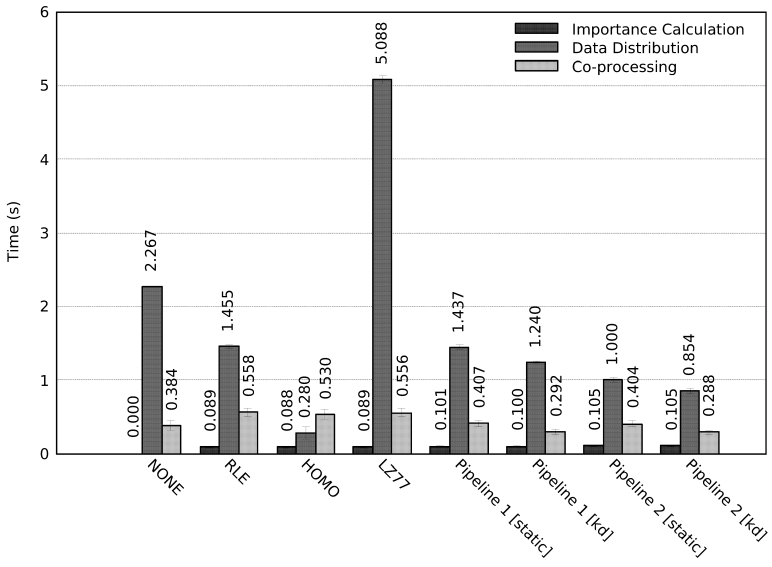
Figure 2.9: Size of the transferred compressed data using the MF, MY, and MZ data sets on (a) cluster A and (b) cluster B. The results of the RLE and Pipeline 1 tests overlap.

Table 2.4: Relative speedups of the total execution time for Pipeline 1 and Pipeline 2 compared to the other evaluated methods.

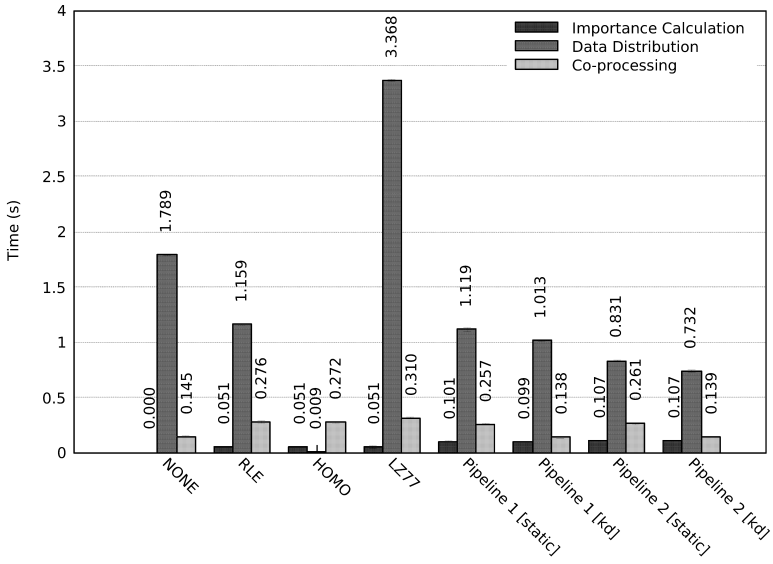
	Cluster A				Cluster B			
	None	RLE	HOMO	LZ77	None	RLE	HOMO	LZ77
Pipeline 1	1.62	1.29	0.55	3.51	1.55	1.19	0.26	2.98
Pipeline 2	2.13	1.69	0.72	4.60	1.98	1.52	0.33	3.81

both pipelines compared to the other evaluated methods.

In more detail, both Pipeline 1 and Pipeline 2 achieved significantly better performance for the data distribution and co-processing stages than the NONE, RLE, and LZ77 methods. The execution time of the co-processing stage for Pipeline 1 was similar to the RLE test when using static load balancing. The uneven load caused this; although multiple transit processes achieved significantly lower computation time than when using only RLE compression, the total execution time is dependent on the slowest process. Using a k-d load balancing technique solved the uneven load on the transit nodes. On cluster A, Pipelines 1 and 2 achieved speedups of 1.91× and 1.94× of the co-processing stage as compared to using RLE compression, and speedups of 1.32× and 1.33× compared to using no compression. Similarly, on cluster B the two pipelines achieved speedups of 2.00× and 1.99× as compared to an RLE compression. Compared to using no compression, the



(a)



(b)

Figure 2.10: Average time step execution time for the three main stages using all three data sets on (a) cluster A and (b) cluster B. The error bars display the standard deviation of the computation times.

speedup of the co-processing stage was 1.05× and 1.04× for the two respective pipelines.

Utilizing dynamic load balancing significantly improved the execution times of the co-processing stage. However, the load balancing only determines where data is sent. The compression time and the amount of data that needs to be transferred from each simulation process do not change. Using a dynamic load balancing technique did not improve the execution time of the data distribution stage to the same extent. On cluster A, the two pipelines achieved speedups of 1.17× and 1.70× as compared to an RLE compression and speedups of 1.83× and 2.65× compared to using no compression. On cluster B, the pipelines achieved speedups of 1.14× and 1.58× compared to RLE compression and speedups of 1.77× and 2.44× compared to no compression.

Interestingly, the lossless Pipeline 1 was able to achieve better performance in all aspects (the compressed data size and the compression, data transfer, and decompression times) than the other lossless compression methods. This results from the fact that both of the pipelines scale with the utilized compression methods; in this case, using NONE, RLE, and HOMO. This scaling can be seen in Fig. 2.9, where the compressed data size of Pipeline 1 closely follows that of the RLE compression. This behavior should extend to other compression methods as well.

2.5 Conclusion

We have presented a method to efficiently determine the importance of regions of interest of simulation data, with an emphasis on using multiple importance metrics. In addition, we have presented a use case where the method was used to accelerate the in-transit co-processing of an RMI simulation by lowering the data transfer time. The approach to accelerate in-transit co-processing uses the importance of regions of interest to determine how to best combine the usage of multiple different compression methods on different subsets of the simulation data. Simulation data is analyzed to determine the importance of all regions of the 3D data sets, which is then used as a basis to utilize multiple compression and reduction methods adaptively.

We have evaluated the performance of the proposed method and approach by conducting tests on two different compute clusters, using multivariate data from an RMI simulation. Our proposed method was able to expeditiously calculate block importance, even when multiple data probes were used. The above was especially the case for probes using sampled data, which could calculate accurate importance values at a much faster rate. The proposed method was also able to adaptively identify regions of important data in a reliable manner. The excellent scalability when using multiple data probes and low computation times also make the method viable to be used in many other in-situ and in-transit scenarios, such as: guiding the simulation; saving important data to permanent storage; or in tandem with additional analysis or visualization software. As for the approach, we were able to achieve better performance in all aspects (the compressed data size and

the compression, data transfer, and decompression times) than the compression methods used for comparison. Compared to an RLE compression, using the proposed approach in a lossless scenario resulted in a speedup of up to 1.29× for the overall execution time, and 2× when performing data decompression. We conclude that the proposed approach significantly can accelerate the in-transit co-processing process.

In future work, we will investigate how the proposed method can be used for other use cases. Furthermore, we plan to extend the proposed approach to work better with in-situ co-processing workflows.

Chapter 3

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

3.1 Introduction

The capabilities of modern supercomputers enable the simulation and visualization of large-scale data sets with high precision and detail. Data sets generated by scientific simulations, often multivariate and spanning multiple time steps, can consist of terabytes of data. Using a sorting scheme called sort-last [30], 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 [56]. Partial images from all processes then need to be composed based on their position and distance from the camera in the volume [3].

The rendering times can vary between processes based on many factors, e.g., used optimization techniques or the characteristics of the data sets. If there are any substantial rendering time imbalances, dynamic load balancing techniques can be used to effectively reduce the total rendering time during in-situ visualization or post-hoc exploration. However, dynamically redistributing data can result in large memory imbalances between processes. Some processes might run out of memory when handling large data sets, making many dynamic load balancing techniques unsuitable for large-scale visualization [57, 58].

Commonly used dynamic load balancing techniques are based on tree structures, e.g., a k-d tree [27]. In the k-d tree structure, the original volume is represented by the root of the tree, as illustrated in Fig. 3.1. 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 at depth 1, 2, or 3, respectively. Data can be load balanced between children of the same branch in the tree, as shown in Fig. 3.1. 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 [22]. Utilizing this structure and ensuring that each process only renders contiguous data results in two positive aspects:

1. **A simple compositing order for partial images rendered by each process.** The k-d tree structure enables processes to composite all partial images generated by its blocks during the rendering stage, without any external communication. Next, the partial images on each process are composited in an inter-process compositing stage. Finally, the remaining partial images can be gathered and merged 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 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 [26]. This behavior should scale with the number of processes, meaning that it could be of greater concern in large-scale visualization. An example of a k-d tree memory imbalance is shown in Fig. 3.2, where the main computational load is focused on the upper quadrant of the volume. Equalizing the rendering 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 memory imbalance occurs limits the use of k-d tree-based dynamic load balancing in large-scale applications, where even small imbalances can result in some processes running out of memory [58].

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

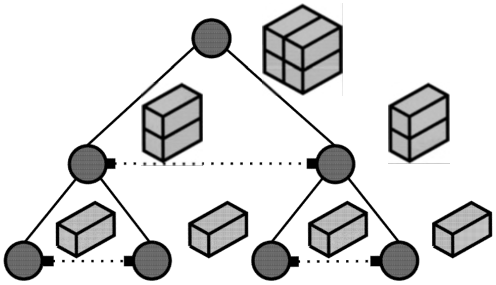


Figure 3.1: k-d tree data distribution and load balancing. Circles represent branches in the tree, whereas blocks are represented by cuboids, each of which results in a partial image when rendered. Branches that can load balance are connected via dotted lines.

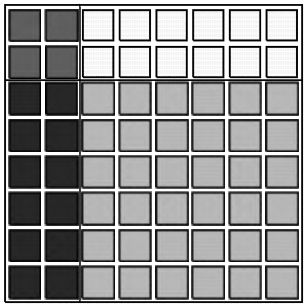


Figure 3.2: 2D representation of a k-d tree block distribution between four processes, in a worst-case scenario where the main computational load is focused on the top left quadrant of the volume. Volume blocks are represented as squares, whereas the color distinguishes different processes.

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 is possible to prioritize load balancing blocks with high rendering times. This would lead to a lower worst-case memory usage and fewer 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.

The contribution of this chapter is a novel compositing pipeline and a load balancing technique that 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 rendering time. A secondary goal is to lower the number 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 and a static distribution by conducting a series of experiments on a cluster using up to 32 processes, each of which has a dedicated graphics processing unit (GPU).

The structure of this chapter is organized as follows. Related work is presented in Section 3.2. The compositing pipeline and the load balancing technique are described in Section 3.3. The technique is then evaluated in Section 3.4. Lastly, our conclusions are presented in Section 3.5.

3.2 Related Work

k-d trees and similar tree structures have been used extensively in many related works to achieve dynamic load balancing [22–26]. The rendering time of the previous frame is often used as a load balancing heuristic [22, 23]. Commonly, uniformly-sized blocks are stored in the k-d tree [22, 25]. Other works have explored using variable block sizes to achieve finer granularity [23]. However, variable block sizes 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 [25].

Zhang et al. [26] 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. Constrains were introduced on the k-d tree data partitioning to sidestep this issue, 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. The authors note that the only way to ensure an optimal distribution is to allow processes to hold the complete volume in memory [26].

Utilizing dynamic load balancing techniques tends to result in an uneven data distribution among processes. Data sets may consist of multiple terabytes of data in large-scale applications; even small-scale data transfers can be time consuming and result in some processes running out of memory. As such, many large-scale visualization projects have utilized static load balancing [58, 59] or limited load balancing to equalizing the data distribution, rather than explicitly lowering the total rendering time [4, 24].

3.3 Two-Layered Dynamic Load Balancing Technique

To lower the memory usage and redundant data transfers 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 Section 3.2. 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.

3.3.1 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. Each process is 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 3.3 depicts the execution flow of the presented technique, whereas Fig. 3.4 shows an example where a block is load balanced to another process. As illustrated in Fig. 3.4b, 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. 3.4c), 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 needs to be composed during a final inter-process compositing step (Fig. 3.4d). 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.

3.3.2 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.

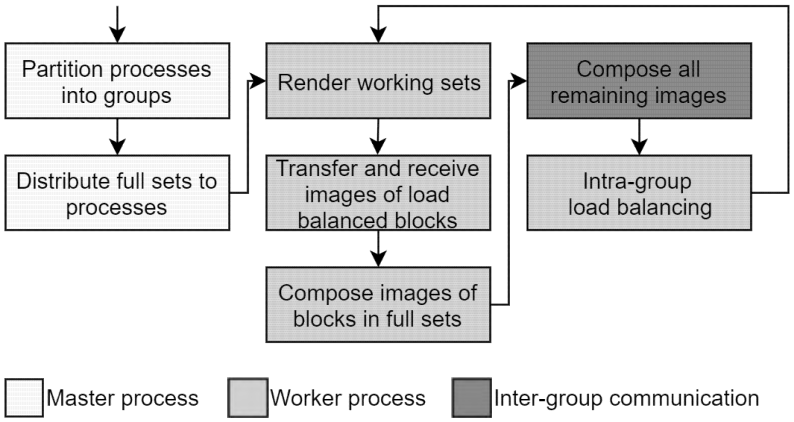


Figure 3.3: Program execution flow of the two-layered group technique. Initially, processes are partitioned into groups and the full sets are distributed to processes. The rendering stage and an initial compositing step are then performed by each process. The remaining images are composed in an inter-group fashion in a second compositing step. Lastly, load balancing is performed within each group before the next frame can be rendered.

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. 3.4a–c) have in such a scenario no inter-group dependence, meaning they can be performed in parallel within each group. By limiting the number 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 number of processes in each group remains constant. An example group structure is displayed in Fig. 3.5.

3.3.3 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

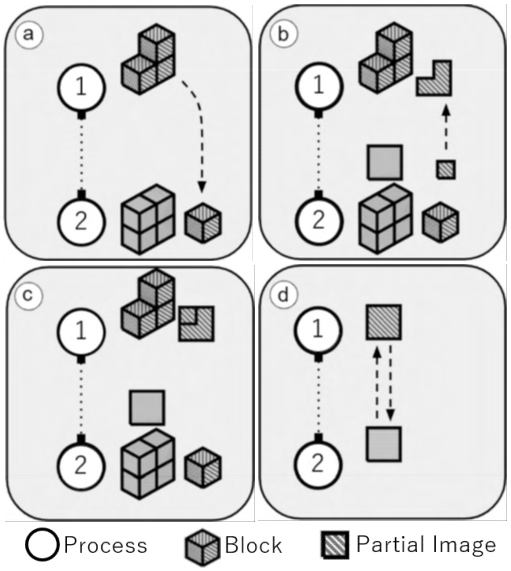


Figure 3.4: Example of how a load-balanced block is rendered and composed. (a) A block is load balanced between two processes in the same group. The load-balanced block makes up a new working set on the receiving process. (b) Each process renders its working sets, starting with blocks belonging to other 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 its full sets. (d) Inter-group image compositing takes place to compose the final image.

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. 3.2 is shown in Fig. 3.6.

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. 3.7. 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 the end of the rendering stage.

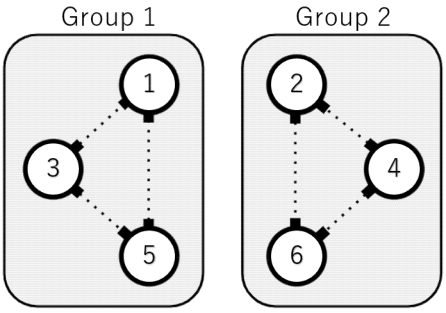


Figure 3.5: An example structure containing two groups. Processes are partitioned into groups in a round-robin fashion. Processes within the same group can freely perform load balancing amongst each other, as illustrated by the dotted lines.

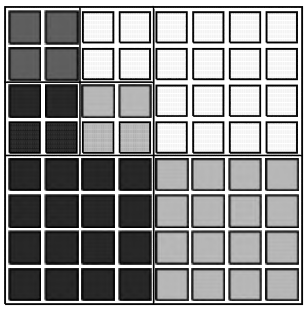


Figure 3.6: 2D representation of a block distribution between four processes using the two-layered group technique, in a worst-case scenario where the main computational load is focused on the top left quadrant of the volume. Volume blocks are represented as squares, whereas the color distinguishes different processes.

After each frame, the average rendering times are calculated in each group. Processes of which the rendering time deviates from the average are sorted into one of two lists depending on if the rendering time is lower or higher. Historical data transfers and current process rendering times are then used to dictate which processes in the two lists perform load balancing. This functionality helps reduce the unnecessary spread of blocks, resulting in fewer image transfers. The load balancing algorithm is shown in Algorithm 2. 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 rendering 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

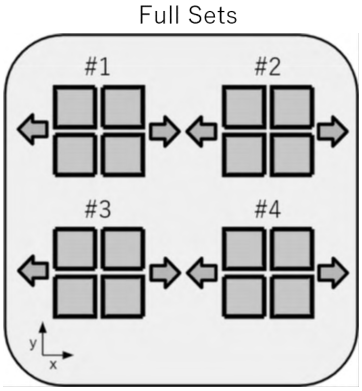


Figure 3.7: Slices of blocks can be load balanced from both directions of the x-axis. However, load balancing for each full set can only be performed with a single process in each direction. Having four full sets on each process means that blocks can be delegated to eight other processes simultaneously.

first operation of Algorithm 2, each process that has a lower-than-average rendering time recalls a previously load-balanced slice of blocks, if possible. Similarly, in the second operation, each process with a higher-than-average rendering 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 rendering 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 rendering time. Finally, if none of the previous operations are possible, each process with a lower-than-average rendering time is delegated a slice of blocks from a process with a higher-than-average rendering time, which then has to be put into a new working set. Processes with lower-than-average and higher-than-average rendering times are iterated starting with the lowest and highest rendering times, respectively. As such, the process with the lowest rendering time performs load balancing with the process that has the highest rendering 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 rendering time that currently has delegated blocks to fewer than two other processes. A slice of blocks is then load balanced from the

Algorithm 2 Intra-group load balancing algorithm executed after each rendered frame

Input:
$\mathcal{L}$: Processes that have a lower-than-average rendering time; ▷ Sorted based on lowest rendering time
$\mathcal{H}$: Processes that have a higher-than-average rendering time; ▷ Sorted based on highest rendering time
$\mathcal{S} \leftarrow \{s_1, \ldots, s_p\}$; ▷ Dictionary of type {process, set of processes from which the process has been delegated blocks}
$\mathcal{E} \leftarrow \{e_1, \ldots, e_p\}$; ▷ Dictionary of type {process, set of processes that the process has delegated blocks to}
$\mathcal{T} \leftarrow \{t_1, \ldots, t_p\}$; ▷ Set containing the total rendering time of each process
Output:
$\mathcal{S}$, $\mathcal{E}$;

$\mathcal{B} \leftarrow \{b_1, \ldots, b_p\}$; ▷ Set containing all processes that have not performed load balancing this frame
for each $t \in \mathcal{L} \cap \mathcal{B}$ **do**
  if $e_t \neq \emptyset$ **then**
    set q where $q \in e_t \cap \mathcal{B}$ such that $t_q \geq t_i$, $\forall i \in e_t \cap \mathcal{B}$; ▷ q has the highest rendering time in $e_t \cap \mathcal{B}$
    recall load-balanced slice from q to t;
    $\mathcal{B} \leftarrow \mathcal{B} \backslash \{t, q\}$; ▷ Exclude t and q from other load balancing events
    if |load-balanced blocks from t to q| = 0 **then**
      $e_t \leftarrow e_t \backslash \{q\}$;
      $s_q \leftarrow s_q \backslash \{t\}$;
for each $t \in \mathcal{H} \cap \mathcal{B}$: **do**
  if $s_t \neq \emptyset$ **then**
    set q where $q \in s_t \cap \mathcal{B}$ such that $t_q \leq t_i$, $\forall i \in s_t \cap \mathcal{B}$; ▷ q has the lowest rendering time in $s_t \cap \mathcal{B}$
    recall load-balanced slice from t to q;
    $\mathcal{B} \leftarrow \mathcal{B} \backslash \{t, q\}$; ▷ Exclude t and q from other load balancing events
    if |load-balanced blocks from q to t| = 0 **then**
      $s_t \leftarrow s_t \backslash \{q\}$;
      $e_q \leftarrow e_q \backslash \{t\}$;
for each $t \in \mathcal{L} \cap \mathcal{B}$ **do**
  if $\mathcal{H} \cap s_t \neq \emptyset$ **then**
    set q where $q \in \mathcal{H} \cap s_t \cap \mathcal{B}$ such that $t_q \geq t_i$, $\forall i \in \mathcal{H} \cap s_t \cap \mathcal{B}$; ▷ q has the highest rendering time in $\mathcal{H} \cap s_t \cap \mathcal{B}$
    load balance slice from q to t;
    $\mathcal{B} \leftarrow \mathcal{B} \backslash \{t, q\}$; ▷ Exclude t and q from other load balancing events
for each $t \in \mathcal{L} \cap \mathcal{B}$ **do**
  if $\mathcal{H} \neq \emptyset$ **then**
    set q where $q \in \mathcal{H} \cap \mathcal{B}$ such that $t_q \geq t_i$, $\forall i \in \mathcal{H} \cap \mathcal{B}$; ▷ q has the highest rendering time in $\mathcal{H} \cap \mathcal{B}$
    load balance slice from q to t;
    $\mathcal{B} \leftarrow \mathcal{B} \backslash \{t, q\}$; ▷ Exclude t and q from other load balancing events
    $s_t \leftarrow s_t \cup \{q\}$;
    $e_q \leftarrow e_q \cup \{t\}$;
return $\mathcal{S}$, $\mathcal{E}$;

positive direction of the x-axis, or the negative direction in case another process already has been delegated blocks from the positive direction.

3.3.4 Image Compositing Pipeline

As described in Section 3.3.1, 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. As each process can load balance blocks to eight other processes, a maximum of $O(p)$ partial images are transferred during this step, where p is the number of processes. This operation can be performed in parallel in each group; here, the maximum number of partial images transferred within

a group is $O(p/g)$, where g is the number of groups. The first 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 that represents the full volume. It is as such performed in an inter-group fashion, and is identical to a k-d tree's or static technique's compositing stage.

We maintain the same resolution for all images generated and used in the two compositing steps. However, we note that utilizing various compression strategies [60] 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.

3.3.5 Process Memory Usage

Using dynamic load balancing, the achieved load depends on the contents of the visualized data set; as such, without knowledge about the underlying data, it is not possible to predict the benefits of using the proposed group technique. However, it is possible to estimate the worst-case memory usage.

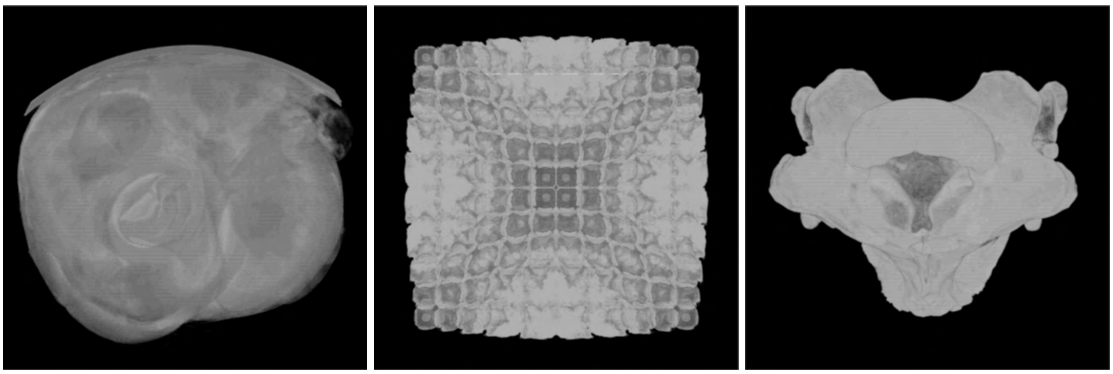
Given a volume of v voxels, if the group technique is used each process has to keep its full sets in memory, resulting in a memory usage of v/p. Furthermore, in the worst-case scenario, a specific process' full sets consist of blocks with near-zero rendering times. The process is then delegated blocks so that its rendering time matches that of the rest of the processes in the group. The absolute worst-case memory usage is as such v/g. However, as described in Section 3.3.3, the load balancing algorithm prioritizes load balancing blocks with high rendering times, ensuring that the process memory usage will remain lower than $2v/p$ other than in extreme scenarios.

3.4 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 rendering times, the process memory usage, the amount of transferred data, and 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.

3.4.1 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 E5-2643 v4 CPU (6 cores), 128 GB of memory, and two



(a) (b) (c)

Figure 3.8: Data sets used for evaluation: (a) a CT scan of a porcine heart, (b) a time step of an RMI simulation [1], and (c) a CT scan of a Spathorhynchus fossorium [2].

Nvidia GeForce GTX 1080 GPUs. Nodes were interconnected via EDR InfiniBand and used GCC version 7.3.0, CUDA version 9.2 [61], and Open MPI version 3.1.0 [62]. Up to two MPI processes were run on each node, each of which was allocated a dedicated GPU.

We chose to rotate the camera 360 degrees around the y-axis to measure the rendering 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 [32]. 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 has its own unique characteristics.

The first data set is a computed tomography (CT) scan of a porcine heart [2] (Fig. 3.8a). The second data set is of an RMI simulation [1] (Fig. 3.8b). The third data set is a CT scan of a Spathorhynchus fossorium [2] (Fig. 3.8c). The three data sets consist of $2048 \times 2048 \times 2612$ voxels (43.8 GB), $2048 \times 2048 \times 1920$ voxels (32.2 GB), and $1024 \times 1024 \times 750$ voxels (6.3 GB), 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 rendering 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 [22], 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 [4]. Based on this information we chose to partition the data sets into same-sized blocks consisting of around 64^3 voxels: $64 \times 64 \times 82$, $64 \times 64 \times 60$, and $64 \times 64 \times 47$ 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 [22]. 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. Rendering 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 [63] to compose images in the final compositing step of all evaluated techniques. Rudimentary empty space skipping [51] was used to avoid rendering empty blocks.

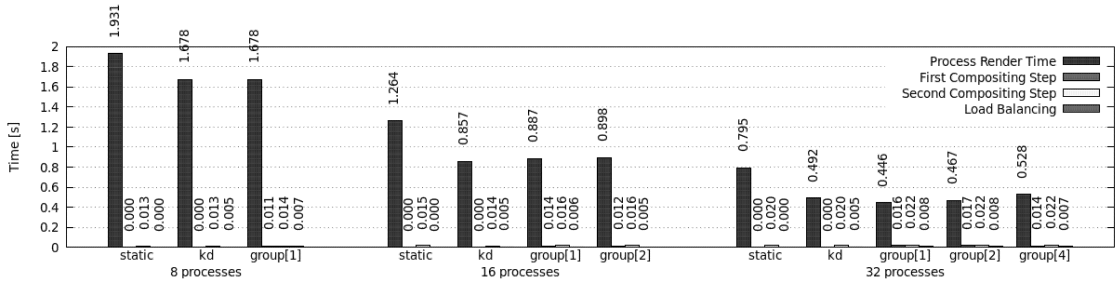
3.4.2 Performance Benefits of the Two-Layered Group Technique

We performed each test for multiple iterations on 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. 3.9. *Process Rendering Time* represents the rendering 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.

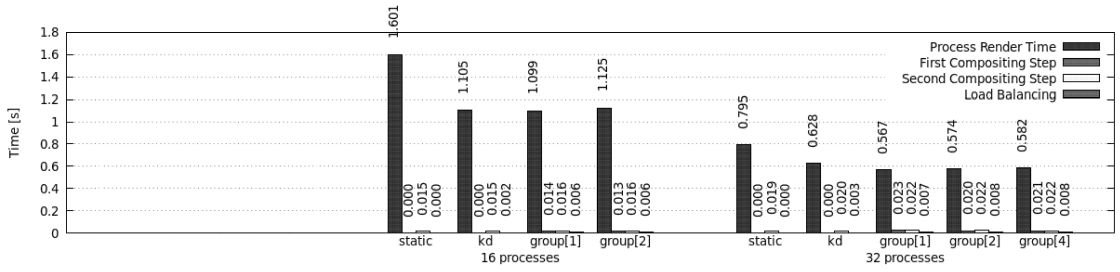
The memory usage was measured by tracking the highest number of blocks held in memory by a single process for each test, shown in Fig. 3.10. Using eight processes did not result in any substantial differences between the two dynamic techniques. For example, when using the porcine heart data set the highest recorded number 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 RMI 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 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.

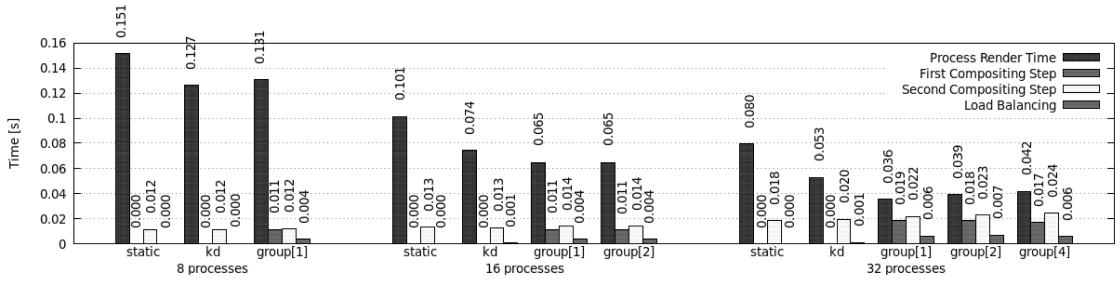
Figure 3.11 shows the average process rendering times for the three data sets. Both



(a)



(b)



(c)

Figure 3.9: Overview of the computation and communication times of the various stages in the rendering pipeline, including the process rendering time, the two compositing steps, and load balancing. The displayed values are the average of all frames using the (a) porcine heart, (b) RMI, and (c) Spathorhynchus fossorium data sets. group[1], group[2], and group[4] represent the group technique using 1, 2, and 4 groups, respectively.

evaluated dynamic techniques achieved lower process rendering times than the static distribution in all tests, clearly demonstrating the benefits of utilizing dynamic load balancing during large-scale visualization. However, as the number of processes increased, the group technique was able to achieve a lower rendering time than the k-d tree technique.

Using 8 or 16 processes resulted in similar process rendering times between the two

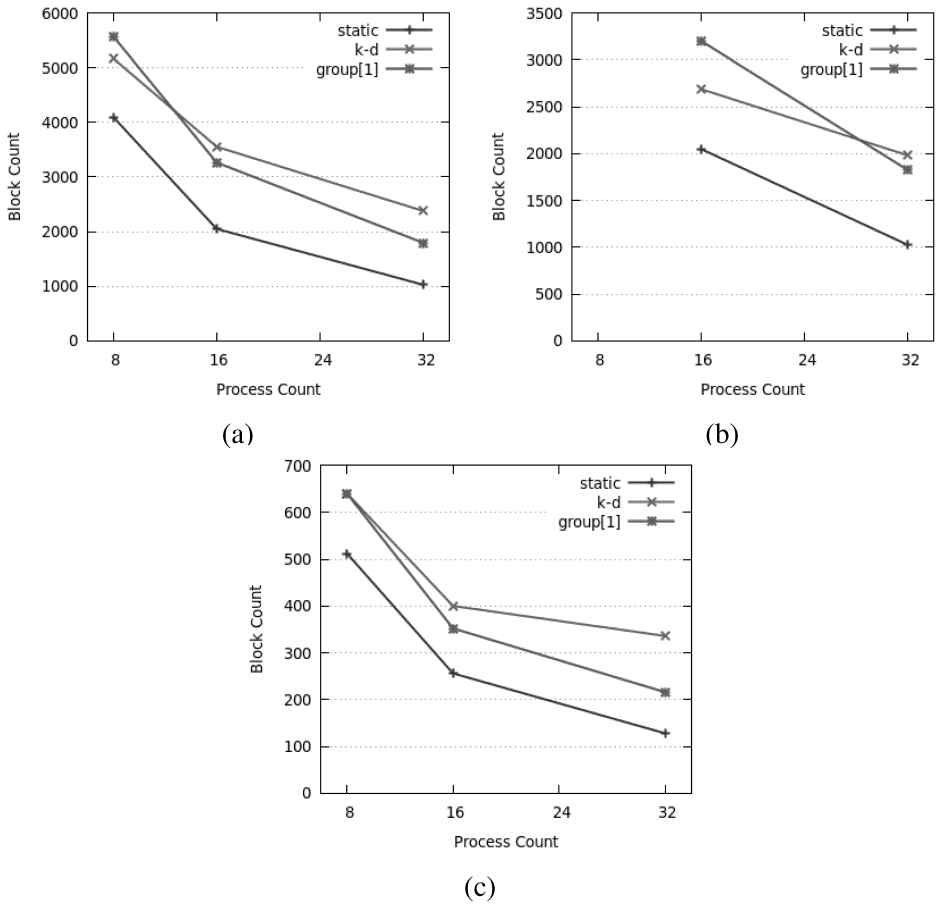


Figure 3.10: The highest number of blocks held in memory by a single process using the (a) porcine heart, (b) RMI, and (c) Spathorhynchus fossorium data sets on 8, 16, and 32 processes. Only one group is used in the case of the group technique.

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 rendering time; between 33.1% (Fig. 3.11c) and 9.5% (Fig. 3.11a) 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. 3.12. 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. 3.12a), 52.9% (Fig. 3.12b), and 260.0% (Fig. 3.12c) more data than the group technique for the three data sets. As an example, for the porcine heart data set 12,981

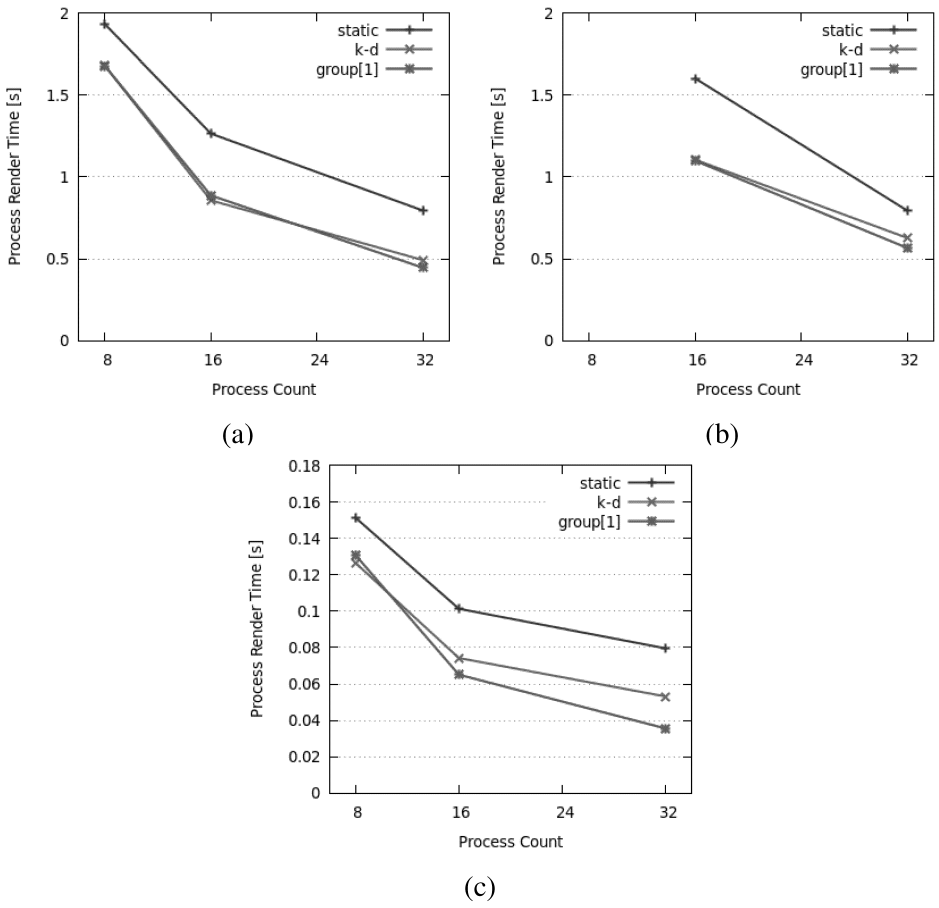


Figure 3.11: Average process rendering times using the (a) porcine heart, (b) RMI, and (c) Spathorhynchus fossorium data sets on 8, 16, and 32 processes. Only one group is used in the case of the group technique.

blocks 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 balancing occurred during the first few frames of the visualization to resolve 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.

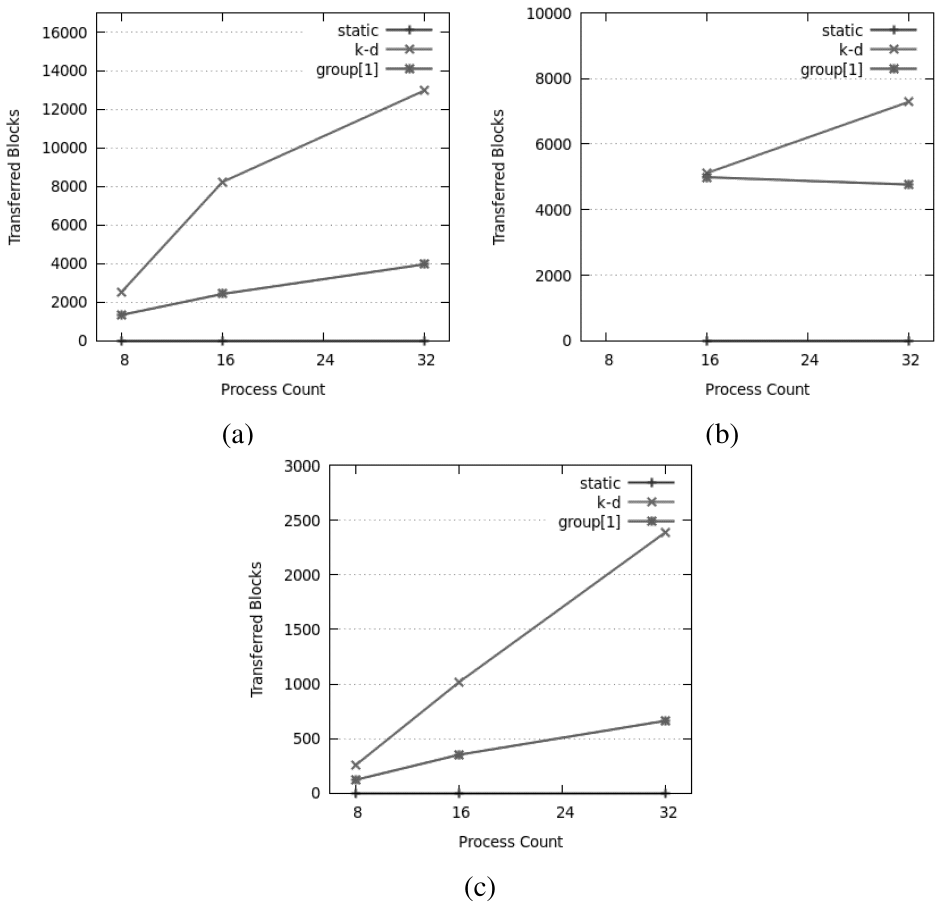


Figure 3.12: The total number of transferred blocks using the (a) porcine heart, (b) RMI, and (c) Spathorhynchus fossorium data sets on 8, 16, and 32 processes. Only one group is used in the case of the group technique.

3.4.3 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. 3.13. We observe that the first-step compositing times are similar for all three data sets and that they do not increase linearly with the number of processes. The highest increase was observed when going from 16 to 32 processes using the Spathorhynchus fossorium data set (Fig. 3.13c). 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 number of groups generally lead to a lower compositing time. Although seemingly

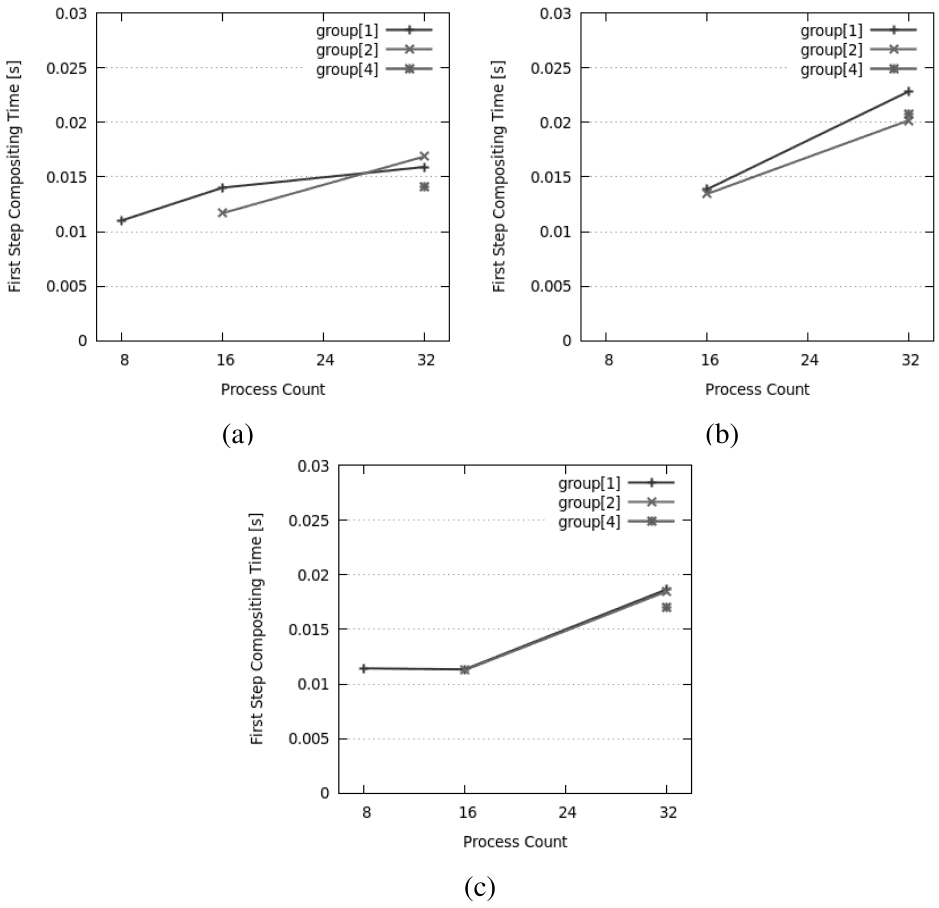


Figure 3.13: The first-step compositing times for the group technique using the (a) porcine heart, (b) RMI, and (c) Spathorhynchus fossorium data sets on 8, 16, and 32 processes. In each test, the group technique is evaluated using as few as eight processes per group.

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 rendering times. Figure 3.9 also includes the process rendering times for the group technique when using multiple groups; down to eight processes per group. Utilizing multiple groups sometimes results in a higher rendering time due to having too few processes in each group, which increases the chance of a high inter-group load imbalance.

3.5 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 number of redundant data transfers. These results were consistently obtained using three distinct data sets, indicating that similar results can be expected in other applications. 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 would like to evaluate the technique using more compute nodes to more accurately assess the benefits of utilizing multiple groups during large-scale visualization.

Chapter 4

Accelerating Multi-Image Compositing Using Dynamic Image Resolutions

4.1 Introduction

In sort-last parallel rendering [30], a data set is partitioned into multiple contiguous and convex blocks, and these blocks are distributed and rendered in parallel by multiple processes. The resulting images present on all processes then need to be composited, i.e., the images and pixels must be merged based on their distance to the camera, to generate a single image of the entire data set. In the context of scientific simulations, visualization is often performed in-situ [4, 6, 13, 14, 40] or in-transit [17, 20, 21, 34] as the simulation data is generated. Multiple images are often produced to represent different viewing angles, variables, and time steps. Certain tools [28, 29] can automate this process by generating thousands to millions of images at regular intervals around the studied phenomenon in 3D space. Such in-situ visualization tools can reduce the stored data size by orders of magnitude, while still allowing for post-hoc analysis of the data. In this chapter, we specifically consider the usage of a visualization tool like Cinema [28].

Here, consider several images, each representing a specific simulation variable or unique camera position. In such a case, each image is typically rendered and composited in sequence (Fig. 4.1); however, this type of image generation pipeline induces substantial overhead, primarily as a result of synchronization and communication that occur during rendering and compositing between different processes. Such overhead can be reduced by rendering and compositing images in batches [32]. In addition, combining batches of images into multi-images can further accelerate the compositing process.

An example of the multi-image pipeline is shown in Fig. 4.2. Here, multiple images are rendered in sequence on each process, and then a number of images (determined by the researcher) on each process are combined into a multi-image, which can be composited to generate images of the visualized data. In the example shown in Fig. 4.2, the first

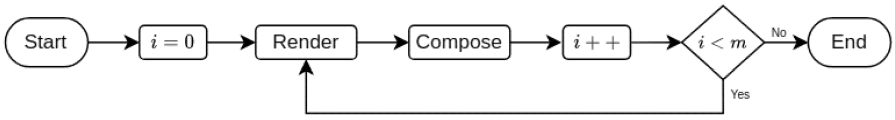


Figure 4.1: Traditional rendering and compositing are performed in sequence until all m images have been generated.

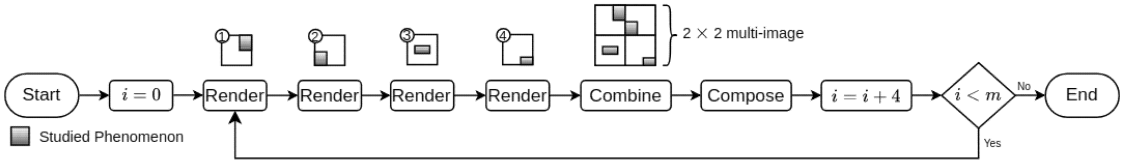


Figure 4.2: Example of batch compositing using a multi-image. In this example, four images are rendered consecutively on each process per batch (labeled 1–4). The images can, for example, represent different variables of a simulation or different camera positions. Images are then combined into a multi-image on each process (two images on each axis if each batch contains four images). Then, the multi-images are composited by the compute processes to produce a final image for each of the four partial images. Multiple batches are processed until all m images have been generated.

image is set to the top-left quadrant of the multi-image, the second image is set to the top-right quadrant, and so on. The output of the compositing stage is a multi-image that contains all of the final images, which otherwise would have been composited sequentially via multiple compositing stages. Thus, the synchronization overhead is reduced by compositing multiple images at once.

Using multi-images changes some typical characteristics of composited images. As shown in Fig. 4.3a, single images are prone to have large areas of blank pixels. Typically, the camera is focused on the studied phenomenon, making such empty space more common toward the outer regions of the image. This leads to redundant computations and load imbalances between processes, which has spawned many optimization techniques for single-image compositing, e.g., interlacing [64] and bounding box [3] techniques. By contrast, as shown in Fig. 4.3b, blank pixels in multi-images are spread fairly evenly throughout the multi-image, thereby making techniques like the bounding box technique significantly less effective.

The amount of data generated by scientific simulations is expected to increase by orders of magnitude in the future because of increased computing capabilities and more detailed simulations. Distributed rendering and compositing of large quantities of images is sure to become an essential method to analyze large-scale simulations. Therefore, it is important to develop new techniques to accelerate the compositing process and, more

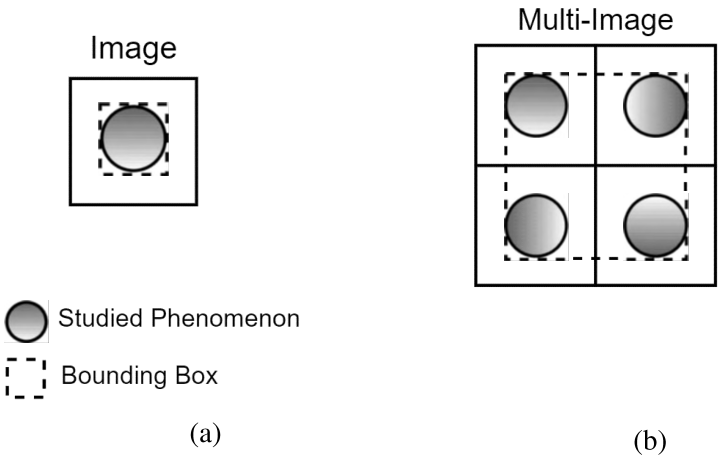


Figure 4.3: Example of (a) an image and (b) a multi-image (two images on each axis) using the bounding box technique. Pixels outside the region outlined by the bounding box can be ignored during the compositing stage.

specifically, utilize the unique characteristics of multi-images.

In this chapter, we present the dynamic image resolutions (DIR) technique to accelerate multi-image batch compositing. The DIR technique generates a *bounding grid* (BG) for each process; a low-resolution 2D grid that maps the regions of blank pixels in the process' multi-image. These grids can be used to avoid compositing blank pixels of a $k \times k$ resolution in the multi-image. Note that this functionality can be compared to empty space skipping [51], which is used in a similar manner to accelerate volume rendering of 3D data. The generated grid is then used to relocate parts of the multi-image and dynamically reduce the resolution of the multi-image through some of the image compositing steps. Here, the regions of pixels are relocated to a subset of the image based on the output of a recursive algorithm in a manner that does not compromise the fidelity or result of the compositing. Reducing the image size results in less computation and faster data transfers, which yields faster compositing times.

The remainder of this chapter is organized as follows. Related work is presented in Section 4.2, followed by a detailed description of the proposed technique in Section 4.3. The proposed technique is evaluated and compared to existing techniques in Section 4.4. Finally, the chapter is concluded in Section 4.5, including a summary of potential future work.

4.2 Related Work

Distributed sort-last image rendering and compositing is a widely researched field of study. As a result, many compositing methods exist [3, 31, 63–68]. Binary swap [3] is an efficient divide and conquer method that has been used in a wide variety of applications since its creation. The binary swap method has also served as a basis for many extensions and derivations [64, 65, 67]. We use the binary swap method with the proposed DIR technique for evaluation purposes. Other distributed compositing methods that use a tree order (e.g., the Radix-k [67] method), optimization techniques, and compression methods [60] can also be used in tandem with the proposed DIR technique.

Distributed image compositing involves compositing partial images, often generated from blocks of 3D data sets spread over multiple processes. Using the binary swap method, compositing consists of $\log\ p$ steps, where p is the number of processes. In each step, partial images are split in half, and then compositing is performed in pairs of processes that share the same image region. As a result, the number of processes that can perform compositing with any given process is reduced by 50% after each compositing step. An example of the distributed compositing process is shown in Fig. 4.4. We employ Z-buffering to manage the depth and how to composite pixels. Using Z-buffering, for each pair of pixels at index i, $0 \leq i < n$, where n is the number of pixels, the pixel with the shortest distance to the camera is composited on top of the other pixel. After all compositing steps are completed, the remaining partial images must be gathered and merged into a final image.

Multi-images were investigated by Larsen et al. [32], who explored two different multi-image strategies. However, these two strategies only differ in how the composited images are saved to permanent storage. The multi-image is set up by concatenating 10×10 images of a 1024×1024 resolution. Distributed rendering could be sped up because of two reasons. First, multi-image compositing is a type of batch-processing of images, meaning that the inter-process synchronization and communication times during rendering and compositing can be reduced by processing multiple images simultaneously. Second, using a multi-image can potentially improve the cache-hit rate and reduce overhead induced by data transfers, because all image data is allocated to a continuous memory region. Our work expands on this research by introducing a novel optimization technique that dynamically can reduce the resolution of the multi-image in a lossless manner.

4.3 Dynamic Image Resolutions

The DIR technique uses BGs and exploits the typical pattern of blank pixels in multi-images to reduce the image resolution dynamically during the compositing stage. Here, non-zero grid cells, and the pixels they represent, are reorganized to fit in the smaller image. This size reduction is achieved with no loss of detail.

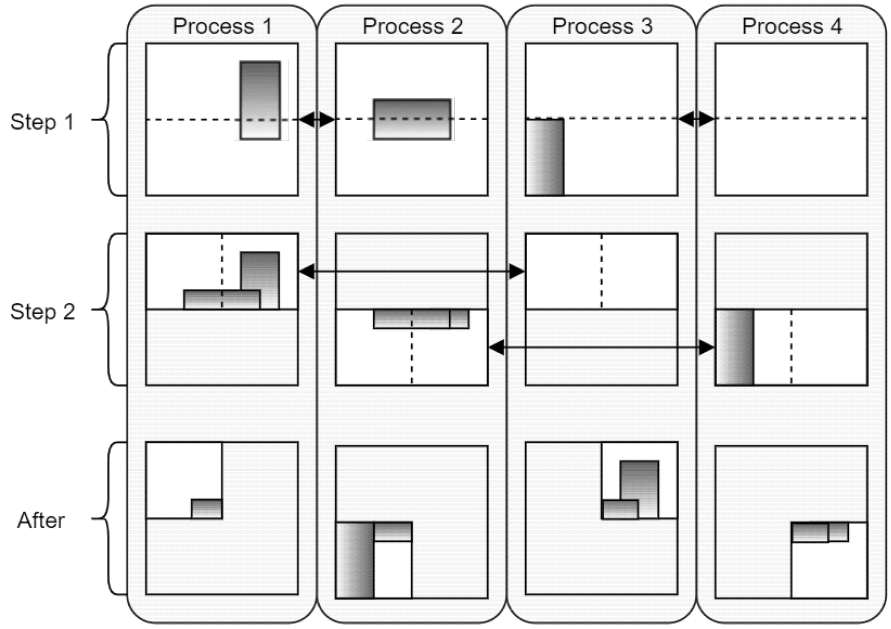


Figure 4.4: Example distributed image compositing between four processes using binary swap [3]. In each compositing step, images are split in half as indicated by the dotted lines. The end result is a fully composited partial image on each process that covers a part of the image domain. The compositing method works for both multi-images and single images.

4.3.1 Overview of the DIR technique

An overview of the DIR technique is shown in Fig. 4.5. The DIR technique consists of six stages. In stage 1, a BG is generated for the partial multi-image present on each process. Then, in stage 2, the grids are used to determine the order of operations required by the DIR technique. The order of operations is determined by an algorithm (described in Section 4.3.6). This algorithm uses the BGs to analyze the multi-images recursively to find the smallest image resolution that can be obtained with no loss of data (constraints explained in Section 4.3.4). The order of operations is then used in stage 3, where the non-blank regions of pixels are reorganized to fit in an image of smaller size (Section 4.3.7). In stage 4, compositing is performed using the reduced resolution as specified by the order of operations. Then, the multi-images are restored to their original resolution in stage 5, i.e., they stop using the smaller image size and relocate the non-blank pixels of each multi-image to their original positions.

Note that the reduced resolution cannot be used in all compositing steps in some cases (Section 4.3.5). In such cases, additional compositing steps are performed using the original resolution in stage 6.

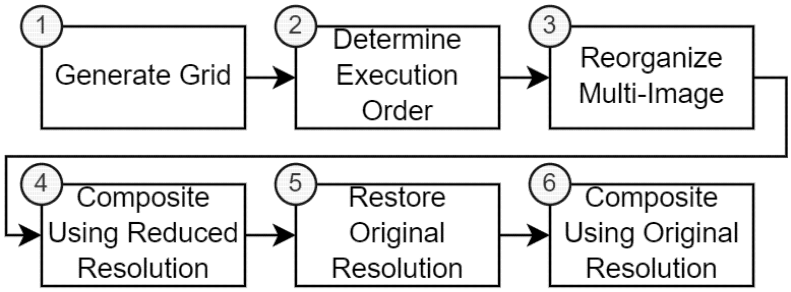


Figure 4.5: Overview of the DIR technique.

4.3.2 Bounding Grids

A BG is a low-resolution grid (i.e., a 2D array), where each cell in the grid represents k pixels on each axis in the original image, for a total of $k \times k$ pixels per grid cell. Figure 4.6 shows an example grid. The BG can identify blank pixels in multi-images more efficiently than a bounding box technique (Fig. 4.3). Whereas a bounding box can identify blank pixels in the outer regions of an image, a BG can identify blank pixels throughout the whole image. Note that BGs also work with single images (not only multi-images).

Before compositing the image, the grid is initialized by iterating through each pixel of the image. Here, for each grid cell, the cell's value is set to 0 if all $k \times k$ pixels it represents are blank; otherwise, the cell's value is set to the index of the cell (starting from 1). After each compositing step, a new image is created on each process by compositing two images. The BG of the new image can be updated using the data from the grids of the two composited images. Note that the resulting time complexity to update the grid is $O(n/k^2)$.

At the start of each compositing step, the grid is used to construct a bounding box for the image to exclude blank pixels at the outer regions. Throughout the compositing process, the grid is continuously checked to verify if an area of the image contains any non-blank data. If not, all pixels in the area can be skipped, which reduces the computational costs.

Given an image, a BG comprises n/k^2 grid cells. Thus, smaller k values increase the memory usage and time required to iterate the entire grid, and larger k values result in reduced memory usage and accelerate the grid iteration speed. However, smaller k values increase the grid resolution, which enables more fine-grained identification of blank pixels.

Similar to the bounding box technique, the time complexity to calculate the initial state of a BG is $O(n)$; however, the time complexity of updating the BG after compositing two images is $O(n/k^2)$, which is greater than the $O(1)$ of the bounding box. By setting $k = \Omega(\sqrt{n})$, the time complexity can be regarded as $O(1)$ instead of $O(n/k^2)$.

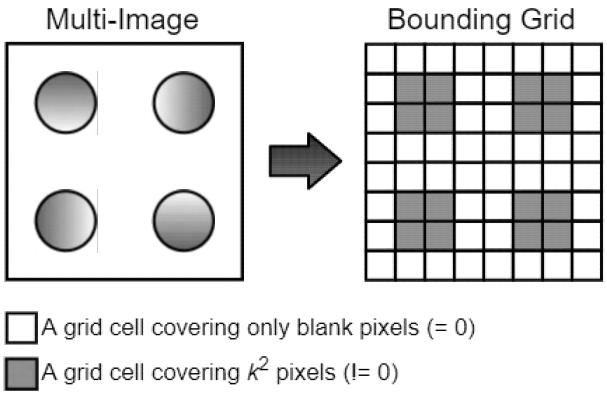


Figure 4.6: Example of the BG of a multi-image. The 8×8 grid keeps track of which image regions that contain non-blank pixels. Empty grid cells (represented by white cells) do not contribute to the final image; thus, pixels represented by empty grid cells can be skipped during the compositing stage.

4.3.3 Order of Operations

The DIR technique is executed through a sequence of operations. Here, three operations are used to realize the specified functionality of the DIR technique.

1. **Reorg**. This operation specifies the new image dimensions (determined by the recursive algorithm) and a grid that specifies the new index of each non-zero grid cell ($k \times k$ image region).

2. **Restore**. This operation restores the image to its original size and all pixels to their original positions.

3. **Composite**. This operation specifies a pair of processes that are to share images and perform compositing.

By executing the sequence of operations, stored as a list on each process, we can ensure correct functionality. Note that the Reorg and Restore operations only are executed once for each composited multi-image.

4.3.4 Constraints of the Lossless Image Reduction

To facilitate the explanation of the image reduction constraints, we introduce the term *end ID*, end ID $\in \{1, 2, \ldots, p\}$. For each cell, the end ID specifies which process the pixels represented by the cell are located on after the final compositing step that uses the reduced image resolution. To demonstrate an example using binary swap, assume that the DIR

technique is employed in Fig. 4.4. Here, in the final result, the top-left quadrant of the original image is present on process 1, i.e., grid cells representing pixels in the top left quadrant would have an end ID of 1. Note that the end ID calculation depends on the employed compositing method, n, k, p, and the resolution of the reduced image.

There are three constraints when reducing the image size and reorganizing non-blank pixels. These constraints ensure that the data relocation and the reduced image resolution do not affect the output of the compositing stage; thus, existing compositing methods can be used without modification. First, the new image (of reduced size) must be able to store all relevant data (non-blank pixels). Second, the DIR technique cannot change which pairs of pixels perform compositing. Pairs of pixels in the reorganized images can only be composited if they come from the same index i, $0 \leq i < n$, in the original image. Specifically, for p multi-images (one on each process), if on process 1 a pixel is relocated from index i to index j, the other $p - 1$ processes must perform the same relocation. Third, after the final compositing step using the reduced image resolution, a process' partial image cannot contain a grid cell (nor the pixels it represents) that, if not for the reorganization and resolution reduction, would be located on a different process.

The first constraint can be satisfied by ensuring that the number of non-zero grid cells with a specific end ID does not exceed the maximum (M) that can be stored.

The second constraint is satisfied by ensuring that, when any two pixels are composited, the values of the grid cells they belong to are identical, i.e., the grid cells they belong to had the same index in the original image.

The third constraint is satisfied if each grid cell, when relocated, is moved to an index of the grid with the same end ID the cell had before the relocation. As a result, the pixels represented by the cell are located on the same process after compositing is completed, regardless of whether the DIR technique is used. In contrast, if the end ID is different, the cell's pixels end up on a different process once the compositing stage has finished, thus not satisfying the third constraint.

The third constraint limits how pixels can be reorganized in the smaller image, and this constraint is the primary reason the DIR technique works well with multi-images. As described in Section 4.1 and illustrated in Fig. 4.3b, blank pixels are fairly evenly spread throughout a multi-image, i.e., the number of non-zero grid cells for each partial image should be similar at the end of the compositing stage. In most cases, this is not true when using single-image compositing; the non-blank pixels would instead be skewed toward a specific end ID, i.e., it would be harder to satisfy both the first and third constraints when reorganizing pixels to the smaller image.

4.3.5 Assigning Colors

Through testing, we discovered that increasing the value of p occasionally made it difficult to achieve any substantial reduction to the image resolution, as the non-blank pixels from the p multi-images could not fit in a smaller image. To make the DIR technique usable

Algorithm 3 CAN_ADD: determine if process h can be assigned color w. The algorithm checks whether the current level allows all non-zero grid cells to be stored.

Input:
p: number of processes;
r_h: BG for process h; ▷ h is the process to be added
b_w: BG for color w; ▷ w is the target color
l: level;
c: number of colors;
Output:
True or False; ▷ Whether h can be added to w

1: **for** $i \leftarrow 1$ **to** p **do**
2: $\mathcal{E}[i] \leftarrow 0$; ▷ Keeps track of the number of non-zero grid cells in every end ID
3: **for** $i \leftarrow 1$ **to** SIZEOF(r_h) **do** ▷ Iterate through all cells
4: **if** $r_h[i] \neq 0$ or $b_w[i] \neq 0$ **then**
5: $j \leftarrow$ GET_END_ID(i); ▷ End ID for i
6: $\mathcal{E}[j] \leftarrow \mathcal{E}[j] + 1$;
7: **if** $\mathcal{E}[j] > (\text{SIZEOF}(r_h) \cdot c) / (p \cdot 2^l)$ **then** ▷ M
8: **return** False; ▷ Cells exhausted
9: **return** True;

in large-scale environments, we assign a color to each process (c colors, where $c \geq 1$). Then, processes are partitioned into p/c independent sets based on the assigned colors. When compositing, communication is initially limited to processes within the same set. Thus, p/c processes communicate, reduce their image resolution, and composite images within each set. The number of multi-images (and processes) participating in these compositing steps is p/c, down from p, making it easier to reduce the image resolution with no loss of detail. Then, the original image resolution is restored, and the remaining $\log p - \log(p/c) = \log c$ compositing steps between processes assigned different colors can be performed normally (not using the reduced image size of the DIR technique). Here, the benefit is that the image resolution can be reduced also in large-scale environments.

To satisfy the first and third constraints, we must ensure that the number of non-zero grid cells belonging to the processes in a color's set do not exceed the maximum amount for any end ID. This verification process is described in Algorithm 3. The maximum number of non-zero cells for each end ID, M, is $\left(n/k^2 \cdot c\right) / \left(p \cdot 2^l\right)$ for each process, where l indicates a *resolution level*; how many times the resolution of the multi-image can be successfully reduced by 50%. For example, in Fig. 4.7, if $n/k^2 = 64$ (number of grid cells), $c = 1$, and $p = 2$, the maximum number for each end ID is 16 and eight for $l = 1$ and $l = 2$, respectively.

Increasing the number of colors also increases the computation time required to determine the order of operations and reduces the number of compositing steps that use the

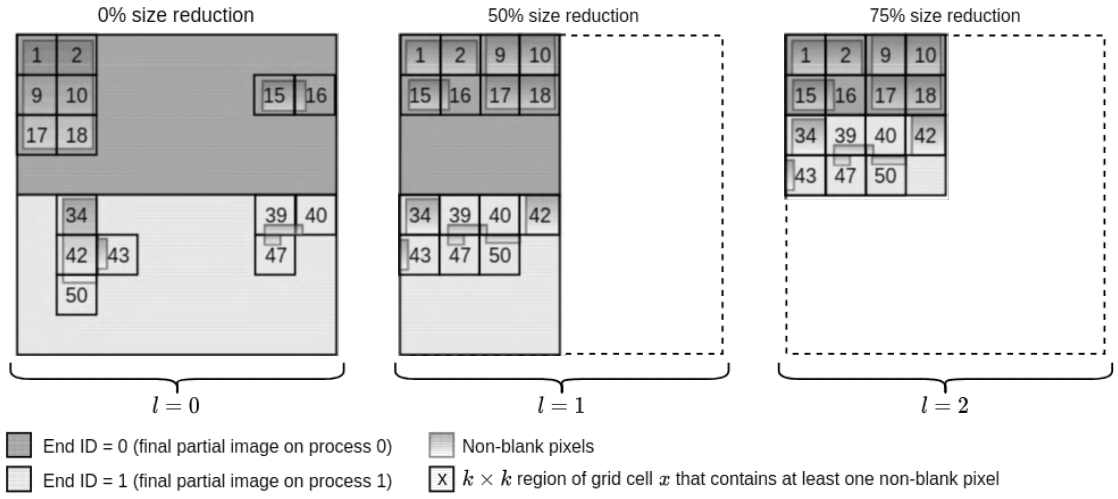


Figure 4.7: Example of data reorganization and size reduction for $l = 0$, $l = 1$ and $l = 2$, with $p = 2$. For any level, the number of pixels in the reduced image, n', is equal to $n/2^l$.

smaller image size. Thus, the maximum number of colors, `MAX_COLOR`, is limited to at most log p.

4.3.6 Image Size Reduction Method

The recursive Algorithm 4 shows how the order of operations is calculated (initially called from Algorithm 5). Here, given a number of processes p, p multi-images of n pixels (one on each process) and their BGs, $\mathcal{R} \leftarrow \{r_1, \ldots, r_p\}$, Algorithm 4 attempts to reduce the multi-image resolution as much as possible with no loss of detail. The output is the order of operations of each process.

To determine the reduced image resolution, the number of colors, and the pixel reorganization, we recursively try multiple l and c values in a bottom-up fashion. A maximum resolution level was determined through testing, and `MAX_LEVEL` = 4 (16× size reduction) was never exceeded in any tested configuration for non-empty images.

Given l and c, Algorithm 4 attempts to assign colors to processes such that exactly p/c processes are assigned to each color. Each color iterates through the list of available processes and adds the first process that can be added while satisfying the constraints (determined using Algorithm 3). Then, if any process has not been assigned a color, either c or l is updated to make partitioning easier. Algorithm 4 first attempts to increase the number of colors, because decreasing the level doubles the number of pixels in the smaller image. If the level is decreased, the number of colors is reset to 1. The order of operations can be generated once a successful partition has been found.

The first generated operation is always the Reorg operation, and then composite oper-

Algorithm 4 GET_OP_ORDER: a recursive function that produces the order of operations for each process. Operations concerning other processes are ignored during program execution.

Input:
p: number of processes;
$\mathcal{H}$: List of all processes, sorted by the number of non-zero grid cells;
$\mathcal{R}$: $\{r_1, \ldots, r_p\}$; ▷ Contains a BG for each process
l: resolution level;
c: number of colors;
q: index of the compute process;
Output:
$\mathcal{D}$: order of operations;

1: $\mathcal{U} \leftarrow \mathcal{H}$; ▷ Save copy of original list
2: $\mathcal{G} \leftarrow \{g_1, \ldots, g_c\}$; ▷ Set of c colors, $\forall g \in \mathcal{G}, g \leftarrow \varnothing$
3: $\mathcal{B} \leftarrow \{b_1, \ldots, b_c\}$; ▷ BGs for all colors
4: **for** $i \leftarrow 1$ **to** p **do**
5: $w \leftarrow i \mod c$;
6: **for** $j \leftarrow 1$ **to** SIZEOF($\mathcal{H}$) **do**
7: $h \leftarrow \mathcal{H}[j]$;
8: **if** CAN_ADD(p, r_h, b_w, l, c) **then** ▷ Algorithm 3
9: Add h to g_w;
10: Remove h from $\mathcal{H}$;
11: Update b_w to include all non-zero cells in r_h;
12: **break**
13: **if** SIZEOF($\mathcal{H}$) > 0 **then** ▷ Color partition unsuccessful
14: $c \leftarrow c \cdot 2$;
15: **if** $c >$ `MAX_COLOR` **then**
16: $c \leftarrow 1$;
17: $l \leftarrow l - 1$;
18: **return** GET_OP_ORDER(p, $\mathcal{U}$, $\mathcal{R}$, l, c, q);
19: **else** ▷ Color partition successful
20: Init order of operations $\mathcal{D}$ with Reorg operation for process q;
21: Add intra-color Composite operations to $\mathcal{D}$;
22: Add Restore operation for process q to $\mathcal{D}$;
23: **if** $c > 1$ **then**
24: Add inter-color Composite operations to $\mathcal{D}$;
25: **return** $\mathcal{D}$; ▷ Finished

ations between pairs of processes of the same color are appended. Here, all compositing pairs belong to the same color; thus, the reduced image resolution can be used with no loss of data. Given $\log(p/c)$ intra-color compositing steps and $p/2c$ pairs of processes

Algorithm 5 Calculates the order of operations on all p processes (function is called in parallel by each process).

Input:
p: number of processes;
$\mathcal{M}$: p multi-images of n pixels; ▷ One on each process
l: `MAX_LEVEL`;
c: `MAX_COLOR`;
q: index of the compute process;
Output:
$\mathcal{D}$: order of operations;

1: $\mathcal{H} \leftarrow [1, \ldots, p]$;
2: $\mathcal{R} \leftarrow \{r_1, \ldots, r_p\}$; ▷ Set to store the p BGs
3: **for** $i \leftarrow 1$ **to** p **do**
4: $\quad r_i \leftarrow$ the BG for the i-th multi-image $\mathcal{M}[i]$;
5: Sort $\mathcal{H}$ by the number of non-zero cells in each BG in $\mathcal{R}$ (descending order);
6: **return** GET_OP_ORDER(p, $\mathcal{H}$, $\mathcal{R}$, l, c, q); ▷ Algorithm 4

for every step, a total of $\log(p/c) \cdot p/2c$ compositing operations are performed within each color's set of processes. Then, as no more compositing steps can be completed using the reduced image resolution, the Restore operation is appended, to restore the images to their original resolution and all pixels to their original indices. Finally, the remaining $\log(c) \cdot p/2$ composite operations between pairs of processes assigned different colors are appended. After the operations have been executed, each process is left with a part of the final image.

4.3.7 Pixel Reorganization

We use the grid of each multi-image to reorganize different regions of pixels; i.e., k^2 pixels are relocated to the smaller image at a time. Figure 4.7 shows an example of the pixel reorganization process. As shown, each grid cell whose value is not zero is relocated to the smallest unoccupied index for its end ID, which satisfies the third constraint. This approach can potentially improve the efficiency of skipping the compositing of empty grid cells because all non-zero cells are concentrated in the same area of the grid. When a grid cell is relocated, the cell's value is still set to the original grid index it had before the reorganization. The index can then be used at the end of the compositing stage to relocate the cells (and the pixels they represent) to their original position in the multi-image. The pixel reorganization is identical for all processes assigned the same color, satisfying the second constraint.

4.3.8 Theoretical Performance Analysis

Maintaining the BG increases the memory usage by bn/k^2 on each process, where b determines how many bytes can be used by each grid cell to record the indices used by the Reorg operation. The best b value depends on the resolution of the multi-image and the k value. The Reorg operation also stores a grid containing bn/k^2 elements. As a result, the memory usage is increased by at most $2bn/k^2$. For example, if $b = 2$, $n = 20480^2$, and $k = 64$, the memory usage would only increase by 0.1%.

To facilitate understanding of the theoretical analysis, we first show the total time for a basic method before that of the DIR technique. The basic method here simply uses the binary swap method, with no optimization techniques applied. We define the total time T as a function of the number of pixels (n). Because the basic method consists of a sequence of compositing steps followed by a gather step, the total time can be given by:

$$T(n) = \sum_{i=1}^{\log p} t_{\text{step}}\left(\frac{n}{2^{i-1}}\right) + t_{\text{gather}}\left(\frac{n}{2^{\log p}}\right), \tag{4.1}$$

where t_{step} is the time required to process a compositing step in the binary swap method and t_{gather} is the time required to merge the partial images after all compositing steps have completed. The number of compositing steps and the image size for each step i depend on the used compositing method; using binary swap results in $\log p$ steps. As the number of pixels in an image is halved after each step, the number of pixels at step i is equal to $n/2^{i-1}$.

The time t_{step} can be further detailed as follows:

$$t_{\text{step}}(n) = t_{\text{comm}}(n) + t_{\text{calc}}(n) + t_{\text{sync}}(n), \tag{4.2}$$

where t_{comm}, t_{calc}, and t_{sync} are the inter-process communication time, the compositing computation time, and the inter-process synchronization overhead for each compositing step, respectively.

We next analyze the total time for the DIR technique. There are three differences compared to the basic method. First, for each compositing step, additional computation and inter-process communication are required to update and transfer the grid between processes. Let these actions be done in t_{grid}. We then have

$$t'_{\text{step}}(n) = t_{\text{step}}(n) + t_{\text{grid}}(n/k^2), \tag{4.3}$$

where t'_{step} is the time required to process a compositing step in the DIR technique. Second, as a result of the dynamic image size reduction, the number of elements is reduced from n to $n' = n/2^l$ for the first $\log(p/c)$ steps. Third, using the DIR technique, the BGs must be generated, called t_{gen}. In addition, the DIR technique involves the computation times of

the Reorg and Restore operations, t_{reorg} and t_{rest}, respectively. As a result, we obtain the following function T' for the total time:

$$T'(n) = \sum_{i=1}^{\log(p/c)} t'_{\text{step}}\left(\frac{n'}{2^{i-1}}\right) + \sum_{i=\log(p/c)+1}^{\log p} t'_{\text{step}}\left(\frac{n}{2^{i-1}}\right) + t_{\text{gather}}\left(\frac{n}{2^{\log p}}\right) + t_{\text{gen}}\left(\frac{n}{k^2}\right) + t_{\text{reorg}}(n) + t_{\text{rest}}(n). \tag{4.4}$$

The phases specific to the DIR technique (t_{gen}, t_{reorg}, and t_{rest}) all depend on n rather than p. As a result, the proposed technique should be more effective in large-scale environments, as the number of pixels typically remains constant; in such scenarios, t'_{step} is bound to account for a greater proportion of the total execution time.

4.4 Evaluation

To evaluate the proposed technique, we ran tests on a 16-node cluster (Table 4.1). Rendering was performed using the OSPRay rendering engine [37], version 1.7.3, and compositing was implemented in C++ and accelerated using OpenMP [69]. The multi-image size was set to 100 (10 images per axis), same as in related work [32]. However, we used an image resolution of 2048×2048, resulting in a resolution of 20480×20480 (1.67 GB) per multi-image, for a total of 26.84 GB of data spread across 16 nodes.

The camera was moved along a spherical spiral around each data set to ensure all images were rendered from different viewing angles and positions. We evaluate the following techniques, all of which use the binary swap method:

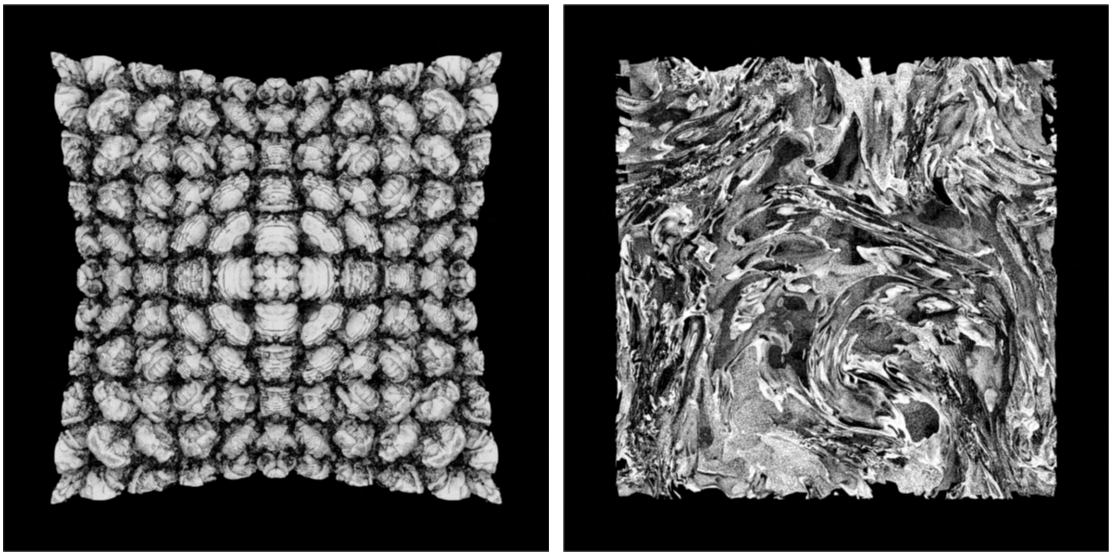
1. **Single**. Traditional single-image compositing.
2. **Basic**. Multi-image compositing with no optimization techniques.
3. **BB**. Multi-image compositing using a bounding box.
4. **DIR**. Multi-image compositing using the proposed DIR technique.

We used the traditional image compositing method shown in Fig. 4.1 in the single technique to demonstrate the differences in rendering times between the traditional approach and batch rendering.

Two data sets were used for testing purposes; referred to as data set 1 and data set 2. Data set 1 is a time step of the mass fraction of a Richtmyer–Meshkov Instability simulation [1], at a $2048 \times 2048 \times 1920$ resolution (64 GB). Data set 2 is a time step of the pressure field of a forced isotropic turbulence simulation [70], at a $4096 \times 4096 \times 4096$ resolution (550 GB). Figure 4.8 displays a visual representation of the two data sets.

Table 4.1: Cluster specifications. The cluster system consists of $p = 16$ nodes.

CPU	Xeon E5-1650 v4 6 cores
Memory	128 GB
Interconnect	InfiniBand EDR
Software	GCC 7.3.0, OpenMPI 3.1.0, and OSPRay 1.7.3



(a) (b)

Figure 4.8: Visualization of (a) data set 1 and (b) data set 2.

4.4.1 Selecting the Grid Size

To evaluate the effect of using different k values in the DIR technique, we ran tests for k values of 16, 32, 64, and 128. The results for data set 1 are shown in Fig. 4.9, which breaks down the compositing times according to the phases outlined in Section 4.3.8. Similar results were obtained for data set 2.

A k value of 128 resulted in a coarse-grained grid, that failed to reduce the resolution of the multi-image to the same extent as lower k values. As a result, the t_{step} phase was more time-consuming than the $k = 32$ or $k = 64$ cases. Note that decreasing the value of k further instead made the grid more fine-grained; however, the extra overhead from the t_{step}, t_{rest}, and t_{reorg} phases introduced by increasing the number of cells negated the benefits of the proposed DIR technique. Based on these results, we used $k = 64$ for all subsequent tests.

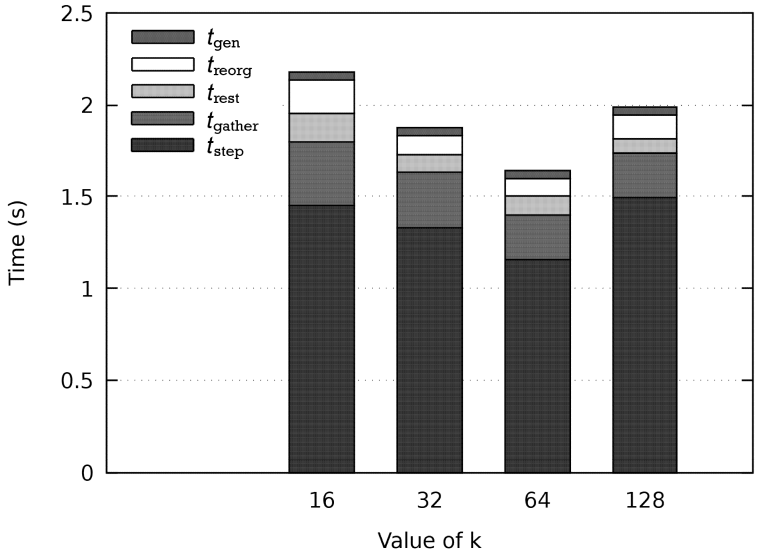


Figure 4.9: Compositing times using different k values on 16 processes ($k = 64$ achieved the fastest compositing time).

4.4.2 Overall Speedup

The total execution times (rendering and compositing) are shown in Fig. 4.10. Using batch processing proved to improve the execution time significantly. The biggest difference was observed between the single and DIR techniques, where speedups of 1.32× and 1.27× were observed on the two respective data sets. In addition, speedups comparing the DIR to the basic technique were 1.12× and 1.07×, respectively. Using batch processing sped up the rendering times by 1.2× on both data sets, which suggests that the performance benefits of rendering images in batches are independent of the size of the data set.

Rendering made up 77% of the total execution time on data set 1 and 85% on data set 2 for the basic technique. Using the DIR technique, the improvements to the overall execution times seem limited, as rendering is more time-consuming than compositing. However, increasing the p value increases the amount of compositing time compared to rendering; indeed, compositing is considered to be a significant bottleneck when performing large-scale visualization [63]. As a result, evaluating the compositing times independently of other computation tasks provides a better understanding of the speedup in different scenarios; such as when using smaller data sets or larger-scale systems.

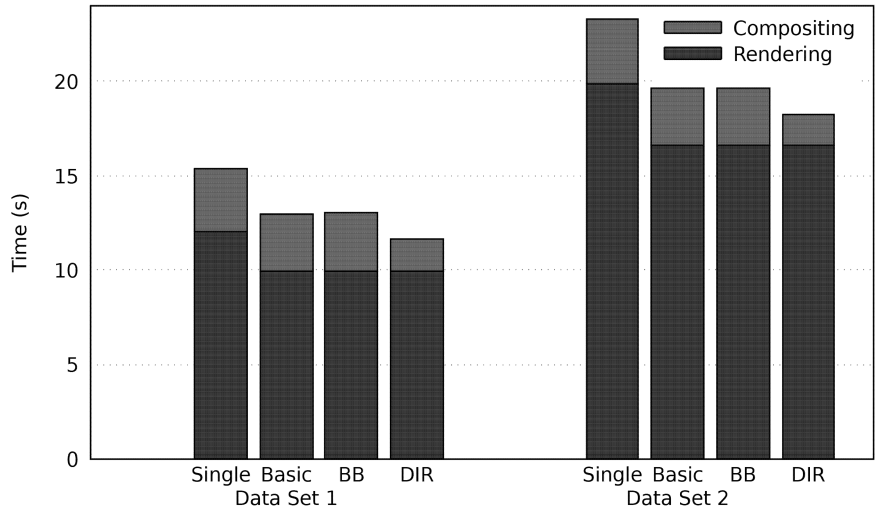


Figure 4.10: Total execution times (rendering and compositing times) of the evaluated techniques on 16 processes.

4.4.3 Compositing Performance

The compositing times of the evaluated techniques are presented in Fig. 4.11. In line with expectations, the bounding box technique did not achieve any significant speedup as compared to basic multi-image compositing (1% slower on both data sets). This result confirms that some common optimization techniques for image compositing are not as effective on multi-images. Using a multi-image was faster than single-image compositing. The basic technique achieved speedups of 1.11× compared to the single technique on both data sets. The proposed DIR technique achieved significant speedups of the compositing stage; 1.82× on both data sets compared to the basic multi-image technique. Similarly, the DIR technique achieved speedups of 2.02× compared to single-image compositing. On both data sets, we observed a 75% size reduction ($l = 2$) in our testing of the DIR technique (1.67 GB to 0.42 GB per multi-image).

Figure 4.12 shows a breakdown of the compositing times based on the phases outlined in Section 4.3.8. Using the DIR technique, t_{reorg} and t_{rest} together account for 12–13% of the total compositing time. However, these operations depend on the resolution of the multi-image (rather than p). As a result, the execution times of these operations should decrease in proportion to the total compositing time as p increases.

The t_{step} phase was significantly faster using the DIR technique than all other evaluated techniques, which is consistent with the theoretical performance improvements outlined in Section 4.3.8. Compared to the basic technique, the proposed DIR technique sped up the t_{step} phase by 2.38× and 2.43× on the two respective data sets. Even when accounting for the extra computation required by the DIR technique (t_{gen}, t_{reorg}, and t_{rest}), speedups of

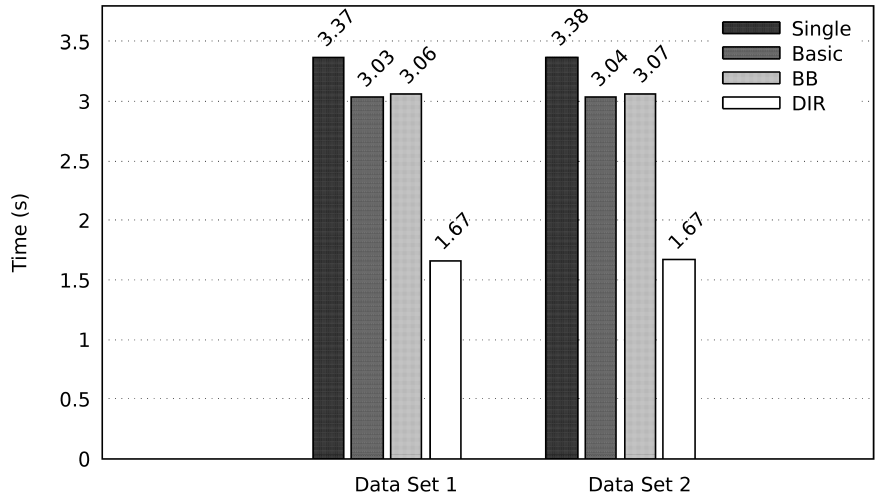


Figure 4.11: Compositing times using the evaluated techniques ($p = 16$).

1.97× and 2.00× were achieved on the two data sets. Based on these results, we conclude that using the DIR technique can accelerate multi-image compositing significantly.

To evaluate the scalability of the proposed DIR technique, we measured the compositing times using 2, 4, 8, and 16 processes (Fig. 4.13). We make two observations; first, the DIR technique achieves a greater speedup compared to the techniques used for comparison as we increase p. This relative improvement is a direct result of the reduced image size. For example, going from $p = 4$ to $p = 16$, the t_{step} execution time of the basic technique increased by 25% on data set 1; compared to an increase of only 4% when using the DIR technique. Second, t_{step} constitutes a greater proportion of the execution time of the DIR technique as the p value increases, consistent with our discussion in Section 4.3.8. Table 4.2 summarizes the execution time of each phase compared to the total compositing time on data set 1. Here, 45% of the compositing time initially consists of t_{step} when $p = 2$. Overhead induced by the t_{rest} and t_{reorg} phases makes the DIR technique less efficient; only a 1.16× speedup over the basic technique. As the value of p increases, the t_{step} phase accounts for a greater proportion of the execution time (over 70% when $p \geq 8$). The proposed DIR technique specifically accelerates the t_{step} phase, meaning that it should be able to further accelerate the compositing stage when the t_{step} phase constitutes a greater portion of the compositing time; in our case, a 1.82× speedup compared to the basic technique for $p = 16$ on both data sets. We conclude that the DIR technique scales better than the techniques used for comparison and that it is especially well-suited for large-scale visualization.

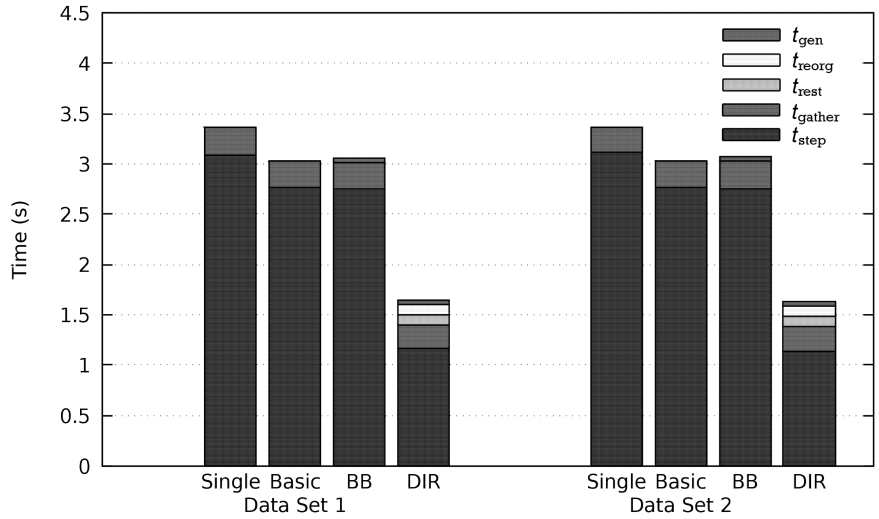


Figure 4.12: Breakdown of the compositing times of the evaluated techniques ($p = 16$). For the BB technique, t_{gen} represents the time to calculate the bounding box.

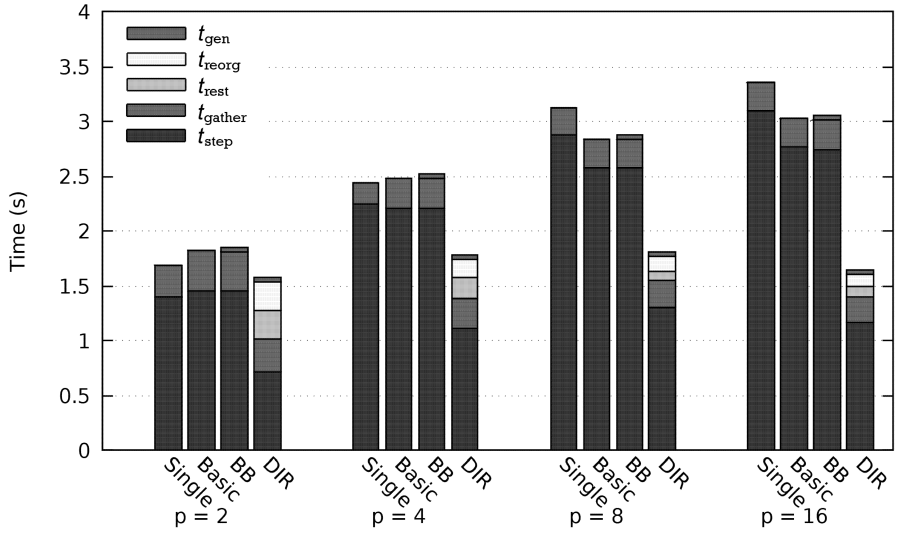
Table 4.2: Time spent by the DIR technique on each phase compared to the total compositing time on data set 1.

	t_{step}	t_{gather}	t_{rest}	t_{reorg}	t_{gen}
$p = 2$	45%	20%	16.3%	16.2%	2.5%
$p = 4$	62.4%	15.8%	10.3%	9.3%	2.3%
$p = 8$	72.2%	13.7%	3.9%	7.8%	2.3%
$p = 16$	70.5%	14.8%	6%	6.1%	2.6%

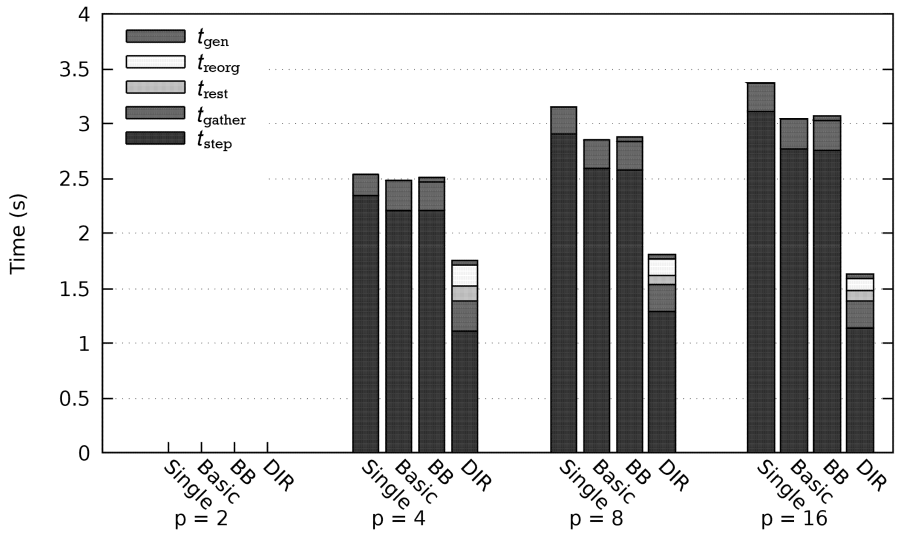
4.5 Conclusion

In this chapter, we proposed the DIR technique to accelerate multi-image batch compositing. Compared to existing multi-image compositing techniques, our contribution is twofold. First, the proposed technique maintains low-resolution grids that track empty regions in the composited multi-images, which can be ignored during the compositing process to reduce the computation time. Second, the grids are analyzed by a novel algorithm, whose output can be used to reduce the total image size in a lossless manner, which can lower the compositing time significantly.

The proposed DIR technique realized a 2.02× speedup of the image compositing stage as compared to traditional single-image compositing and 1.82× compared to existing multi-image techniques. Similar results were obtained using multiple distinct data sets,



(a)



(b)

Figure 4.13: Breakdown of the compositing times of the evaluated techniques on (a) data set 1 and (b) data set 2 using 2, 4, 8, and 16 processes. For the BB technique, t_{gen} represents the time to calculate the bounding box. Note that no results are presented for the case using two processes on data set 2; the processes ran out of memory and were not able to render the images.

meaning that the performance benefits of the proposed technique should not be limited to this application. We also showed that the DIR technique can accelerate the compositing for computing clusters of different sizes consistently. Moreover, we observed that the proposed technique achieved greater speedups as the number of processes increased. Thus, we conclude that the presented technique can accelerate the batch compositing of images and that the technique is especially suitable in large-scale cluster environments where extensive visualization is needed. The speedup compared to related techniques is excellent; using the DIR technique can potentially reduce the compositing time by up to 45% in large-scale applications.

In future work, we plan to continue improving the DIR technique to further accelerate the compositing process. In addition, we plan to investigate how multi-image compositing on GPUs can be accelerated by using different scheduling techniques and data structures.

Chapter 5

Conclusions

Here, we present a summary of the dissertation and discuss future work.

5.1 Summary of Work

In this dissertation, we identified and addressed three challenges facing large-scale co-processing of simulation data: (1) how to identify essential simulation data, (2) how to perform memory-efficient load balancing, and (3) how to accelerate multi-image batch compositing. These challenges were addressed in Chapter 2, Chapter 3, and Chapter 4, respectively.

In Chapter 2, we proposed a method that efficiently can identify important regions of simulation data, and then used the proposed method to combine the usage of multiple compression methods. Using a data-driven approach, we could successfully accelerate data transfers and data compression in an in-transit setting; achieving an overall speedup of 1.29× compared to using RLE. The proposed approach significantly accelerated the in-transit co-processing and should be usable in many in-transit applications.

In Chapter 3, we presented a dynamic load balancing technique by which processes can render data from non-contiguous regions of 3D data sets. We were able to reduce the amount of transferred data in large-scale environments by 72.2% and lower the highest observed memory usage compared to a typical k-d load balancing technique by 35.7%, while not negatively affecting the rendering performance. Similar performance benefits were shown for the three distinct data sets used for evaluation, meaning that similar results should be obtained in other applications. The technique has the potential to be used in large-scale or memory-limited scenarios where other dynamic load balancing techniques do not suffice.

In Chapter 4, we presented a technique that dynamically can reduce the size of multi-images with no loss of data, thus able to accelerate the image compositing process significantly. We were able to display excellent scalability compared to other techniques,

and achieved a speedup of 2.02× compared to traditional image compositing and 1.82× compared to existing multi-image techniques on two distinct data sets. Based on the evaluation results, the DIR technique can reduce the compositing time by up to 45% in other applications.

Summarizing the work of this dissertation, we successfully addressed three time-consuming challenges facing large-scale co-processing of scientific simulations, finding novel solutions that function at scale in distributed settings. We believe that the novel techniques and findings presented in this work can be applied to many applications that need to expeditiously process simulation data on the fly. We also believe that our techniques will perform even better in the future thanks to the excellent scalability shown in our testing.

5.2 Future Work

In future research, we would like to investigate how these applications can be used in combination to accelerate all stages of the visualization pipeline. The memory-efficient load balancing technique can be used to dynamically balance the computational load during the rendering stage, the DIR technique can accelerate the compositing stage, and by identifying essential data, we can prioritize different types of co-processing tasks at interesting time steps, compress data, and save important simulation data to permanent storage.

We also detail three topics of future research that build on the applications presented in this dissertation. First, we would like to further explore how identifying important simulation data can be used to accelerate and improve the co-processing process. In addition to many time-saving measures, we believe that many simulation-specific tasks can be automated; for example, the calculated importance can be used to determine which time steps to analyze. Second, continue work on developing new optimization techniques for batch image visualization using multi-images. Third, investigate how dynamic load balancing can be improved by knowing the importance of different data regions.

Bibliography

[1] R. H. Cohen, W. P. Dannevik, A. M. Dimits, D. E. Eliason, A. A. Mirin, Y. Zhou, D. H. Porter, and P. R. Woodward. Three-dimensional simulation of a Richtmyer–Meshkov instability with a two-scale initial perturbation. *Physics of Fluids*, 14(10): 3692–3709, 2002. doi: 10.1063/1.1504452.

[2] P. Klacansky. Open SciVis Datasets, 2018. URL `https://klacansky.com/open-scivis-datasets/`.

[3] K. Ma, J. Painter, C. D. Hansen, and M. F. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, 1994.

[4] 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. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 269–278, 2016.

[5] C. Wang, H. Yu, and K. Ma. Importance-driven time-varying data visualization. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1547–1554, 2008.

[6] B. Nouanesengsy, J. Woodring, J. Patchett, K. Myers, and J. Ahrens. ADR visualization: A generalized framework for ranking large-scale scientific data using analysis-driven refinement. In *IEEE 4th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 43–50, 2014.

[7] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53(3):484–512, 1984.

[8] H. Schive, J. A. ZuHone, N. J. Goldbaum, M. J. Turk, M. Gaspari, and C. Cheng. gamer-2: a GPU-accelerated adaptive mesh refinement code - accuracy, performance, and scalability. *Monthly Notices of the Royal Astronomical Society*, 481(4):4815–4840, 2018.

[9] T. Shimokawabe and N. Onodera. A high-productivity framework for adaptive mesh refinement on multiple GPUs. In *Computational Science – ICCS 2019*, pages 281–294, 2019.

[10] A. Biswas, S. Dutta, J. Pulido, and J. Ahrens. In situ data-driven adaptive sampling for large-scale simulation data summarization. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV '18)*, pages 13–18, 2018.

[11] S. Dutta, A. Biswas, and J. Ahrens. Multivariate pointwise information-driven data sampling and visualization. *Entropy*, 21(7):699, 2019. doi: 10.3390/e21070699.

[12] K. Moreland. The tensions of in situ visualization. *IEEE Computer Graphics and Applications*, 36(2):5–9, 2016.

[13] K. Ma. In situ visualization at extreme scale: Challenges and opportunities. *IEEE Computer Graphics and Applications*, 29(6):14–19, 2009.

[14] M. Rivi, L. Calori, G. Muscianisi, and V. Slavnić. In-situ visualization: State-of-the-art and some use cases. White paper, PRACE, 2012.

[15] H. Yu, C. Wang, R. W. Grout, J. H. Chen, and K. Ma. In Situ Visualization for Large-Scale Combustion Simulations. *IEEE Computer Graphics and Applications*, 30(3):45–57, 2010.

[16] J. C. Bennett, H. Abbasi, P. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*, pages 1–9, 2012.

[17] B. Friesen, A. Almgren, Z. Lukic, G. Weber, D. Morozov, V. Beckner, and M. Day. In situ and in-transit analysis of cosmological simulations. *Computational Astrophysics and Cosmology*, 3:4, 2016.

[18] M. Hahn, D. Drikakis, D. L. Youngs, and R. J. R. Williams. Richtmyer–Meshkov turbulent mixing arising from an inclined material interface with realistic surface perturbations and reshocked flow. *Physics of Fluids*, 23(4):046101, 2011. doi: 10.1063/1.3576187.

[19] I. W. Kokkinakis, D. Drikakis, and D. L. Youngs. Vortex morphology in Richtmyer–Meshkov-induced turbulent mixing. *Physica D: Nonlinear Phenomena*, 407:132459, 2020. ISSN 0167-2789. doi: https://doi.org/10.1016/j.physd.2020.132459.

[20] J. Kress. *In-Line vs. In-Transit In Situ: Which Technique to Use at Scale?* PhD thesis, University of Oregon, 2020.

[21] M. Walldén, M. Okita, F. Ino, D. Drikakis, and I. Kokkinakis. Accelerating in-transit co-processing for scientific simulations using region-based data-driven analysis. *Algorithms*, 14(5):154, 2021.

[22] S. Marchesin, C. Mongenet, and J. Dischler. Dynamic load balancing for parallel volume rendering. In *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization*, pages 43–50, 2006.

[23] C. Müller, M. Strengert, and T. Ertl. Adaptive load balancing for raycasting of non-uniformly bricked volumes. *Parallel Computing*, 33(6):406–419, 2007.

[24] W. Lee, V. P. Srini, W. Park, S. Muraki, and T. Han. An effective load balancing scheme for 3d texture-based sort-last parallel volume rendering on GPU clusters. *IEICE Transactions on Information and Systems*, E91-D(3):846–856, 2008.

[25] V. Bruder, S. Frey, and T. Ertl. Prediction-based load balancing and resolution tuning for interactive volume raycasting. *Visual Informatics*, 1(2):106–117, 2017.

[26] 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 Transactions on Visualization and Computer Graphics*, 24(1):954–963, 2018.

[27] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[28] J. Ahrens, S. Jourdain, P. OLeary, J. Patchett, D. H. Rogers, and M. Petersen. An image-based approach to extreme scale in situ visualization and analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*, pages 424–434, 2014.

[29] A. Kageyama and T. Yamada. An approach to exascale visualization: Interactive viewing of in-situ visualization. *Computer Physics Communications*, 185(1):79–85, 2014.

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

[31] M. Wallden, S. Markidis, M. Okita, and F. Ino. Memory efficient load balancing for distributed large-scale volume rendering using a two-layered group structure. *IEICE Transactions on Information and Systems*, E102-D(12):2306–2316, 2019.

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

[33] M. Flatken, C. Wagner, and A. Gerndt. Distributed post-processing and rendering for large-scale scientific simulations. In *Scientific Visualization: Uncertainty, Multifield, Biomedical, and Scalable Visualization*, pages 381–398. Springer London, London, 2014. ISBN 978-1-4471-6497-5.

[34] K. Moreland, R. Oldfield, P. Marion, S. Jourdain, N. Podhorszki, V. Vishwanath, N. Fabian, C. Docan, M. Parashar, M. Hereld, M. E. Papka, and S. Klasky. Examples of in transit visualization. In *Proceedings of the 2nd International Workshop on Petascal Data Analytics: Challenges and Opportunities (PDAC '11)*, pages 1–6, 2011.

[35] 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. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 89–96, 2011.

[36] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, K. Bonnell, M. Miller, G. H. Weber, C. Harrison, T. Fogal, C. Garth, A. Sanderson, E. Wes Bethel, M. Durant, D. Camp, J. M. Favre, O. Rübel, P. Navrátil, M. Wheeler, P. Selby, and F. Vivodtzev. VisIt: An end-user tool for visualizing and analyzing very large data. In *Proceedings of SciDAC*, 2011.

[37] I. Wald, G. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffersand J. Günther, and P. Navratil. OSPRay - a CPU ray tracing framework for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):931–940, 2017.

[38] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129–147, 1982.

[39] J. Ahrens, S. Jourdain, P. O'Leary, J. Patchett, D. H. Rogers, and M. Petersen. An image-based approach to extreme scale in situ visualization and analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*, pages 424–434, 2014.

[40] J. Woodring, J. Ahrens, J. Figg, J. Wendelberger, S. Habib, and K. Heitmann. In-situ sampling of a large-scale particle simulation for interactive visualization and analysis. In *Proceedings of the 13th Eurographics / IEEE - VGTC Conference on Visualization (EuroVis 2011)*, pages 1151–1160, 2011.

[41] A. G. Landge, V. Pascucci, A. Gyulassy, J. C. Bennett, H. Kolla, J. Chen, and P. Bremer. In-situ feature extraction of large scale combustion simulations using segmented merge trees. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*, pages 1020–1031, 2014.

[42] P. Bremer, G. Weber, J. Tierny, V. Pascucci, M. Day, and J. Bell. Interactive exploration and analysis of large-scale simulations using topology-based data segmentation. *IEEE Transactions on Visualization and Computer Graphics*, 17(9): 1307–1324, 2011.

[43] H. Zou, F. Zheng, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, Q. Liu, N. Podhorszki, and S. Klasky. Quality-aware data management for large scale scientific applications. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis (SCC '12)*, pages 816–820, 2012.

[44] T. Jin, F. Zhang, Q. Sun, H. Bui, M. Parashar, H. Yu, S. Klasky, N. Podhorszki, and H. Abbasi. Using cross-layer adaptations for dynamic data management in large scale coupled scientific workflows. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*, pages 74:1–74:12, 2013.

[45] J. Gu, B. Loring, K. Wu, and E. W. Bethel. HDF5 as a vehicle for in transit data movement. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV '19)*, pages 39–43, New York, NY, USA, 2019. Association for Computing Machinery.

[46] P. Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2674–2683, 2014. doi: 10.1109/TVCG.2014.2346458.

[47] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, and F. Cappello. Error-controlled lossy compression optimized for high compression ratios of scientific datasets. *2018 IEEE International Conference on Big Data (Big Data)*, pages 438–447, 2018.

[48] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. ISSN 1557-9654. doi: 10.1109/TIT.1977.1055714.

[49] S. Di and F. Cappello. Fast error-bounded lossy HPC data compression with SZ. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 730–739, 2016.

[50] G. Wang, J. Xu, and B. He. A novel method for tuning configuration parameters for spark based on machine learning. In *Proceedings of the 18th International Conference on High Performance Computing and Communications (HPCC '16)*, pages 586–593, 2016.

[51] W. Li, K. Mueller, and A. Kaufman. Empty space skipping and occlusion clipping for texture-based volume rendering. In *Proceedings of the 14th IEEE Visualization (VIS'03)*, pages 317–324, 2003.

[52] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9 (3):245–261, 1990.

[53] M. Matsui, F. Ino, and K. Hagihara. Parallel volume rendering with early ray termination for visualizing large-scale datasets. In *Parallel and Distributed Processing and Applications*, pages 245–256, 2005.

[54] Cybermedia Center. Cybermedia Center, Osaka University >> Blog Archive >> OCTOPUS, 2019. URL `http://www.hpc.cmc.osaka-u.ac.jp/en/octopus/`.

[55] F. F. Grinstein, A. A. Gowardhan, and A. J. Wachtor. Simulations of Richtmyer–Meshkov instabilities in planar shock-tube experiments. *Physics of Fluids*, 23(3): 034106, 2011. doi: 10.1063/1.3555635.

[56] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. *SIGGRAPH Computer Graphics*, 22(4):65–74, 1988.

[57] 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. In *Proceedings of the Eurographics / IEEE VGTC Conference on Visualization: State of the Art Reports*, pages 577–597, 2016.

[58] 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*, 29:340–351, 2012.

[59] T. Peterka, H. Shen, Y. Hong, K. Ma, H. Yu, and K. Moreland. Visualization and parallel I/O at extreme scale. *Journal of Physics: Conference Series*, 125(1):012099, 2008.

[60] J. Ahrens and J. Painter. Efficient sort-last rendering using compression-based image compositing. In *Proceedings of the 2nd Eurographics Workshop on Parallel Graphics and Visualization*, pages 145–151, 1998.

[61] NVIDIA, P. Vingelmann, and F. H.P. Fitzek. CUDA, release: 9.2, 2019. URL https://developer.nvidia.com/cuda-toolkit.

[62] The Open MPI Project. Open MPI: Open Source High Performance Computing, 2019. URL https://www.open-mpi.org/.

[63] K. Moreland, W. Kendall, T. Peterka, and J. Huang. An image compositing solution at scale. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*, pages 1–10, 2011.

[64] A. Takeuchi, F. Ino, and K. Hagihara. An improved binary-swap compositing for sort-last parallel rendering on distributed memory multiprocessors. *Parallel Computing*, 29(11):1745–1762, 2003.

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

[66] U. Neumann. Communication costs for parallel volume-rendering algorithms. *IEEE Computer Graphics and Applications*, 14(4):49–58, 1994.

[67] T. Peterka, D. Goodell, R. Ross, H. Shen, and R. Thakur. A configurable algorithm for parallel image-compositing applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, 2009.

[68] A. V. P. Grosset, A. Knoll, and C. Hansen. Dynamically scheduled region-based image compositing. In *Proceedings of the 16th Eurographics Symposium on Parallel Graphics and Visualization (EGPGV '16)*, pages 79–88, 2016.

[69] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[70] P. K. Yeung, D. A. Donzis, and K. R. Sreenivasan. Dissipation, enstrophy and pressure statistics in turbulence simulations at high Reynolds numbers. *Journal of Fluid Mechanics*, 700:5–15, 2012. doi: 10.1017/jfm.2012.5.