Scientific Visualization on Supercomputers: A Survey

Roba Binyahib

Department of Computer and Information Science, University of Oregon, Eugene, OR, USA

Abstract

Supercomputers increase both computing power and available memory. This allows scientists to generate high resolution physics-based simulations. Most of these simulations produce a massive amount of data, resulting in potentially trillions of cells. Scientific visualization is an essential method for understanding this simulation data. Visualization algorithms are usually run on supercomputers to leverage additional memory and computational power. Running visualization algorithms in distributed memory settings is challenging for several reasons such as I/O cost, communication cost, load balancing, and coordination between the nodes. In this paper, we survey the challenges and techniques for visualizing large data sets on supercomputers, discussing different visualizing algorithms and analyzing the factors impacting the performance.

Keywords: Supercomputing, distributed-memory, scientific visualization, parallel computing.

1. Introduction

Simulations enable scientists to study complex phenomena which may have been too difficult or expensive to study experimentally. That said, simulations can only replace experiments if they have sufficient accuracy, and achieving this accuracy often requires fine mesh resolution. Supercomputers allow scientists to achieve finer mesh resolutions by performing calculations at a massive scale. Examples of fields that regularly use large scale simulations are high energy physics, biology, and cosmology. For these fields and others, the simulations produce data sets potentially containing trillions of cells. These massive amounts of data are the key to future discoveries and scientific breakthroughs. Further, visualization is a powerful tool to achieve that goal, enabling scientists with ways to explore, extract, understand, and analyze important information.

The size of the data sets produced by today's large scale simulations make visualization difficult for several reasons. One complication is that data transfer is expensive, which often prevents transfers to local desktops or visualization clusters. Another complication is in the processing of large data. Some techniques reduce the processing costs by focusing on coarser versions of the data or on subsets of the data. These techniques, including multiresolution techniques and streaming, are used regularly in non-HPC environments. However, in the context of supercomputing, the dominant processing technique is parallelism, i.e. using the same supercomputer for not only simulation but also visualization. This is done by distributing data or workloads across multiple nodes, where each node visualizes its assigned portion. In most cases, the processing is done at the native mesh resolution and storing the entire mesh in memory (although distributed), requiring significant computational power, large memory, and I/O bandwidth. These requirements are often acceptable, however, since performing visualization on a supercomputer allows visualization algorithms to take advantage of the supercomputer resources. That said, visualizing such large data on a supercomputer (i.e., a distributed memory setting) adds new challenges. Even though most visualization algorithms are embarrassingly parallel, others require heavy communications and coordination. In addition, load balance must be maintained to run these algorithms efficiently, even with embarrassingly parallel algorithms.

These challenges have been the subject of various research efforts to improve the scalability and efficiency of visualization algorithms. In this survey, we cover techniques for visualizing large data sets at scale with an exclusive focus on distributed memory parallelism algorithms and their challenges. Section 2 provides areas of background for scientific visualization in a distributed memory setting. The majority of the research on distributed memory visualization are focused on three algorithms: particle advection (Section 3), volume rendering (Section 4), and image compositing (Section 5). The remainder of visualization algorithms are also discussed in this survey (Section 6), as are supporting infrastructures used by visualization algorithms (Section 7).

Explicitly, this survey focuses on performing visualization algorithms on supercomputers, and in particular the methods and optimizations required to visualize large data in a distributed memory setting. Related topics to this survey include multiresolution processing, streaming, hybrid parallelism, and in situ processing; while these topics are discussed when relevant to visualization on

© 2019 The Author(s)

supercomputers, they are otherwise considered out of scope for the survey.

2. Background

In this section, we cover important areas of background for scientific visualization on distributed memory. We start by discussing the impact of I/O on the visualization pipeline (Section 2.1). Next, we discuss the processing techniques for visualization algorithms (Section 2.2). Then, we take a look at the framework design used in most of visualization tools (Section 2.3). Finally, we discuss the parallelization design of visualization algorithms (Section 2.4).

2.1. I/O in Scientific Visualization

Computational power is increasing tremendously, while I/O systems are not improving at nearly the same pace. The main limiting factor for large scale visualization performance is I/O [CPA*10a, PYR*09]. Several techniques have been proposed to reduce the cost of I/O operations for visualization algorithms such as multiresolution techniques [CMNR07, PF01], subsetting [CBB*05, RPW*08], or parallel processing. In multiresolution techniques, data sets are stored in a hierarchical structure, and visualization is performed starting from the coarser data up to the finer ones. Subsetting is used to read and process only the portion of the data that will contribute to the visualization result. In parallel processing, the visualization method use the computational power of multiple nodes to process the data faster. Despite the presence of the first three techniques, the supercomputing community use parallel processing.

As supercomputers are pushing toward exascale, the gap between computation power and I/O is expected to increase even more. Consequently, I/O constraints are an important factor to take into account when designing visualization systems. Each one of the above mentioned techniques addresses I/O constrains. Multiresolution and subsetting solutions reduce the required I/O. Parallel processing increases the available I/O bandwidth.

2.2. Processing Technique

There are two processing techniques for visualization algorithms: post-hoc and in situ. The traditional paradigm is post-hoc processing, where scientists visualize their data as a post-processing step. In this model, the simulation code saves data to disk and is either read back later on the same computational resources or transferred to another machine for visualization. An alternative solution to reduce the cost of I/O is to use in situ processing [BAA*16], where the visualization is performed while the simulation is running. The data is streamed from the simulation code to the visualization. In situ visualization adds new challenges to both simulation codes and visualization systems which must be addressed. These challenges include for instance code modification, data flow management, synchronization between tasks, and difference of data models between the simulation and the visualization tool. Successful examples of in situ systems include Catalyst [FMT*11], and Libsim [WFM11a], which work along with Paraview and VisIt respectively.

The remainder of this survey will focus on efficient parallelization techniques regardless of their processing model.

2.3. Data Flow Framework

Parallel visualization frameworks have been developed to help users visualize their data. These frameworks include VTK [SML96], AVS [UJK*89], MegaMol [GKM*15], VisIt [CBW*12], and Paraview [AGL*05]. Most of these systems implement a data flow framework. A data flow framework executes a pipeline of modules where a module is an operation on its input data, and a link between two modules is a data stream. A module in the pipeline can be: 1) a source, 2) a filter, or 3) a sink. A source is a module that generates data, usually by reading data from a file. A filter is a module that takes data as an input, applies an operation, and produces an output. A sink is a module that receives data and produces a final result which can be written to file or displayed on a screen. These frameworks implement each visualization algorithm as an independent module. Figure 1 shows an example of a visualization pipeline: the source (read operation) reads data from a file, the filters (compute density, clip data, and compute isosurface) apply operations on the data and generate new data, and the sink (write operation) receives the data to produce an output.

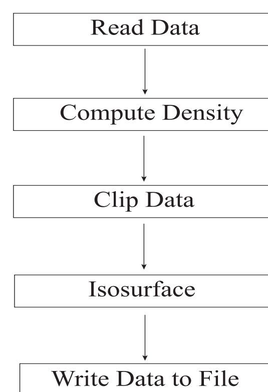

Figure 1: *A data flow example.*

Using a data flow framework has several advantages:

- The framework is abstract and hides the complexity from the users
- The framework is flexible and allows users to add new modules without requiring to modify old modules
- Modules of the framework can be combined to create advanced analysis.

2.4. Parallelization Design

The main challenge for parallel visualization algorithms is to decompose the work into independent segments, where processors can process their segments in parallel. These segments are usually data blocks. Most of visualization systems use a scatter-gather design. In this design, segments are scattered across different processors. Each processor reads its segment and applies the visualization pipeline using data flow network. Each processor has an identical

© 2019 The Author(s)

data flow network and processors differ in the segments they operate on. Then the results of different processors are gathered in the rendering phase.

Visualization algorithms can be classified int two categories: 1) embarrassingly parallel and 2) non-embarrassingly parallel. In an embarrassingly parallel algorithm, each processor can apply visualization on its segment independently. On the other hand, a non-embarrassingly parallel algorithm depends on other processor's computations. The majority of visualization algorithms are embarrassingly parallel.

2.5. Load Balance

A major challenge when running algorithms in parallel is maintaining load balance. Load balance is defined as the allocation of the work of single application to processors so that the execution time of the application is minimized [SS94]. Maintaining load balance is essential to achieve good performance since the execution time is determined by the time of the slowest processor. There are two categories of load balancing: 1) static, and 2) dynamic. In static load balancing, the workload is distributed among processors during the initialization phase. The workloads then remain on their computational resources during the entire execution of the visualization. The challenge in static load balancing is to guarantee equal workload, which can be difficult for some visualization algorithms. In dynamic load balancing, the workload is distributed during run time by a processor acting as the master. Dynamic load balancing can be used when the workload is unknown before run time.

Most of the solutions in this survey focus on solving load imbalance for different visualization algorithms. Load imbalance can be defined with the following equation:

$$\text{Load imbalance} = \frac{\mathrm{T_s}}{\sum_{0<\mathrm{p}<\mathrm{N}} \mathrm{T_p}/\mathrm{N}}$$

Where T_p is the total non-idle execution time for processor P, and T_s is the total non-idle execution time of the slowest processor.

Load balance is a major focus of this survey as many of the solutions and optimizations were suggested to maintain load balance.

3. Particle Advection

This section provides an overview of the particle advection technique. Advection is the process of moving a massless particle depending on a vector field. This results in an integral curve (IC), which represents the trajectory the particle travels in a sequence of advection steps from the seed location to the final particle location. Particle advection is a fundamental building block for many flow visualization algorithms [GGTH07, Hul92, GTS*04, MLZ09, KGJ09, MLP*10].

3.1. Integration Methods

An approximation of the integral curves (ICs) is calculated using numerical integration methods [HNW93]. The complete IC is calculated on several steps until reaching the maximum number of steps or exiting the data. At each step, a part of the curve is computed between the previous particle location and the current. The vector field around the current location is used to determine the direction of the next location.

There are different methods to calculate the next location. The Euler method [HNW93] is the simplest and least expensive method. It uses only the vector field of the current location to calculate the next location. Equation 1 shows the Euler method, where p_{i+1} is the next location of the particle, p_i is the current location of the particle, h is the length of the advection step, and $v(t_i, p_i)$ is the vector field value at the current location at the current time step. Runge Kutta (RK) [PD81] is a higher order method that uses Euler in its steps. There are different orders of the method; the most commonly used is the 4th order method referred to as RK4. Using RK4 produces more accurate results than Euler, but it is more expensive since it uses more points. Equation 2 shows the RK4 method, where p_{i+1} is the next location of the particle, p_i is the current location of the particle, h is the advection step, and $v(t_i, p_i)$ is the vector field value at the current location at the current time step. In both methods, as the advection step size decreases the accuracy of the trajectory increases, as well as the complexity. And as the total number of advection steps increases, the accuracy of the trajectory increases as well as the complexity.

$$p_{i+1} = p_i + h \times v(t_i\, p_i) \tag{1}$$

$$\begin{aligned} p_{i+1} = p_i + \frac{1}{6} \times h \times (k_1 + 2k_2 + 2k_3 + k4) \\ k1 = v(t_i\, p_i) \\ k2 = v(t_i + \frac{h}{2} + p_i + \frac{h}{2} \times k1) \\ k3 = v(t_i + \frac{h}{2} + p_i + \frac{h}{2} \times k2) \\ k4 = v(t_i + h + p_i + h \times k3) \end{aligned} \tag{2}$$

3.2. Parallelization Overview

Particle advection is computationally expensive. Several components that impact the cost of particle advection: 1) data size, 2) the number of steps, 3) size of advection step, 4) the number of particles, and 5) complexity of the vector field.

The computation of ICs is an expensive problem due to the nonlocal nature of particle advection and the data dependency. The trajectory of the particle determines which data blocks need to be loaded, which is unknown before run time. The nature of this problem makes parallelization of particle advection a challenging problem and prone to load imbalance.

There are two main parallelization techniques [PCG*09]: 1) parallelize over data, and 2) parallelize over particles. In parallelize over data, data blocks are distributed among processors, and each processor advects particles in its data block. In parallelize over particles, particles are distributed among processors, and each processor advects its particles. In this section, we start by discussing the

© 2019 The Author(s)

challenges of parallel particle advection. Next, we survey the different parallel solutions and categorize them under one of three categories: 1) parallelize over data, and 2) parallelize over particles, and 3) a hybrid between the first two.

Parallel particle advection requires efficient memory, computation, communication, and I/O. Thus choosing a scalable parallelization technique depends on the case. The choice of parallelization methods depends on four factors: 1) data set size, 2) number of particles, 3) particles distribution, and 4) vector field complexity.

- **Data set size:** A given data set can be small enough to fit the main memory of a node or not. If the data set is small enough, it allows data to be replicated among processors and favors the distribution of particles (parallelize over particles). As the size of the data increases, distributing data becomes necessary, thus parallelizing over data might lead to better performance.
- **Number of particles:** When the number of particles is large, the computation complexity increases and thus distributing this complexity is important. If the number of particles is small, then it is better to distribute the data (parallelize over data) to reduce the I/O cost.
- **Particles distribution:** Particles can be located in a region of the data (dense) or be more scattered (sparse). If the particle distribution is dense, only a subset of the data set will be required, reducing significantly the cost of I/O. This setup is more favorable to parallelize over particle because in the case of parallelizing over data only a small number of processors would work. On the over hand, if the particles are spread out (sparse) and cover the whole data set, the cost of I/O will become more significant. In this case, parallelizing over data would be more favorable to limit the cost of I/O.
- **Vector field complexity:** As mentioned before, the vector field determines the next position of the particle, which determines which part of the data is needed. For example, if the vector field is circular, the same data blocks will be needed more than once. Consequently, a method that has low I/O is an optimal choice (parallelize over data). Another example is, the case where the vector field has a critical point, and most particles are advecting toward a specific point (i.e., specific data block). Thus using parallelize over data, in this case, would lead to load imbalance. Figure 2 shows examples of complex vector fields of four blocks data. This dependency increases the complexity of parallelization since this information is not known before run time without conducting any prior studies as done by several solutions [CF08, YWM07, NLS11].

3.3. Parallelize over Data

Parallelize over data was introduced first by Sujudi and Haimes [SH96]. In this method, data is distributed between different processors. Each processor advects the particles located at its block until they exit the block or terminate. When a particle leaves the current data block, the particle is communicated to the processor that owns the needed data block.

This technique reduces the cost of I/O which is more expensive than the cost of computation. While this technique performs well for uniform vector fields and sparse particles distribution, it can lead to load imbalance in other situations. This technique is sensitive to particles distribution and vector field complexity. Particles distribution can impact this method negatively in cases where the particles are located in a certain region of the data. Thus load imbalance might occur due to the unequal work distribution. In cases where the vector field is circular, the communication cost can increase. Examples of both cases have been presented in Section 3.2. Different solutions have been presented to avoid load imbalance.

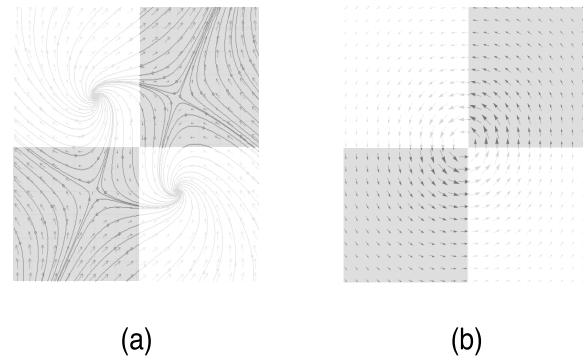

Figure 2: *Complex vector fields of four blocks data, (a) circular vector field, (b) a vector field with critical points, where particles advect toward the two data blocks colored white.*

Peterka et al. [PRN*11] presented a solution that used round-robin block assignment to guarantee that processors are assigned blocks in different locations. Their solution eliminates the load imbalance that could occur in cases where particles are located in a certain region of the data. While their method can reduce load imbalance, it can also increase the communication cost.

Different solutions have used a pre-processing step to maintain load balance. Chen et al. [CF08] presented an algorithm that reduced communication cost. Their solution considered the particles distribution and the vector field while partitioning the data into blocks. Their method partitioned the data depending on the vector field direction, thus reducing I/O cost.

Another solution that considered vector field was presented by Yu et al. [YWM07]. Their solution clustered data based on their vector field similarity. Next, the algorithm computed a workload estimation for each cluster. This estimation was used while distributing the data among processors.

Nouanesengsy et al. [NLS11] presented a method that also used a pre-processing step. Their algorithm used a pre-processing step to estimate the workload of each block using the advection of the initial particles. The results from the pre-processing step were used to distribute the work among processors. Each processor was assigned a percentage of the work of each block. Blocks are loaded to all processors that share the workload of the block. Their solution maintained load balance and improved performance. While these solutions resulted in better load balance, they introduced a new cost which is the pre-processing step. This cost can become expensive when the data size is large.

To avoid the cost of the pre-processing step, Peterka et al.

[PRN*11] proposed a solution that used dynamic load balancing. Their method repartitioned data blocks depending on their workload using recursive coordinate bisection (RCB) algorithm [BB87]. The workload was computed as the total advection steps so far. The algorithm checked the workloads at regular intervals during run time and redistributed the work.

While the round robin solution presented in [PRN*11] was simple, it did show good results. This solution is sensitive to particles distribution and vector field complexity. The algorithm did reduce the potential of load imbalance when the particle distribution is dense, but it might still occur if the data size is large and a small region of the data has all the particles. This will lead to a small group of processors doing all the work. If the vector field is complex, such as circular, the communication cost can become expensive, and if the vector field has a critical point, this will lead to load imbalance.

Using a pre-processing step to distribute the workload among processors can improve performance. While these solutions [CF08, YWM07, NLS11] can lead to better load balance, the cost of the pre-processing step might be expensive leading to reduced overall performance. This cost increases when the data size is large, and most likely the pre-processing step has to be performed by different processors, thus introducing additional communication cost. Paying the additional cost might not lead to improved overall performance especially when the number of particles is small.

Although dynamic load balancing [PRN*11] can maintain load balance while avoiding the pre-processing cost, their solution showed lower performance than the static round robin distribution. This is due to the additional cost of data redistribution and communication.

3.4. Parallelize over Particles

In this technique, particles are distributed across different processors. Particles are sorted spatially before distributing them to different processors to enhance spatial locality. Each processor advects its particles and loads data blocks as needed. To minimize the cost of I/O, a processor advects all particles that belong to the loaded block until the particles are on the boundaries of the block. This technique uses the least-recently used LRU mechanism to cache blocks. If there is not enough space when a new block is loaded, the least recently used block is discarded. Each processor terminates when all its active particles are terminated.

Since this technique loads data on demand, the cost of I/O dominates most of the run time. Data prefetching [RTBS05, AR13] has been used to reduce this cost. The idea of data prefetching is to load the next predicted needed data block while advecting the current particles to hide the I/O cost. Since the performance of this method depends on the accuracy of the prediction, the I/O access patterns are stored. Several solutions [CXLS11, CNLS12, CS13] have computed an access dependency graph to improve prediction accuracy. They performed a pre-processing step to compute the graph.

Camp et al., [CCC*11] used an extended memory hierarchy to reduce the cost of I/O. In their solution, data was stored in solid state drives (SSDs) and local hard drives instead of the file system. The algorithm treats SSDs as a cache where data blocks are loaded. Since the cache can hold a smaller amount of data than memory, blocks are removed in a least-recently used (LRU) mechanism when exceeding the maximum specified number of blocks. When a data block is not found in cache, the algorithm checks local hard drives before accessing the file system. This extended hierarchy increased the size of the cache which leads to less memory access and thus reducing the I/O cost.

Even though the number of particles distributed between different processors is the same, the workload might still be unbalanced. This is because particles might have different advection steps (some particles terminated early). To guarantee equal workload, different algorithms used dynamic load balancing methods. One of the load balancing methods is based on a work stealing approach [DLS*09]. In this approach, once a processor is done advecting its particles it steals particles from another busy processor. The processor stealing the particles is called a thief, and the other processor is called a victim. Each processor stores its particles in a queue, the thief processor transfer particles from the victim's queue. The most common approach to choose a victim is randomly [BL99]. Work stealing showed good results but it is difficult to implement. Thus another method was presented by Muller et al. [MCHG13] called work requesting. In work requesting, the victim sends the particles to the thief, while in work stealing the thief takes the particles without any action from the victim. While work requesting adds additional communication cost, it is easier to implement than work stealing especially on a distributed memory setting.

Other solutions used k-d tree decomposition to balance the workload during run time. Morozov et al. [MP16] presented a solution that used k-d tree decomposition to redistribute workload. Their algorithm checked for active particles at regular time intervals. Active particles are divided into groups, where all groups have the same workload. Next, each processor was assigned a group. While this method achieved load balance, it required access to the entire data set, which can increase the I/O and memory cost. Zhang et al. [ZGH*18] proposed a solution to avoid this cost. Their algorithm assigned a data block with ghost layers to each processor before run time. When the algorithm performed the particles decomposition, it considered the data blocks assignment. Thus each processor received particles which were in its data block. Their results showed improved load balance while maintaining the cost of I/O.

The most expensive step in parallelize over particles is I/O [CPA*10b]. While several solutions [CXLS11, CNLS12, CS13] reduced this cost by using prediction to apply prefetching, they introduced additional costs and these cost increases as the data size and/or the number of particles increase. Camp et al., [CCC*11] reduced the I/O cost, but the algorithm can perform poorly when dealing with complex vector fields and large data size. In addition, the algorithm can suffer from load imbalance if the advection steps vary between processors. The solution suggested by [MCHG13] avoided load imbalance but at the cost of additional communications. While dynamic load balancing [MP16, ZGH*18] avoided the cost of pre-processing and it considered the change in the vector field, it added an additional cost of redistributing particles. This cost could increase when the number of particles is large.

© 2019 The Author(s)

3.5. Hybrid Parallel Particle Advection Solutions

Both parallelize over data and parallelize over particles have limitations as presented in Sections 3.3, and 3.4. Hybrid solutions have been proposed to address these limitations and maintain load balance while reducing additional costs.

Pugmire et al. [PCG*09] proposed a hybrid solution known as the master/worker. In their algorithm, processors were divided into groups, where each group had a master. The algorithm partitioned the data statically and loaded data blocks on demand. The master distributed particles between the workers and monitored the workload to ensure load balance. When a processor needed a data block, the master followed a set of rules to decide whether the worker should load the block or send the particle to another worker. Their algorithm showed better performance than both traditional parallelization techniques and has been used in the VisIt framework [CBW*12].

DStep [KWA*11] is another similar hybrid solution. DStep is a framework designed for parallelization domain traversal techniques. They used a static round robin to assign the data blocks to processors. Processors were divided into groups, where each group had workers and had a communicator processor (master). The algorithm stored particles in a queue, and the communicator assigned particles to processors depending on their workloads. Processors of the same group can communicate and send particles, and communicators of each group exchange particles between different groups. Their algorithm showed scalability and has been used in several implementations [GYHZ13, GHS*14, LGZY16].

Lu et al. [LSP14] presented another hybrid solution to compute stream surfaces (Section 3.7). A stream surface is a union of streamlines (particles trajectories) connected to form a surface. Their solution distributed data blocks among processors. Next, particles were distributed among processors in segments, where a segment is a part of the surface that is computed between two particles. Each processor had a queue that stored the assigned segments. During run time, when a processor needed a data block, it requested it from the processor that owns the block. To make sure the load is balanced if particles of a segment diverge, the segment was divided into two segments and pushed to the processor queue. When a processor was idle, it acquired more work by stealing work from other processors. Their results showed load balance and good scalability.

Hybrid solutions reduce load imbalance, but they are more complicated to implement, and they introduce additional costs. The solution presented by Pugmire et al. [PCG*09] does not require a pre-processing step, and it avoids redistributing the data. The algorithm showed better results than traditional techniques but it could still suffer from high I/O cost since it loads data on demand. Algorithms based on a master/worker design [PCG*09, KWA*11] can perform poorly when the number of processors is small, and the number of particles is large. This is because not all processors are performing computation (advection). When the number of processors is large, the communication between workers and masters can become a bottleneck, thus finding the correct group size can impact the performance. DStep [KWA*11] lowered the potential of the communication congestion between the workers and master by allowing processors of one group to communicate directly. The solution proposed by Lu et al. [LSP14] avoided this communication congestion, but it had the additional cost of communicating data blocks between processors.

3.6. Summary

Table 1 shows a summary of the factors mentioned in Section 3.2 and the best configuration for each of these factors using the two traditional parallelization techniques. Each one of these factors impacts the choice of the parallelization technique, and they should all be considered when choosing a parallel solution.

Parallelize over data is generally favored when the data size is large since it has less I/O cost than parallelize over particles. However, it can suffer from load imbalance if the vector field has a critical point. The round robin distribution suggested by Peterka et al. [PRN*11] might still lead to load imbalance since the critical point can be located in one block of the data. And while the several solutions proposed for parallelize over data [CF08, YWM07, NLS11, PRN*11] can maintain load balance, these solutions can be expensive when the data size or the number of particles increases due to the pre-processing or redistribution costs (when using dynamic load balancing). On the other hand, parallelize over particles is favored when the number of particles is large or when the particles are densely distributed. Yet this technique can suffer from load imbalance if the number of advection steps per processor is very different. While the suggested solutions proposed for parallelize over particles [CXLS11, CNLS12, CS13, MCHG13, MP16, ZGH*18] can maintain load balance, they can still lead to bad overall performance due to the cost of pre-processing, communication, or I/O.

Hybrid solutions can be a viable alternative to traditional parallelization techniques. While these solutions maintain load balance and showed better performance, they can be are more complicated to implement, and typically have a high communication cost.

3.7. Flow Visualization Algorithms

As mentioned previously, particle advection is used in many flow visualization algorithms. In this section, we give a brief description of some of these flow visualization algorithms, such as streamlines, pathlines, streaklines, timelines, and stream surfaces. A streamline [CF08, CGC*11, CXLS11, PCG*09, NLS11, PRN*11] is the trajectory of the particle from the seed location to the final location. Streamlines are the basis of other flow visualization algorithms. A pathline [YWM07, CNLS12, CS13] is the trace of a particle through a period of time. Each pathline shows the moment of a certain particle through multiple time steps. A streakline is a line that connects the positions of different particles that passed a certain point. A timeline is a time that connects adjacent particles at a given time. A stream surface [CCG*12, LSP14] is a union of streamlines connected to form a surface. Figure 3 shows the different flow visualization algorithms. The most commonly used algorithms in scientific visualization are streamlines, pathlines, and stream surfaces.

As mentioned before, pathlines are traces of particles over time. This means that for each particle the algorithm is computing an additional value (three points for position and one for time), which increases the computational cost. In time varying data set, an additional challenge arises since particles might move from one block to

Table 1: *Comparing parallel particle advection methods. For each parameter, the table indicates which methods is best suited depending on the parameter.*

Problem Classification	Parallelize over data	Parallelize over particles
Data set size	Large	Small
Number of particles	Small	Large
Seed distribution	Sparse	Dense
Vector field	No critical point	No circular field

another over time. Thus the change over time has to be taken into consideration. Yu et al. [YWM07] presented a solution that used parallelize over data technique. Their algorithm considered time as a fourth dimension and performed a clustering based on the vector field similarity. Processors were assigned clusters depending on their workload, thus guaranteeing load balance over time.

The default setup for storing time varying data is to store each time step separately. Since a pathline algorithm computes the location of the next position in the next time step, the algorithm will need to access a different file with every integration step if parallelize over particles technique is used. This increases the I/O cost and might result in poor performance.

Chen et al. [CNLS12] presented an algorithm that reordered the storing of time varying flow data. Their algorithm used parallelize over particles technique and used data prefetching to load data blocks. The algorithm performed a pre-processing step to optimize the file layout and enhance the accuracy of prefetching. They divided the data into spatial blocks depending on their spatial locality. Next, particles that were in the same spatial block but in sequential time steps were grouped into a time block. In the pre-processing step, the algorithm computed an access dependency graph [CXLS11]. This graph was used to store time blocks and enhance data prefetching accuracy. Another solution that used access dependency graph to reduce I/O cost was presented by Chenet al. [CS13]. Their algorithm computed this graph in a pre-processing step, and grouped particles to the same block depending on their trajectories similarity. During run time, at each time step, processors advected particles in groups. They are thus reducing the number of I/O operations.

Stream surfaces are computed using a front-advancing approach that was introduced by Hultquist [Hul92] and used by other serial stream surfaces solutions [GTS*04, MLZ09]. In this approach, the algorithm started by placing the seeding curve, which are the initial particles. Next, these particles are advected forming streamlines. An arc is created between adjacent pairs of streamlines; these arcs result in a stream surface. The computation of the surface depends on the advection of the particles at the front of the surface. New particles are inserted or deleted depending on the divergence or convergence of the surface. There are additional challenges when parallelizing stream surfaces. For example, when the particles in the front of the surface diverge, new particles needs to be added. This adds to the workload of the processor owning that segment of the surface, which can lead to load imbalance. To reduce the potential of load imbalance,

Lu et al. [LSP14] presented an efficient solution that used work stealing technique to balance the work between processors. The algorithm is a hybrid between parallelize over data and parallelize over particles. Their solution divided the curve into segments that are distributed among processors. Each processor stored the segments in a queue and advected the particles in its segments. When the surface diverges and new particles are added, the algorithm formed new segments and inserted them to the processor queue. When a processor has no segments left, it requested segments from another processor. Camp et al. [CCG*12] presented another solution for stream surfaces. However, their solution did not apply the front-advancing approach. Instead, their algorithm computed streamlines independently (regardless of the parallelization technique) and created the surface between these lines (triangulation) after advection. After the advection step, the algorithm performed an adaptive refinement check. If the distance between adjacent streamlines was larger than a given threshold, a new particle was inserted. This new workload was distributed between processors (regardless of the parallelization technique) to perform the advection. Their algorithm reduced the potential load imbalance caused by the additional inserted particles.

4. Volume Rendering

There are two types of rendering: 1) surface rendering, and 2) direct rendering. Surface rendering is generating an image from a geometry that was produced by the visualization pipeline by converting the geometry into pixels through rasterization [WB99], or ray tracing [SAM05]. Direct volume rendering is generating an image directly from the data using ray-casting [DCH88]. This is done by sampling and mapping samples into color and opacity using a transfer function. In this section, we discuss direct volume rendering.

4.1. Ray Casting

Ray casting is commonly used due to its simplicity and the quality of the results. For each pixel in the screen, a ray is cast into the volume and samples are computed along the ray. Next, each sample is mapped into a color and opacity (RGBA values) using the transfer function [Max95]. These RGBA values are accumulated to compute the final color of the pixel. The accumulation process can be performed either in a front-to-back order or in back-to-front order. Equation 3 and 4 presents a front-to-back and back-to-front order accumulation, respectively.

$$C = \sum_{i=0}^{n} C_i \prod_{j=0}^{i-1} (1 - A_i) \tag{3}$$

Where C is the RGBA value of the pixel, C_i is the color of the

© 2019 The Author(s)

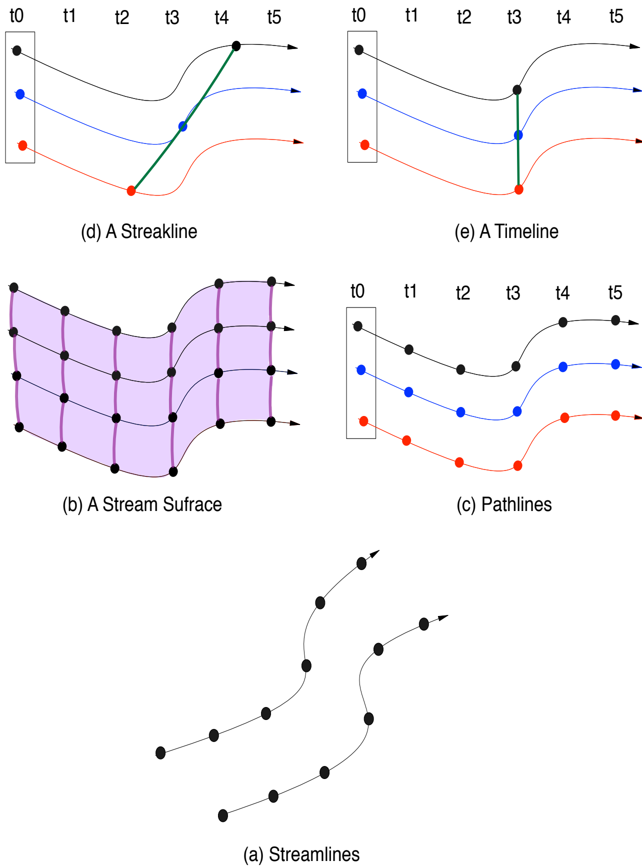

Figure 3: *Different flow visualization algorithms that use particle advection.*

current scalar value at sample i, n is the number of samples along the ray, and A_i is the opacity at sample i.

$$C = \sum_{i=n}^{0} C_i \times (1 - A) \quad (4)$$

Where C is the RGBA value of the pixel, C_i is the color of the current scalar value at sample i, n is the number of samples along the ray, and A is the accumulated opacity along the ray.

Figure 4 shows an example of the ray-casting process.

Ray-casting is expensive, thus different acceleration techniques have been used to reduce this cost. One of the most used acceleration techniques is early ray termination [HKRs*06]. Ray casting computes the color of the pixel by accumulating the colors and opacities of the samples along the ray. If the accumulated opacity is high, samples that are far from the camera will not contribute to the final color and will be hidden. The idea of early termination is to stop the compositing along the ray when the accumulated opacity is high, which reduces the total time. However, this optimization is only possible with front-to-back compositing.

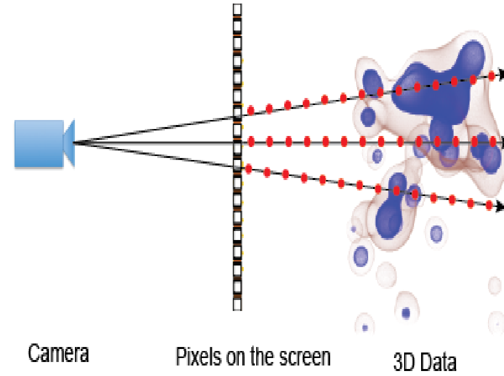

Figure 4: *Ray-Casting*

4.2. Parallelization Overview

Volume rendering is computationally expensive, and its cost increases with the size of the data set. Parallelizing such heavy computation is essential to visualize data in a timely manner. However, performing parallel ray-casting introduces new challenges, especially with respect to load balancing (Section 2.5). There are two main techniques for parallel volume rendering [MCEF94]: 1) image order (sort first), and 2) object order (sort last). In the image order technique, the parallelization happens over pixels. In the object order technique, the parallelization happens over cells (subvolumes). In this section, we start by discussing the challenges of parallel volume rendering. Next, we survey the different parallel solutions and categorize them under one of three categories: 1) image order (sort first), 2) object order (sort last), and 3) a hybrid between the first two.

The performance of ray-casting depends on two components: 1) the number of cells, and 2) the number of samples. These two components are heavily impacted by four factors, each of which can cause significant load imbalances and influence the choice of parallelization method. These four factors are the following: 1) camera position, 2) camera view is changing, 3) image size, and 4) data Set size.

- **Camera Position:** It impacts the performance in two points: 1) which part of the data is visible, and 2) the number of samples per cells (cell sizes). If the camera is zoomed in, it implies: 1) there are no empty pixels, and 2) cells that are in the camera view have more samples (larger cells). If the camera is zoomed out, it implies: 1) there are empty pixels, and 2) cells have a similar number of samples (equal sizes). Image order performs well when the camera is zoomed in since there are no empty pixels. However, it performs poorly when the camera is zoomed out since there are parts of the image that are empty. On the other hand, object order performs well when the camera is zoomed out because the cells are distributed evenly among processor and most of the cells are in the camera view. However, it can suffer from load imbalance when the camera is inside the volume because only the processors having visible cells (in the camera view) will do the work (larger cells).
- **Moving Camera View :** If the camera view is changing between

© 2019 The Author(s)

frames, the visible portion of the data changes between frames. The image order technique is expensive with this configuration because it requires to redistribute data blocks among processors for every new camera view. In some cases, the data is replicated to avoid redistributing the blocks, but this becomes challenging when the size of the data is large and cannot fit into a single memory. On the other hand, object order works well for cases where the camera view frequently changes since each processor works on its cells independently from the camera view.
- **Image Size:** In order to produce the final pixel color, a processor needs to have all the data required for that pixel. In image order, each processor has the data required to produce its part of the image; no exchange is needed between processors. In object order, processors need to exchange samples (i.e., image compositing) to calculate the final color of the pixel. The communication cost of this step is expensive and could become a bottleneck when the size of the image is large. Thus image order works better than object order for large image sizes.
- **Data Set Size:** If the data size is small enough to fit into a single memory, data can be replicated when using image order. As the size increases, using image order becomes difficult and could add additional costs of redistributing data blocks. Object order offers scalability when the data size is large.

4.3. Image Order

In the image order technique, pixels are distributed among processors in groups of consecutive pixels, also known as tiles. Each processor is responsible for loading and sampling the cells that contribute to its tile. Then, each processor generates a sub-image corresponding to its tile. The sub-images from all the processors are then collected onto one processor to produce the final image.

This technique allows each processor to generate its sub-image independently, avoiding the communication cost of image compositing. Load imbalance can occur if processors have un-equal cell distribution. This can happen when some tiles have more cells than others, which means some processors are performing more work than others, resulting in load imbalance. Different solutions have been proposed to avoid load imbalance by introducing additional steps to guarantee equal cells distribution.

Samanta et al. [SZF*99] presented a solution that reduced the probability of un-equal cell distribution by using virtual tiles. These virtual tiles are flexible in their shapes and size depending on the workload. Their solution maintained load balance by assigning similar cell load to each processor.

Erol et al. [EEP11] used a dynamic load balancing method to maintain load balance. Their algorithm divided the workload into tiles and used the previous rendering times to distribute the tiles among processors.

Moloney et al. [MAWM11] reduced load imbalance by introducing a bricking step. In this step, the data is divided into bricks, and bricks outside the view frustum are excluded. Next, the view frustum is divided between processors and each processor sampled the bricks within its view. Using the bricking step divides the visible part of the image among processors and eliminates assigning a processor an empty tile.

While the previous solutions maintained load balance which improved the performance, all of these solutions needed a pre-processing step and some included redistribution of the data. Both [SZF*99] and [EEP11] required a pre-processing step to determine the load of different tiles, and have the cost of redistributing the data. The third solution, [MAWM11], required performing camera transformation to determine visible data, which avoided the cost of redistributing the data.

4.4. Object Order

Object order is the most common technique for parallel volume rendering. With the object order approach, data is divided into blocks and distributed among processors. Each processor starts sampling the cells of its blocks independently of the other processors. Next, samples from all processors are composited to produce the final image.

Unlike the image order technique, this technique requires processors to communicate with each other to do the final compositing (i.e., image compositing), which could become a bottleneck [BCH12]. Load imbalance can occur if processors have un-equal samples distribution. This can happen when dealing with unstructured data. Unstructured data have different cell sizes creating different workloads: one processor could have large cells thus more workload. Different solutions have been proposed to avoid load imbalance by introducing additional steps to guarantee equal samples distribution.

Marchesin et al. [MMD06] presented a solution to guarantee load balance by performing an estimation step. In their solution, they divided data into blocks and discard any blocks that were outside the camera view or blocks that were invisible. Next, the remaining blocks were distributed among processors, and each processor sampled its blocks. Finally, binary swap [MPHK94] was used as an image compositing method.

Ma et al. [MC] presented a solution that used round robin cells assignment to perform interleaved cell partitioning. This assignment reduces the probability of load imbalance since usually, cells that are spatially close have similar sizes. Assigning these cells to different processors helps to avoid heavy workload for some processors. In addition, this assignment achieved load balance when the camera is zoomed into a region of the data. Samples from different processors are stored in a linked list. To allow for early compositing of the samples, processors sample the cells in the same region at the same time.

Steiner et al. [SPEP16] achieved load balance by using a work package pulling mechanism [EMP09]. In their solution, work was divided into equal packages and inserted into a queue. Clients asked the server for work whenever they are done with their assigned workload.

Muller et al. [MSE06] used a dynamic load balancing technique. Their method calculated the balance of each processor while sampling the cells. Data were redistributed between processors to achieve load balance.

Most of the presented solutions focused on how to improve blocks assignment to processors, which lead to better load balance.

© 2019 The Author(s)

This is done either through a pre-processing step or at runtime. The work presented by [MMD06] performed the camera transformation and had to use an estimation step to distribute the data dynamically.

While [MSE06] achieved load balance, the cost of redistributing the data could be very expensive. This cost could be a bottleneck when the size of the data is large or if the camera is zoomed into a region of the data that belongs to one processor. This resulted in redistributing most of the data blocks in the camera view.

4.5. Hybrid Parallel Volume Rendering Solutions

Both image order and object order techniques have limitations and often can result in load imbalance. While several solutions have been proposed (Section 4.3, and 4.4) for both techniques to eliminate load imbalance, most of these solutions have additional costs such as a preprocessing step or redistribution of the data. Using a hybrid solution to overcome the limitations that both techniques have individually, and can reduce load imbalance at a lower cost.

Montani et al. [MPS93] presented a hybrid solution, where they used an image order distribution followed by an object order. In their work, nodes are divided into clusters, and the pixels are distributed among clusters using the image order technique. Each cluster loads the data contributing to its pixels, and data are distributed among nodes of the cluster using the object order technique. Their solution reduces the potential of load imbalance compared to traditional techniques, in addition to achieving data scalability. Load imbalance can still occur either at the clusters level or at the nodes level. At the clusters level, load imbalance can occur if some clusters were assigned an empty tile. At the nodes level, load imbalance can occur if some nodes of the cluster are assigned larger cells that need more work than others.

Childs et al. [CDM06] presented another hybrid solution, where they used an object order distribution followed by an image order. In their solution, data were distributed among processors using the object order technique. Their solution began by categorizing cells into small and large cells, depending on the number of samples (see Section 4.2). Each processor was responsible for its own cells and classified them by comparing the number of samples with a given threshold. Next, each processor sampled small cells only. Then, pixels were distributed among processors using the image order technique. Depending on the pixels assignment, the algorithm exchanged two types of data: 1) samples that were generated from small cells, and 2) large cells that were not sampled. Next, each processor sampled the large cells contributing to its pixels. Then, samples from both sampling steps were composited generating a sub-image. Finally, sub-images were combined to produce the final image. As an extension for this algorithm, Binyahib et al. [BPL*18] presented a full evaluation of [CDM06], where they compared the hybrid solution with traditional solutions. They also improved the original algorithm, where they reduced the memory and communication costs. In addition, their solution used hybrid parallelism to improve the performance and take advantage of many core architectures.

Samanta et al. [SFLS00] presented another hybrid solution that partitioned pixels into tiles and distributed cells into groups. Their algorithm used the camera view to determine visible cells. Next, the algorithm partitioned the visible region along the longest axis, assigning cells that are in the same screen space to the same processor. This is done by having two lines at the end of each side of the longest axis. The line moved into the opposite direction until there are N tiles, each containing N cell. Finally, each tile was assigned to a processor. Their solution achieved load balance by assigning N cells and N tiles to N processors. Figure 5 shows an example of the algorithm.

Garcia et al. [GS02] presented a hybrid algorithm, where they used an object order distribution followed by an image order. Their algorithm classified processors into clusters. Then data was distributed among different clusters using the object order technique. At each cluster, pixels were distributed among processors of the cluster using the image order technique. Next, communication happened between the different clusters to perform the image compositing step and produce the final image, thus reducing the communication cost. To reduce the memory requirement, their algorithm used an interleaved loading method. Each processor loaded every N_{th} row of the data, where N is the number of processors in the cluster. This meant that processors only had a partial data set to sample. Next, each processor used this sub-data to produce its part of the image, where interpolation was used for the missing rows. While this method reduced the memory cost, it came at the cost of image quality and accuracy. Increasing the number of processors per cluster had a direct impact on the final image accuracy. This method could be used to explore new data, but it would not be accurate enough to use for generating production images. In addition, load imbalance might still occur if the camera is focused on a region of the data that belongs to one cluster.

While the solution presented by [MPS93] reduced the potential of load imbalance, this algorithm might not perform well in extreme camera conditions. For example, when the camera is inside the volume. The solution provided by [CDM06,BPL*18] performs better in these conditions, but it has additional communication cost in other camera positions such as when the camera is in the middle. The solution presented by [SFLS00] has an idle initialization time since all servers have to wait for the client to do the screen space transformation and then assign work to servers. While this algorithm might work on a small scale, it could perform poorly on a large scale. Finally, the solution presented by [GS02] reduces the potential of load imbalance and reduces the cost of the image compositing step. But load imbalance might still occur if some clusters have more work than others due to the camera view focus.

4.6. Summary

Table 2 shows a summary of the factors mentioned in Section 4.2 and the best configuration for each of these factors using image order and object order techniques. Each one of these factors impacts the choice of the technique, but these factors should be all considered when choosing a technique.

For example, [SZF*99], and [EEP11] presented solutions to redistribute the workload to avoid load imbalance when using image order for the zoomed out case. While this could achieve good results when the size of the data is small, it could become very expensive when the data size increases. Another example is [MMD06]

© 2019 The Author(s)

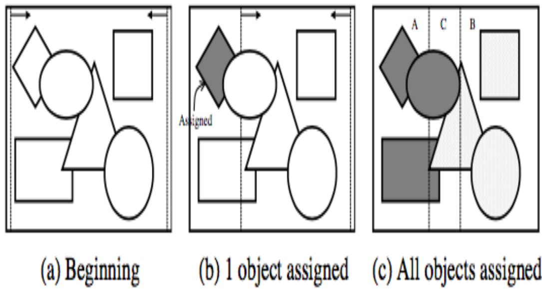

Figure 5: *Example execution of the hybrid partition algorithm [SFLS00].*

and [MSE06] solutions to reduce load imbalance for object order when the camera is inside the volume. While their solution reduced imbalance they added an additional cost of redistributing the data. Ma et al. [MC] solution avoided this cost, but it could suffer from load imbalance if the data has an unusual mesh, where the cell sizes differ in a strange pattern.

As mentioned in Section 4.2 the performance depends on the number of samples and the number of cells per processor. Load imbalance occurs when there is an uneven distribution in one of them. The hybrid solutions combined both image order and object order to limit the imbalance in these two factors. Thus they can be viable alternatives to the two traditional techniques. While these solutions improve performance and have better results, they still have some limitations or additional costs.

4.7. Unstructured Data and Volume Rendering

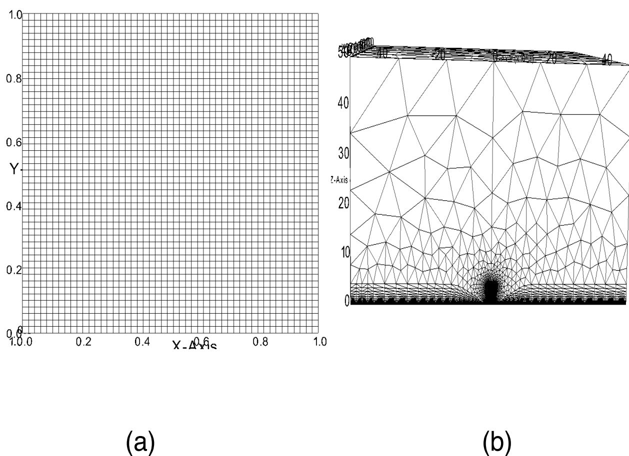

Figure 6: *Example of (a) structured, and (b) unstructured meshes.*

An unstructured mesh represents different cell sizes and sometimes different cell types in an arbitrary order. Figure 6 shows an example of structured and unstructured meshes. Unlike structured data, unstructured data does not have an implicit indexing approach, and thus the cell connectivity information is not available. This increases the complexity of volume rendering.

Different solutions have been proposed to reduce this cost. Ma [Ma95] presented an algorithm that computed the cell connectivity in a pre-processing step so it would not impact the performance while rendering. Each processor performed this step to acquire the cell connectivity information. In this step, the algorithm specified the external faces, which are faces that are not shared between cells. Next, the algorithm stored face-node, cell-face, and cell-node relationships in a hierarchical data structure. The algorithm excluded the cells that were outside the camera view. Then, each processor sampled its data. For each ray, it entered the volume from an external face, and the cell connectivity information was used to determine the next cell. A ray exited the volume when it intersected a second external face. Finally, the image compositing step was performed to exchange samples between processors and produce the final image.

Max et al. [MWSC03] proposed an algorithm that used slicing. Three slices were generated for each cell perpendicular to the X, Y, and Z axes. Depending on the camera view, one of these slices was used. While sampling, the values were computed using interpolation between the cell vertices. Next, the computed scalar values were used as 1D texture coordinates to obtain the color. Finally, the slices were rendered in back-to-front order, starting with slices that were furthest from the camera. The colors of these slices were composited to produce the final color.

Larsen et al. [LLN*15] presented an algorithm where cells were sampled in parallel using multi-threading. Cells were distributed among different processors. Each processor created a buffer that has the size of $Width \times Height \times Number of Samples per Ray$. Each processor sampled its cells in parallel and samples were stored in the buffer. The index of each sample in the buffer was computed depending on its screen space coordinates (x, y, z). Finally, in the image compositing step, processors exchange samples, and samples of each ray were composited to produce the final color.

The solution presented by Ma [Ma95] had the additional cost of the pre-processing step, which could become expensive when the data size is large. While Max et al. [MWSC03] algorithm did not have this cost, their algorithm might have a high cost at the compositing step. This is because their algorithm composited the slices in a back-to-front order, which means they cannot use the early termination technique, mentioned in Section 4.1. The algorithm introduced by Larsen et al. [LLN*15] could take advantage of the early termination techniques if the image compositing was done in front-to-back. But their algorithm can suffer from high memory cost if the size of the image ($Width \times Height$) is large and/or the number of samples is large.

5. Image Compositing

Image compositing is the final step of parallel volume rendering when using the object order technique (Section 4.4). The goal of this step is to order samples in the correct depth order to compute the final pixel color. Image compositing includes two operations: 1)

© 2019 The Author(s)

Table 2: *Factors impacting the performance of parallel volume rendering, and the best configuration for each of these factors using image order and object order techniques.*

	Image Order	Object Order
Camera Position	zoom in	zoom out
Moving Camera View	No	Yes
Image Size	Large	Small
Data size	Small-Medium	Large

communicating samples between processors, 2) compositing these samples to produce the color of the pixel. Image compositing is expensive and can become the bottleneck of the object order approach [BCH12]. Thus several solutions have been proposed to reduce the cost of this step. In this section, we survey and compare these solutions.

5.1. Image Compositing Methods

There are three main image compositing methods: 1) direct send, 2) binary swap, and 3) radix-k.

The most straightforward method to implement is direct send [EP07], where all processors communicated with each other. In this method, image pixels were assigned to processors, where each processor was responsible for compositing a part of the image. Depending on this assignment, processors exchanged data. Figure 7 shows an example of a direct send compositing between four processors. While direct send is easy to implement it could be inefficient with a large number of processors since all processors are communicating with each other.

Another image compositing method is binary swap [MPHK94]. This method required the number of processors to be a power of two. In this method, the communication between processors happened in rounds. The algorithm performed $log_2(N)$ rounds, where N is the number of processors. Processors communicated in pairs, and each round the pairs were swapped. At each round, the size of the exchanged tiles was reduced by half. Figure 8 shows an example of a binary swap compositing between four processors. Binary swap reduced network congestion and had good scalability [BCH12], but it had the limitation of requiring the number of processors to be a power of two. Thus an improved version, 2-3 swap, was implemented by Yu et al. [YWM08] to overcome this limitation. Their algorithm worked with any number of processors and processors communicated in rounds. At each round, processors were divided into groups of size two and three, and processors in the same group communicated with each other. This method had the flexibility in the number of processors while taking advantage of the efficiency of the binary swap. Another improved version of binary swap is 234 composite [NOF15, NOF18]. Their solution used an elimination process named 3-2 and 2-1 [RT04] that was developed for optimizing reduction for a non-power of two number of processors. The 234 compositing method divided processors into groups of size three and four. For each round, a pair of processors exchanged half the image. At the end of a round, all processors of the same group have communicated and the result from each group is two halves of the image. The total number of half images produced from all groups is a power of two. A binary swap method is applied to collect these partial images into a full image.

Peterka et al. [PGR*09] proposed another image compositing method known as radix-k. Their method also performed communication in multiple rounds. At each round, it defined a group size k_i, where i is the current round. The multiplication of the group sizes of all rounds is equal to N, where N is the number of processors. For this algorithm, the product of all k_i must be equal to N. At each round, each processor was responsible for $1/k$ of the image. Processors within a group communicated with each other using a direct send. Figure 9 shows an example of a radix-k compositing between six processors. This method avoided network congestion while providing the flexibility to work with any number of processors.

Moreland et al. [MKPH11] introduced a technique named telescoping to deal with non power of two number of processors. This technique grouped the largest power of two processors and defined it as the largest group. Then it took the largest power of two processors from the remaining processors and defined it as the second largest group. This process continued until all the processors have been assigned to a group. In each group, processors applied a compositing method, either binary swap or radix-k. Next, the smallest group sent its data to the second smallest group for compositing. The second smallest group did the compositing and sent the data to the third smallest group. This continued until all the data was sent to the largest group. They compared binary swap and radix-k using telescoping against the traditional methods, and their results showed overall improvement.

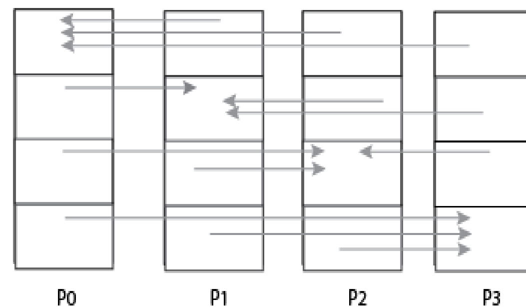

Figure 7: *Image compositing using Direct Send method between four processors.*

Direct send is flexible and easy to implement. While it has been used in several solutions, its performance can decrease when the number of processors is large due to the increase in the number of

© 2019 The Author(s)

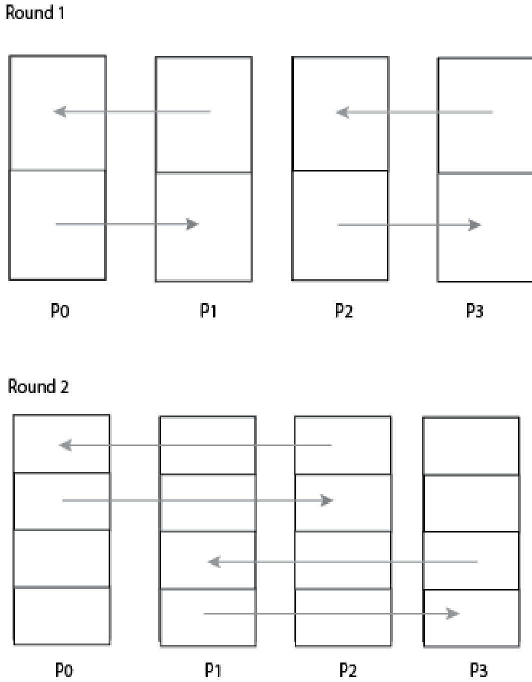

Figure 8: *Image compositing using Binary Swap method between four processors.*

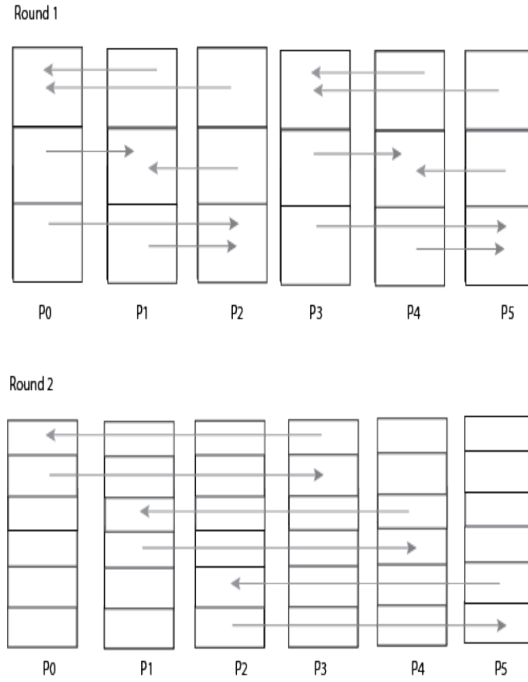

Figure 9: *Image compositing using Radix-k method between six processors and* $k = [3,2]$.

messages. Binary swap and radix-k solve this by allowing groups of processors to communicate at each round. Although this reduces the communication cost, it introduces a synchronization overhead at the end of each round.

5.2. Image Compositing Optimization

While the previous section focused on communication patterns for image compositing, in this section, we discuss optimization methods that have been presented for the compositing operation.

Active pixel encoding has been used to reduce the cost of image compositing. When using active pixel encoding, the bounding box and opacity information is used to mark inactive pixels. These pixels are removed to reduce the cost of communicating and compositing. Using this technique showed improvement in the performance in several solutions [AP98, MWP01, YYC99, TIH03, KPH*10].

Load imbalance can increase the cost of image composing. This happens when a part of the image contains more samples; thus the processor that owns this part of the image has to do more work. Thus different solutions have been proposed to reduce load imbalance. One of the methods used is interlace [MCEF94, TIH03], where non-empty pixels are distributed among processors. The data pixels are rearranged so that all processors have a similar workload. While the traditional interlace technique has its advantage, it introduces an overhead at the final step to arrange the pixels into their correct order and this overhead could be expensive when the image size increases. To reduce this cost, Moreland et al. [MKPH11] proposed an improvement. Their solution guaranteed the slices that are created during the data rearrangement are equal to the final image partitions created by the compositing method (binary swap or radix-k). Thus reducing the cost of pixels arrangement by avoiding extra copies which would have been necessary if the slices sizes did not match the final image partitions.

5.3. Image Compositing Comparative Studies

In this section, we discuss some of the papers that compared different image compositing methods.

Moreland et al. [MKPH11] compared the traditional binary swap method with different factorings of radix-k, where the group size varies. They used the Image Composition Engine for Tiles (IceT) framework [Mor11]. Their paper tested these methods at scale and added an improvement that was mentioned in the previous sections (Section 5.2). They compared binary swap and radix-k with these improvements against the traditional implementations and their results showed overall improvement.

Moreland [Mor18] presented a paper where he compared different versions of the binary swap with the IceT compositor [Mor11], which uses telescoping and radix-k. His paper focused on testing the performance when dealing with non-power of two number of processors. Variations of binary swap included 2-3 swap [YWM08], 234 swap [NOF15, NOF18], telescoping [MKPH11], a naive method, and a reminder method. The first three methods were discussed earlier in the section. The naive method finds the largest number of processors that is a power of two. Then, the remaining processors send their data to processors that are in the group and

© 2019 The Author(s)

stays idle for the rest of the communication. The reminder method applies a 3-2 reduction to the remaining processors which is similar to the one mentioned in 234 compositing [NOF15]. He ran each algorithm for multiple frames and different camera configurations. His experiments showed better performance for the telescoping and reminder methods, while the naive method performed poorly when dealing with a non-power of two number of processors. Finally, IceT showed better performance than all versions of binary swap.

5.4. Summary

There are two main factors impacting the performance of image compositing: 1) the number of processors, 2) the distribution of non-empty pixels, which is impacted by the camera position as mentioned in Section 4.2.

While different compositing and optimization methods have been proposed to improve the performance, sometimes paying the additional overheads introduced for these methods can be more expensive. When the number of processors is small enough, using direct send might result in better performance than using binary swap or radix-k. Since the number of processors is small, the probability of network congestion is low and thus it avoids the synchronization overhead introduced for more complex methods. As the number of processors increases, paying the cost of this overhead leads to better overall performance. If the distribution of non-empty pixels is dense in one region of the image (zoomed out camera position), this could lead to load imbalance. Thus using the optimization techniques mentioned in Section 5.2 and paying the additional cost can be necessary to improve the performance. Other cases show that simple solutions can be more efficient as well. According to [MKPH11] findings, the overhead of interlace could be larger than the gain when using a small number of processors. Another example is presented in [Mor18], where the author showed that the reminder algorithm gives better performance than other more complicated methods.

6. Other Visualization Techniques

In this section, we present other visualization algorithms and the challenges added by parallelism.

6.1. Contouring

One of the most used visualization techniques is iso-contours. An iso-contour displays a line or a surface representing a certain scalar value. This value is represented by an isoline in the case of 2D data or an isosurface in the case of 3D data. For example, displaying the isosurface of the density in a molecular simulation to represent the boundaries of atoms. There are different techniques for isosurface extraction; the most commonly used is Marching Cubes [LC87]. The marching cube method extracts a surface by computing triangles depending on a set of cases. Iso-contour extraction is composed of two steps: 1) the search step, and 2) the generation step. In the search step, the algorithm finds the cells containing the isovalue. In the generation step, the algorithm generate the isosurface triangles through interpolations of the cells scalar values. The computational cost of this method increases with the size of the data set. A parallel solution is therefore needed to process large data sets.

Out of core solutions have been proposed to handle large data sets. While these solutions are useful, they add additional I/O costs. Chiang et al. [CFSW01] presented an isosurface extraction algorithm that was an extension of a previous work [CSS98]. The extension included parallelizing the I/O operations and isosurface extraction. Their work introduced a concept called meta-cells. Cells that are spatially near each other were grouped into a meta-cell. Their algorithm used a preprocessing step that partitioned the dataset into spatially coherent meta-cells. These meta-cells were similar in size. Thus the cost of reading these cells from memory is similar. Each meta-cell had two lists: a list contained the vertices information, and a list contained the information of the cells . For each vertex in the first list, the algorithm stored x, y, z, and a scalar value. For each cell in the second list, the algorithm stored pointers to the vertices of the cell. Using pointers allowed the algorithm to avoid storing each vertex more than once for each meta-cell. For each meta-cell, the algorithm computed meta-intervals, each meta-interval stored min and max values. Next, the algorithm computed a binary-blocked I/O (BBIO) interval tree, which is an indexing structure. The BBIO stored meta-intervals and the meta-cell ID for each interval; this ID is a pointer to the meta-cell. The algorithm stored the meta-cells and the BBIO on the disk. During run time, the algorithm used the BBIO to find the meta-cells that intersected with the isovalue. Next, the algorithm read meta-cells from disk one at a time and generated the isosurface triangles. In their algorithm, they used a self-scheduling technique [Qui87], where one node acted as a client that assigned work to servers. The client scanned the BBIO and determined the active meta-cells. Next, the client maintained a queue of all active meta-cells. When servers had no more work, they sent a request to the client to be assigned more work. Each server read meta-cells from disk and computed isosurface triangles.

Another out of core solution was proposed by Zhang et al. [ZBB01]. Their algorithm maintained load balance by decomposing the data depending on their workloads. They used the contour spectrum [BPS97] to get the workload information. The contour spectrum is an interface that provided a workload histogram for different isovalues. The algorithm reduced the I/O time by using a new model, where instead of having one disk that can be accessed by different processors, each processor has a local disk. Thus different processors could read data from their local disks in parallel. When a processor needed data from a remote disk, data were sent by the owner processor. For each local disk, the algorithm built an I/O-optimal interval tree [AV96] as an indexing structure. During run time, each processor searched its local disk for active cells and computed isosurface triangles.

Additional challenges are arising when extracting an isosurface on an Adaptive mesh refinement (AMR) data [BC89]. Different regions of simulation data need different resolutions depending on the importance of accuracy in that region. Adaptive mesh refinement (AMR) solves this by giving a finer mesh to regions of interest. AMR data is a hierarchy of axis-aligned rectilinear grids which is more memory efficient than using unstructured grid since it does not require storing connectivity information. While AMR reduces memory cost, it can create discontinuities at boundaries when transitioning between refinement levels, thus causing cracks in the resulting isosurface. One way to prevent the formation of these cracks is by creating transition regions between the different

© 2019 The Author(s)

refinement levels [FWC*04]. Generating such transition region is difficult because of the difference of resolution between two grids and the hanging nodes (or T-junctions) caused by this difference. Hanging nodes are nodes found at the border between two grids but which only exist in the fine grid. Weber et al. [WCM12] presented a solution that used dual grids [WKL*01] to remove these discontinuities. Their implementation mapped the grid from cell-centered to vertex-centered by using the cell centers as the vertices of the vertex-centered dual grid. This resulted in a gap between the coarse grid and the fine grids, which they solved by generating stitch cells between coarse and fine regions. Figure 9 shows a 2D dual grid before and after stitch cell generation. The approach used a case table to determine how to connect vertices to form suitable stitches. Performing isosurface extraction in parallel can lead to artifacts around the boundaries of the different data blocks. This happens because a processor does not necessarily own all the neighboring cells of its local cells. Instead, some neighboring cells can be owned by other processors. Thus their algorithm used ghost cells to avoid these artifacts. Since AMR data has different resolution levels, the algorithm decomposed data into boxes, where each box had one level only. Data was distributed among different processors and each processor performed the iso-surface extraction and generated stitch cells for its local data.

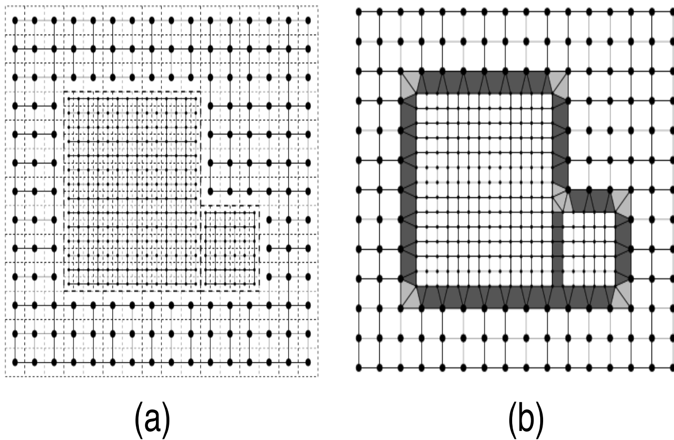

Figure 10: *A Dual grid with three refinement levels. (a) Before stitch cell generation, the original AMR grids are drawn in dashed lines and the dual grids in solid lines. (b) After stitch cell generation. [WKL*01]*

The solution proposed by Chiang et al. [CFSW01] maintained load balance by using the self-scheduling technique (one client assigns the work to all the servers). However, this technique can become inefficient at large scale because many servers have to communicate with a single client. This might lead to having high communication cost. The solution proposed by Zhang et al. [ZBB01] used a pre-processing step to guarantee load balance across processors. Despite the addition of a pre-processing step, high communication cost could still happen because of block exchanges between processors which can be expensive for large data sets.

The solution proposed by Weber et al. [WCM12] for AMR data is efficient, but it is dependent on the existence of ghost data. In cases where ghost data was not generated by the simulation code, it needs to be dynamically generated, which can increase the total execution time.

6.2. Connected Components Computation

Connected components are used in several image processing techniques. For a given graph, a connected component is a group of vertices that are connected through an edge forming a sub-graph. Figure 11 shows a graph with four components, one of which is a single vertex. Connected components computation is used in different visualization algorithms [GCS*12, KAC15] to perform different types of clustering. Performing connected components computation in a distributed memory setting is challenging because vertices are scattered across multiple processors. There are two main steps for computing connected components on a disturbed memory system. The first step is finding local connected components. The second step is merging local connected components into global connected components. In this section, we survey the solutions presented for connected components in scientific visualization.

Harrison et al. [HCG11] proposed an algorithm that computed connected sub-meshes of 3D data. The algorithm used the Union-find method [CLRS01] to find connected components. The union operation merges two components into one. The find operation determines which component does an element belongs to. This is used to determine if two elements are in the same component. The algorithm had four main steps. In the first step, each processor computed its local connected components using the Union-find method. In the second step, the algorithm transformed local labels for the connected components into global labels and added a new field to the mesh that stored the global label for each cell. In the third step, the algorithm re-distributed data using a binary space partitioning (BSP) [FKN88] to guarantee that cells that are topologically next to each other are assigned to the same processor. The BSP is a technique that partitions the data into N pieces, where N is the number of processors. It maintains load balance by assigning a similar number of blocks to different processors. The BSP technique partition the data set by iteratively scanning the current partition along the different axis. For each iteration, the algorithm selected and scanned an axis to find possible locations to split a region along the current axis; these locations are called pivots. The algorithm performed multiple random iterations to find a good pivot. This recursive scan operations along the axes continued until the number of regions was equal to the number of processors. Then, the Union-find method was used to compute connected components of cells, and global labels were assigned to merged cells. A record stored all the union operations. Next, the data was re-distributed as the original assignment. Finally, in the fourth step of the algorithm, all processors exchanged a list of the union operation. Next, the unions from all different processors were stored in a list, and each processor applied the Union-find method to identify the global connected components. An extension of the previous solution was presented by Harrison et al. [HWB*15]. This new algorithm improved the search method for a good pivot. The original method chose pivots randomly, which lead to performing multiple iterations to find

© 2019 The Author(s)

a good pivot. To avoid a large number of iteration, the algorithm selected the best pivot found until now, even if this pivot leads to imbalanced partitions. Instead, the improved algorithm chose five pivot points that are evenly spaced through the data. Then, the algorithm found the best two pivots of those five and placed new five pivot points between them. Next, the algorithm found the best pivot from the new five points and used that to split the region. This improvement showed better balance and less communication time.

Additional challenges are arising when computing connected components for Adaptive mesh refinement (AMR) data [BC89] since AMR data has multiple refinement levels (Section 6.1). Zou et al. [ZWB*15] proposed a solution for finding connected components in AMR data. Their algorithm considered connected cells that are in the same refinement level as a box. The algorithm had three phases. In the first phase, each processor computed local connected components of each level using the Union-find method. In the second phase, processors that had the same refinement level merged their components. One processor acted as the leader and assigned global labels to components. Next, the leader merged similar components. In the third phase, components of different refinement levels were merged. A master communicated with leaders of the different refinement levels and assigned global labels. Next, the master merged similar components together.

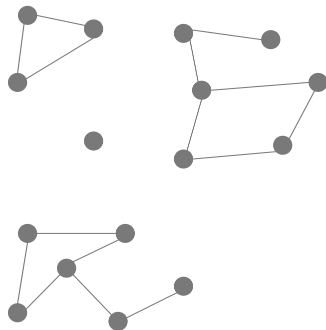

Figure 11: *A graph with four connected components.*

All of the previous solutions performed local components merge followed by a global merge. The solution presented by Harrison et al. [HCG11, HWB*15] reduced the cost of the global merge by reassigning data blocks and doing a second local components merge. The solution proposed by Zou et al. [ZWB*15] addressed the additional problem of having multiple refinement levels by performing two global merges. The second merge operation could become a bottleneck when the number of refinement levels increases; thus increasing the number of leaders communicating with the master.

6.3. Halo finding

Cosmological simulations have been used to study the evolution of the universe. These simulations usually contain a large amount of data making it hard for scientists to analyze. One of the most used algorithms in analyzing such data is finding dark matter halos. Dark matter halos are halos that contain dense dark matter particles. The most common techniques for finding halos in distributed memory solutions are: 1) friends-of-friends (FOF) clustering [DEFW85], and 2) the HOP [EH98] technique.

In the FOF technique, the algorithm searches for all particles that are accessible via a link that is shorter than a given distance (known as the linking length) resulting in a network of linked particles. Each connected component in this network is considered a halo. Next, halos in the network are filtered using a halo size parameter, where all halos that are under the specified size are ignored. This is tested by comparing the number of particles in the halo with the halo size parameter. The advantage of this technique is that it does not make assumptions on the shape of the halo and thus results in arbitrary shapes. The disadvantage comes from the linking length parameter; if this parameter has a large value, it could merge two different halos as one.

The HOP technique takes into consideration particles density in addition to their spatial locality. The first step is calculating the normalized density of each particle using local smoothing based on the masses and distances to its nearest neighbors (64 neighbor particles by default). The kD-tree [FBF77] algorithm is used to find the nearest neighbors. Next, the algorithm links (creates a chain) each particle to its densest nearest neighbor. Chains are created between particles going from the particle with the lower density to the one with the higher density. Finally, adjacent chains are merged into a halo and particles that have a density less than a given threshold are excluded.

Due to the size and complexity of cosmological simulations, parallelization has been used for finding halos. In this section, we survey available parallel solutions and discuss some of the parallelization challenges.

An FOF halo finding algorithm was implemented in Paraview [AGL*05] based on previous sequential solutions [HAH10, WHA*11]. In this algorithm, data was distributed among processors, where each processor was responsible for finding halos in its data. Ghost data was used to make sure each processor had the neighboring particles. The algorithm used a kD-tree reduction to find halos. Each processor built a kD-tree from its particles, where each node represented a particle. The algorithm started from the leaf particles. At each iteration, if the distance between particles was less than the linking length, then these particles were joined into halos. The algorithm compared the bounding boxes of sub-trees. If the bounding box is too distant to all the points, then no checks were performed. If the bounding box is close enough to all the points, then it was merged into a halo immediately. By comparing the bounding boxes of sub-trees, the algorithm reduced the number of required checks. After processors found their local halos, a global halo finding was performed (i.e., halo stitching). In this halo stitching step, the algorithm performed a merge operation to merge the same halos located on different processors.

Another FOF solution was presented by Rasera et al. [RAC*10]. In their algorithm, data was divided into sub-cubes and distributed among processors. Each processor was responsible for finding the halos in its sub-cube. When a processor found a halo that is near the edge of the current sub-cube, it checked if other particles belonging to this halo were found in the neighboring sub-cube. This process

© 2019 The Author(s)

was repeated until all halos that were across multiple sub-cubes are merged.

Liu et al. [LLC03] presented a parallel halo finding solution that used the HOP algorithm. The algorithm used a kD-tree to find the nearest neighbor and to balance the workload. Their solution divided particles into spatially closed regions that had a similar number of particles. Next, the regions were distributed among processors, and each processor stored the global tree but was responsible for a part of the tree. Next, each processor performed the HOP algorithm on its particles and chains were created between particles. When a particle created a chain to a remote particle, it was communicated to the processor owning the remote particle. Then, each processor created its local halos by merging adjacent chains. Finally, processors exchanged their local halos, and similar halos were merged.

Another solution using the HOP algorithm was presented by Skory et al. [STNC10]. Their solution started by distributing data among processors. Next, each processor calculated the density of its particles and built a local kD-tree. The algorithm performed a padding step, where a padding region was created around each sub-data. The goal of using padding is to acquire needed neighboring particles owned by other processors but which are necessary to compute the correct set of nearest neighbors. For each sub-data, the algorithm stored the size of padded region in a global lookup table. Processors used this table to communicate needed particles to each other. Next, each processor performed the HOP algorithm on its particles and chains were created between particles. No chains are allowed from a padded particle to other particles since padded particles might have a denser neighbor on another processor. When the chains between particles ended up in padded particles, these chains were communicated to processors that had these particles. Virtual links were created between chains on the sending processor and chains on the receiving processor. Finally, the linked chains were collected globally.

The difference between the two FOF solutions [AGL*05, RAC*10] is the merging of halos. The first solution [AGL*05] merged halos at the end by checking halos across processors. The second solution [RAC*10] merged halos iteratively whenever a processor had a particle that is on the edge of the cube.

Both solutions [LLC03, STNC10] had a local halo finding step, followed by a merging step, and both solutions used a kD-tree. In the first solution [LLC03], each processor stored the global kD-tree, which could be expensive when dealing with a large data set. While in the second solution [STNC10], each processor built a local kD-tree and padding was used to guarantee correct results.

7. Supporting Infrastructure

In this section, we discuss different supporting algorithms that are used in parallel visualization.

7.1. Ghost Data

Parallel visualization algorithms usually distribute data among different processors, with each processor applying the algorithm on its sub-data. Different visualization algorithms depend on the values of neighboring cells, such as iso-contour extraction, and connected components. For example, in the case of isosurface extraction, interpolating the scalar value of a point depends on the scalar values of the neighboring cells. If a point is located on the boundaries of the sub-data, the result of the interpolation will be incomplete without considering the neighboring cells. Ghost data [ILC10, KS10] is used to allow parallelization of such algorithms. Ghost data is an extra set of cells added to the boundaries of the sub-data. These additional cells are usually only used for computations at the boundaries but are not taken into account during the rendering phase to avoid artifacts. For instance, in the case of iso surface extraction, ghost cells are used to correctly interpolate scalar values on the cells at the boundaries of the sub data, but the ghost cells themselves are not interpolated.

Different visualization tools support the use of ghost data, which has been used in previous solutions for different visualization algorithms such as isosurfaces [AGL*05, CBW*12, JPW00, WCM12], particle advection [ZGH*18], and connected components [HWB*15]. Ghost data is usually generated by the simulation code and most of visualization tools do not support the generation of the ghost data. Paraview provided a Data Decomposition (D3) filter [KM07] that generated ghost data by repartitioning the data. Patchett et al. [PNP*17] presented an algorithm to generate ghost data; this algorithm was integrated into the Visualization Toolkit (VTK) [SML96]. Each processor exchanged its external boundaries information with all other processors. Next, each processor compared its external boundaries with received external boundaries from all other processors. If the processor found an intersection, it sent the cells to the processor owning that boundary. Biddiscombe [Bid16] presented an algorithm for generating ghost data, where he integrated the partitioning library Zoltan [BCCD12] with VTK and ParaView [AGL*05]. The algorithm provided the user with a selection of ghost cell generation options.

7.2. Metadata

Many visualization systems use a data flow framework. A data flow framework executes a pipeline of operations (or modules) with data being transmitted between modules. A pipeline usually applies different visualization algorithms known as filters. The optimization required to achieve good performance for visualization algorithms varies from one algorithm to another. It is important when optimizing to take into consideration the operations performed through the pipeline. For this reason, visualization frameworks use metadata, which is a brief description of the data that improves algorithms execution. There are different forms of metadata [Mor13] including regions, and contracts.

Regions are a description of the spatial range of the whole data domain and the spatial bounds of different blocks. This information can be updated by the three pipeline passes [ABM*01] depending on the filters. For example, with a select operation, only a specific region of the data is needed. The pipeline updates the regions metadata so that only that part of the data is read.

Contract [CBB*05] is a data structure that provides optimization by allowing each filter to declare its impact. The data structure has data members that define constraints and optimizations. Each filter

© 2019 The Author(s)

in the pipeline modifies this data member to make sure it contains its constraints and optimization requirements. Before performing any of the filters, the contract is passed to each filter in the pipeline starting from the last filter. After this process is done, filters are executed with the required optimizations. Contracts can be used to specify different parameters, such as identifying ghost data, cells to exclude, type of load balancing used by the framework, etc. Different visualization tools used contracts in post-hoc [CBW*12] and in situ [WFM11b, MDRP17].

7.3. Delaunay tessellation

N-body simulations such as cosmological or molecular dynamics simulations generate particles. However, it may be necessary to derive a mesh from these particles to better analyze and visualize certain properties. This is for instance, the case in cosmology simulation to analyze the density of dark matter [PKP*12, PMP14] Delaunay tessellation [GKS92] is a geometric structure for creating a mesh from a set of points. Performing tessellation on large simulations is computationally expensive and must be performed in parallel.

Peterka et al. [PKP*12, PMP14] presented an algorithm that performed tessellation in parallel. Their algorithm distributed the data among different processors which then exchanged needed neighboring points. Each processor computed the tessellation using one of two libraries Qhull [BDDH96], or CGAL [FP09]. Then, each processor wrote the results to memory. To balance the number of points per block, a solution was proposed by Morozov et al. [MP16] where they used kD tree decomposition.

7.4. Out of Core

Out of core algorithms [AV88, Vit01] (external-memory algorithms) have been used to allow visualization of large data that does not fit into the main memory. In out of core solutions, data is divided into pieces that can fit into main memory. An out of core algorithm reads and processes one piece of data at a time. This process is known as streaming. There are two paradigms of out of core solutions [Vit01]: 1) batched computations, and 2) on-line computations. In the batched computation paradigm, there is no pre-processing step, and the entire data is streamed one piece at a time. In the on-line computation paradigm, a pre-processing step is performed, and the data is organized into a data structure to improve the search process. Using the on-line computation paradigm is effective for visualization since usually only a portion of the data contributes to the final result.

Different pre-processing techniques have been used to improve I/O efficiency. These techniques include meta-cells [CFSW01], and binary-blocked I/O interval tree (BBIO Tree) [CSS98, CS99].

The out of core model has been used in several visualization algorithms such as particle advection [CXLS11, CNLS12, CS13], and isosurfaces [CFSW01, ZBB01]. It is also used by many visualization tools such as VisIt [CBW*12], VTK [SML96], Paraview [AGL*05], and the Insight Toolkit (ITK) [LI05].

7.5. Usage of Visualization Systems

Modern visualization tools such as Paraview or VisIt support three different modes. The first one is a client-server model, where the user runs a lightweight client on a local machine and connects to a server (supercomputer) that hosts the data. The computations are performed on the server and visualization is streamed back (geometry or images) to the client machine for display. The second mode executes the entire pipeline in batch without displaying the visualization and saves images on the supercomputer. Finally, the third mode is using the local machine of the scientist exclusively. Data is transferred from the supercomputer to a local machine to execute the visualization pipeline and explore data. Even though this mode might be convenient for the end user, it is often not practical anymore due to the extreme size of today's data sets which prevent moving data outside of the supercomputer. Additionally, a local machine or small cluster would not have the computational power and/or memory to process data in a timely manner.

7.6. Hybrid Parallelism

Hybrid parallelism refers to the use of both distributed- and shared-memory techniques. A distributed memory algorithm runs multiple tasks across multiple nodes in parallel and tasks communicate via the message passing interface (MPI) [Nag05]. Multiple tasks can be running on the same node, usually one MPI task per core. A hybrid parallel algorithm run a fewer number of tasks per node (usually one per node) and use the remaining cores via threading using OpenMP [CJvdPK08], or POSIX [NBFF96]. Threads on the same node share the same memory, which allows for optimization. It is possible to take advantage of multicore CPUs with MPI only by running multiple MPI tasks per node. However, the threading programming model has proven to be more efficient. It requires less memory footprint and performs less inter-chip communication. Hybrid parallelism showed improved performance for volume rendering [HBC12, HBC10], and particle advection [PCG*09, CGC*11].

8. Conclusion

This survey covered the state of the art for multiple visualization algorithms, and supporting infrastructures.

This paper showed that many scientific visualization algorithms share common challenges at scale such as data sizes, I/O costs, efficient communication patterns, and load-balancing. These challenges are also impacted by more specific factors such as the complexity of a vector field in particles advection (Section 3.2), or the position of the camera in volume rendering (Section 4.2). Overall, this survey showed that all factors should be considered when parallelizing a visualization method.

Many solutions presented in this survey demonstrated improved performance and scalability compared to traditional methods. However, it can be difficult for an end user to select which method to use for their particular use cases because of the lack of comparison between advance methods. That is for instance the case with particle advection techniques where several efficient parallelization methods are available. Our future work will address this gap by studying and comparing multiple state of the art particle advection methods.

© 2019 The Author(s)

We hope that having a single platform to study multiple algorithms with multiple data sets will help both the end users to better select a visualization method fitting their needs, and the visualization community to evaluate where are the remaining use cases or performance issues needing to be addressed if any.

References

[ABM*01] AHRENS J., BRISLAWN K., MARTIN K., GEVECI B., LAW C. C., PAPKA M.: Large-scale data visualization using parallel data streaming. *IEEE Computer Graphics and Applications 21*, 4 (July 2001), 34–41. doi:10.1109/38.933522. 17

[AGL*05] AHRENS J., GEVECI B., LAW C., HANSEN C., JOHNSON C.: 36-paraview: An end-user tool for large-data visualization. *The visualization handbook 717* (2005). 2, 16, 17, 18

[AP98] AHRENS J., PAINTER J.: Efficient sort-last rendering using compression-based image compositing. In *in Proceedings of the 2nd Eurographics Workshop on Parallel Graphics and Visualization* (1998), pp. 145–151. 13

[AR13] AKANDE O. O., RHODES P. J.: Iteration aware prefetching for unstructured grids. In *2013 IEEE International Conference on Big Data* (Oct 2013), pp. 219–227. doi:10.1109/BigData.2013.6691578. 5

[AV88] AGGARWAL A., VITTER JEFFREY S.: The input/output complexity of sorting and related problems. *Commun. ACM 31*, 9 (Sept. 1988), 1116–1127. URL: http://doi.acm.org/10.1145/48529.48535, doi:10.1145/48529.48535. 18

[AV96] ARGE L., VITTER J. S.: Optimal dynamic interval management in external memory (extended abstract). In *FOCS* (1996). 14

[BAA*16] BAUER A. C., ABBASI H., AHRENS J., CHILDS H., GEVECI B., KLASKY S., MORELAND K., O'LEARY P., VISHWANATH V., WHITLOCK B., BETHEL E. W.: 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* (Goslar Germany, Germany, 2016), EuroVis '16, Eurographics Association, pp. 577–597. URL: https://doi.org/10.1111/cgf.12930, doi:10.1111/cgf.12930. 2

[BB87] BERGER, BOKHARI: A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers C-36*, 5 (May 1987), 570–580. doi:10.1109/TC.1987.1676942. 5

[BC89] BERGER M., COLELLA P.: Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics 82*, 1 (1989), 64 – 84. URL: http://www.sciencedirect.com/science/article/pii/0021999189900351, doi:https://doi.org/10.1016/0021-9991(89)90035-1. 14, 16

[BCCD12] BOMAN E. G., CATALYUREK U. V., CHEVALIER C., DEVINE K. D.: The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring. *Scientific Programming 20*, 2 (2012), 129–150. 17

[BCH12] BETHEL E. W., CHILDS H., HANSEN C.: *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*, 1st ed. Chapman & Hall/CRC, 2012. 9, 12

[BDDH96] BARBER C. B., DOBKIN D. P., DOBKIN D. P., HUHDANPAA H.: The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw. 22*, 4 (Dec. 1996), 469–483. URL: http://doi.acm.org/10.1145/235815.235821, doi:10.1145/235815.235821. 18

[Bid16] BIDDISCOMBE J.: High-Performance Mesh Partitioning and Ghost Cell Generation for Visualization Software. In *Eurographics Symposium on Parallel Graphics and Visualization* (2016), Gobbetti E., Bethel W., (Eds.), The Eurographics Association. doi:10.2312/pgv.20161181. 17

[BL99] BLUMOFE R. D., LEISERSON C. E.: Scheduling multithreaded computations by work stealing. *J. ACM 46*, 5 (Sept. 1999), 720–748. URL: http://doi.acm.org/10.1145/324133.324234, doi:10.1145/324133.324234. 5

[BPL*18] BINYAHIB R., PETERKA T., LARSEN M., MA K., CHILDS H.: A scalable hybrid scheme for ray-casting of unstructured volume data. *IEEE Transactions on Visualization and Computer Graphics* (2018), 1–1. doi:10.1109/TVCG.2018.2833113. 10

[BPS97] BAJAJ C. L., PASCUCCI V., SCHIKORE D. R.: The contour spectrum. In *Proceedings. Visualization '97 (Cat. No. 97CB36155)* (Oct 1997), pp. 167–173. doi:10.1109/VISUAL.1997.663875. 14

[CBB*05] CHILDS H., BRUGGER E., BONNELL K., MEREDITH J., MILLER M., WHITLOCK B., MAX N.: A contract based system for large data visualization. In *VIS 05. IEEE Visualization, 2005.* (Oct 2005), pp. 191–198. doi:10.1109/VISUAL.2005.1532795. 2, 17

[CBW*12] CHILDS H., BRUGGER E., WHITLOCK B., MEREDITH J., AHERN S., PUGMIRE D., BIAGAS K., MILLER M., HARRISON C., WEBER G. H., KRISHNAN H., FOGAL T., SANDERSON A., GARTH C., BETHEL E. W., CAMP D., RÜBEL O., DURANT M., FAVRE J. M., NAVRÁTIL P.: VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*. Oct 2012, pp. 357–372. 2, 6, 17, 18

[CCC*11] CAMP D., CHILDS H., CHOURASIA A., GARTH C., JOY K. I.: Evaluating the benefits of an extended memory hierarchy for parallel streamline algorithms. In *2011 IEEE Symposium on Large Data Analysis and Visualization* (Oct 2011), pp. 57–64. doi:10.1109/LDAV.2011.6092318. 5

[CCG*12] CAMP D., CHILDS H., GARTH C., PUGMIRE D., JOY K. I.: Parallel stream surface computation for large data sets. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)* (Oct 2012), pp. 39–47. doi:10.1109/LDAV.2012.6378974. 6, 7

[CDM06] CHILDS H., DUCHAINEAU M., MA K.-L.: A scalable, hybrid scheme for volume rendering massive data sets. 153–161. URL: http://dx.doi.org/10.2312/EGPGV/EGPGV06/153-161, doi:10.2312/EGPGV/EGPGV06/153-161. 10

[CF08] CHEN L., FUJISHIRO I.: Optimizing parallel performance of streamline visualization for large distributed flow datasets. In *2008 IEEE Pacific Visualization Symposium* (March 2008), pp. 87–94. doi:10.1109/PACIFICVIS.2008.4475463. 4, 5, 6

[CFSW01] CHIANG Y.-J., FARIAS R., SILVA C. T., WEI B.: A unified infrastructure for parallel out-of-core isosurface extraction and volume rendering of unstructured grids. In *Proceedings IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics (Cat. No.01EX520)* (Oct 2001), pp. 59–151. doi:10.1109/PVGS.2001.964405. 14, 15, 18

[CGC*11] CAMP D., GARTH C., CHILDS H., PUGMIRE D., JOY K.: Streamline integration using mpi-hybrid parallelism on a large multicore architecture. *IEEE Transactions on Visualization and Computer Graphics 17*, 11 (Nov 2011), 1702–1713. doi:10.1109/TVCG.2010.259. 6, 18

[CJvdPK08] CHAPMAN B., JOST G., VAN DER PAS R., KUCK D.: *Using OpenMP: Portable Shared Memory Parallel Programming*. No. v. 10 in Scientific Computation Series. Books24x7.com, 2008. URL: https://books.google.com/books?id=MeFLQSKmaJYC. 18

[CLRS01] CORMEN T. H., LEISERSON C. E., RIVEST R. L., STEIN C.: Introduction to algorithms second edition, 2001. 15

[CMNR07] CLYNE J., MININNI P. D., NORTON A., RAST M.: Interactive desktop analysis of high resolution simulations : application to turbulent plume dynamics and current sheet formation. 2

[CNLS12] CHEN C., NOUANESENGSY B., LEE T., SHEN H.: Flow-guided file layout for out-of-core pathline computation. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)* (Oct 2012), pp. 109–112. doi:10.1109/LDAV.2012.6378984. 5, 6, 7, 18

[CPA*10a] CHILDS H., PUGMIRE D., AHERN S., WHITLOCK B.,

© 2019 The Author(s)

HOWISON M., , WEBER G. H., BETHEL E. W.: Extreme scaling of production visualization software on diverse architectures. *IEEE Computer Graphics and Applications 30*, 3 (May 2010), 22–31. doi:10.1109/MCG.2010.51. 2

[CPA*10b] CHILDS H., PUGMIRE D., AHERN S., WHITLOCK B., HOWISON M., PRABHAT, WEBER G. H., BETHEL E. W.: Extreme scaling of production visualization software on diverse architectures. *IEEE Computer Graphics and Applications 30*, 3 (May 2010), 22–31. doi:10.1109/MCG.2010.51. 5

[CS99] CHIANG Y.-J., SILVA C. T.: External memory algorithms. American Mathematical Society, Boston, MA, USA, 1999, ch. External Memory Techniques for Isosurface Extraction in Scientific Visualization, pp. 247–277. URL: http://dl.acm.org/citation.cfm?id=327766.327800. 18

[CS13] CHEN C., SHEN H.: Graph-based seed scheduling for out-of-core ftle and pathline computation. In *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)* (Oct 2013), pp. 15–23. doi:10.1109/LDAV.2013.6675154. 5, 6, 7, 18

[CSS98] CHIANG Y.-J., SILVA C. T., SCHROEDER W. J.: Interactive out-of-core isosurface extraction. In *Proceedings of the Conference on Visualization '98* (Los Alamitos, CA, USA, 1998), VIS '98, IEEE Computer Society Press, pp. 167–174. URL: http://dl.acm.org/citation.cfm?id=288216.288241. 14, 18

[CXLS11] CHEN C., XU L., LEE T., SHEN H.: A flow-guided file layout for out-of-core streamline computation. In *2011 IEEE Symposium on Large Data Analysis and Visualization* (Oct 2011), pp. 115–116. doi:10.1109/LDAV.2011.6092326. 5, 6, 7, 18

[DCH88] DREBIN R. A., CARPENTER L., HANRAHAN P.: Volume rendering. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1988), SIGGRAPH '88, ACM, pp. 65–74. URL: http://doi.acm.org/10.1145/54852.378484, doi:10.1145/54852.378484. 7

[DEFW85] DAVIS M., EFSTATHIOU G., FRENK C. S., WHITE S. D.: The evolution of large-scale structure in a universe dominated by cold dark matter. *The Astrophysical Journal 292* (1985), 371–394. 16

[DLS*09] DINAN J., LARKINS D. B., SADAYAPPAN P., KRISHNAMOORTHY S., NIEPLOCHA J.: Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), SC '09, ACM, pp. 53:1–53:11. URL: http://doi.acm.org/10.1145/1654059.1654113, doi:10.1145/1654059.1654113. 5

[EEP11] EROL F., EILEMANN S., PAJAROLA R.: Cross-Segment Load Balancing in Parallel Rendering. In *Eurographics Symposium on Parallel Graphics and Visualization* (2011), Kuhlen T., Pajarola R., Zhou K., (Eds.), The Eurographics Association. doi:10.2312/EGPGV/EGPGV11/041-050. 9, 10

[EH98] EISENSTEIN D. J., HUT P.: HOP: A new group-finding algorithm forN-body simulations. *The Astrophysical Journal 498*, 1 (may 1998), 137–142. doi:10.1086/305535. 16

[EMP09] EILEMANN S., MAKHINYA M., PAJAROLA R.: Equalizer: A scalable parallel rendering framework. *IEEE Transactions on Visualization and Computer Graphics 15*, 3 (May 2009), 436–452. doi:10.1109/TVCG.2008.104. 9

[EP07] EILEMANN S., PAJAROLA R.: Direct send compositing for parallel sort-last rendering. In *Proceedings of the 7th Eurographics Conference on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2007), EGPGV '07, Eurographics Association, pp. 29–36. URL: http://dx.doi.org/10.2312/EGPGV/EGPGV07/029-036, doi:10.2312/EGPGV/EGPGV07/029-036. 12

[FBF77] FRIEDMAN J. H., BENTLEY J. L., FINKEL R. A.: An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw. 3*, 3 (Sept. 1977), 209–226. URL: http://doi.acm.org/10.1145/355744.355745, doi:10.1145/355744.355745. 16

[FKN88] FUTCHS H., KEDEM Z. M., NAYLOR B. F.: Tutorial: Computer graphics; image synthesis. Computer Science Press, Inc., New York, NY, USA, 1988, ch. On Visible Surface Generation by a Priori Tree Structures, pp. 39–48. URL: http://dl.acm.org/citation.cfm?id=95075.95084. 15

[FMT*11] FABIAN N., MORELAND K., THOMPSON D., BAUER A. C., MARION P., GEVECIK B., RASQUIN M., JANSEN K. E.: The paraview coprocessing library: A scalable, general purpose in situ visualization library. In *2011 IEEE Symposium on Large Data Analysis and Visualization* (Oct 2011), pp. 89–96. doi:10.1109/LDAV.2011.6092322. 2

[FP09] FABRI A., PION S.: Cgal - the computational geometry algorithms library. pp. 538–539. 18

[FWC*04] FANG D. C., WEBER G. H., CHILDS H., BRUGGER E. S., HAMANN B., JOY K. I.: Extracting geometrically continuous isosurfaces from adaptive mesh refinement data. In *Proceedings of 2004 Hawaii International Conference on Computer Sciences* (2004), pp. 216–224. 15

[GCS*12] GAITHER K. P., CHILDS H., SCHULZ K. W., HARRISON C., BARTH W., DONZIS D., YEUNG P.: Visual analytics for finding critical structures in massive time-varying turbulent-flow simulations. *IEEE Computer Graphics and Applications 32*, 4 (July 2012), 34–45. doi:10.1109/MCG.2012.63. 15

[GGTH07] GARTH C., GERHARDT F., TRICOCHE X., HANS H.: Efficient computation and visualization of coherent structures in fluid flow applications. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (Nov. 2007), 1464–1471. URL: https://doi.org/10.1109/TVCG.2007.70551, doi:10.1109/TVCG.2007.70551. 3

[GHS*14] GUO H., HONG F., SHU Q., ZHANG J., HUANG J., YUAN X.: Scalable lagrangian-based attribute space projection for multivariate unsteady flow data. In *2014 IEEE Pacific Visualization Symposium* (March 2014), pp. 33–40. doi:10.1109/PacificVis.2014.15. 6

[GKM*15] GROTTEL S., KRONE M., MÜLLER C., REINA G., ERTL T.: Megamol: A prototyping framework for particle-based visualization. *IEEE Transactions on Visualization and Computer Graphics 21*, 2 (Feb 2015), 201–214. doi:10.1109/TVCG.2014.2350479. 2

[GKS92] GUIBAS L. J., KNUTH D. E., SHARIR M.: Randomized incremental construction of delaunay and voronoi diagrams. *Algorithmica 7*, 1 (Jun 1992), 381–413. URL: https://doi.org/10.1007/BF01758770, doi:10.1007/BF01758770. 18

[GS02] GARCIA A., SHEN H.-W.: An interleaved parallel volume renderer with pc-clusters. In *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2002), EGPGV '02, Eurographics Association, pp. 51–59. URL: http://dl.acm.org/citation.cfm?id=569673.569682. 10

[GTS*04] GARTH C., TRICOCHE X., SALZBRUNN T., BOBACH T., SCHEUERMANN G.: Surface techniques for vortex visualization. In *Proceedings of the Sixth Joint Eurographics - IEEE TCVG Conference on Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2004), VISSYM'04, Eurographics Association, pp. 155–164. URL: http://dx.doi.org/10.2312/VisSym/VisSym04/155-164, doi:10.2312/VisSym/VisSym04/155-164. 3, 7

[GYHZ13] GUO H., YUAN X., HUANG J., ZHU X.: Coupled ensemble flow line advection and analysis. *IEEE Transactions on Visualization and Computer Graphics 19*, 12 (Dec. 2013), 2733–2742. URL: http://dx.doi.org/10.1109/TVCG.2013.144, doi:10.1109/TVCG.2013.144. 6

[HAH10] HSU C., AHRENS J. P., HEITMANN K.: Verification of the time evolution of cosmological simulations via hypothesis-driven comparative and quantitative visualization. In *2010 IEEE Pacific Visualization Symposium (PacificVis)* (March 2010), pp. 81–88. doi:10.1109/PACIFICVIS.2010.5429606. 16

© 2019 The Author(s)

[HBC10] HOWISON M., BETHEL E. W., CHILDS H.: Mpi-hybrid parallelism for volume rendering on large, multi-core systems. In *Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2010), EG PGV'10, Eurographics Association, pp. 1–10. URL: http://dx.doi.org/10.2312/EGPGV/EGPGV10/001-010, doi:10.2312/EGPGV/EGPGV10/001-010. 18

[HBC12] HOWISON M., BETHEL E. W., CHILDS H.: Hybrid parallelism for volume rendering on large-, multi-, and many-core systems. *IEEE Transactions on Visualization and Computer Graphics 18*, 1 (Jan 2012), 17–29. doi:10.1109/TVCG.2011.24. 18

[HCG11] HARRISON C., CHILDS H., GAITHER K. P.: Data-parallel mesh connected components labeling and analysis. In *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2011), EGPGV '11, Eurographics Association, pp. 131–140. URL: http://dx.doi.org/10.2312/EGPGV/EGPGV11/131-140, doi:10.2312/EGPGV/EGPGV11/131-140. 15, 16

[HKRs*06] HADWIGER M., KNISS J. M., REZK-SALAMA C., WEISKOPF D., ENGEL K.: *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006. 8

[HNW93] HAIRER E., NØRSETT S. P., WANNER G.: *Solving Ordinary Differential Equations I (2Nd Revised. Ed.): Nonstiff Problems*. Springer-Verlag New York, Inc., New York, NY, USA, 1993. 3

[Hul92] HULTQUIST J. P. M.: Constructing stream surfaces in steady 3d vector fields. In *Visualization, 1992. Visualization '92, Proceedings., IEEE Conference on* (Oct 1992), pp. 171–178. doi:10.1109/VISUAL.1992.235211. 3, 7

[HWB*15] HARRISON C., WEILER J., BLEILE R., GAITHER K., CHILDS H.: A distributed-memory algorithm for connected components labeling of simulation data. In *Topological and Statistical Methods for Complex Data* (Berlin, Heidelberg, 2015), Bennett J., Vivodtzev F., Pascucci V., (Eds.), Springer Berlin Heidelberg, pp. 3–19. 15, 16, 17

[ILC10] ISENBURG M., LINDSTROM P., CHILDS H.: Parallel and streaming generation of ghost data for structured grids. *IEEE Computer Graphics and Applications 30*, 3 (May 2010), 32–44. doi:10.1109/MCG.2010.26. 17

[JPW00] JOHNSON C. R., PARKER S. G., WEINSTEIN D.: Large-scale computational science applications using the scirun problem solving environment. In *In Supercomputer* (2000). 17

[KAC15] KRESS J., ANDERSON E., CHILDS H.: A visualization pipeline for large-scale tractography data. In *2015 IEEE 5th Symposium on Large Data Analysis and Visualization (LDAV)* (Oct 2015), pp. 115–123. doi:10.1109/LDAV.2015.7348079. 15

[KGJ09] KRISHNAN H., GARTH C., JOY K.: Time and streak surfaces for flow visualization in large time-varying data sets. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (Nov 2009), 1267–1274. doi:10.1109/TVCG.2009.190. 3

[KM07] KENNETH MORELAND LISA AVILA L. A. F.: Parallel unstructured volume rendering in paraview, 2007. URL: https://doi.org/10.1117/12.704533, doi:10.1117/12.704533. 17

[KPH*10] KENDALL W., PETERKA T., HUANG J., SHEN H.-W., ROSS R.: Accelerating and benchmarking radix-k image compositing at large scale. In *Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2010), EG PGV'10, Eurographics Association, pp. 101–110. URL: http://dx.doi.org/10.2312/EGPGV/EGPGV10/101-110, doi:10.2312/EGPGV/EGPGV10/101-110. 13

[KS10] KJOLSTAD F. B., SNIR M.: Ghost cell pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns* (New York, NY, USA, 2010), ParaPLoP '10, ACM, pp. 4:1–4:9. URL: http://doi.acm.org/10.1145/1953611.1953615, doi:10.1145/1953611.1953615. 17

[KWA*11] KENDALL W., WANG J., ALLEN M., PETERKA T., HUANG J., ERICKSON D.: Simplified parallel domain traversal. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), SC '11, ACM, pp. 10:1–10:11. URL: http://doi.acm.org/10.1145/2063384.2063397, doi:10.1145/2063384.2063397. 6

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1987), SIGGRAPH '87, ACM, pp. 163–169. URL: http://doi.acm.org/10.1145/37401.37422, doi:10.1145/37401.37422. 14

[LGZY16] LIU R., GUO H., ZHANG J., YUAN X.: Comparative visualization of vector field ensembles based on longest common subsequence. In *2016 IEEE Pacific Visualization Symposium (PacificVis)* (April 2016), pp. 96–103. doi:10.1109/PACIFICVIS.2016.7465256. 6

[LI05] LUIS IBANEZ WILL SCHROEDER L. N. J. C.: Streaming large data. in the itk software guide. 18

[LLC03] LIU Y., LIAO W.-K., CHOUDHARY A.: Design and evaluation of a parallel hop clustering algorithm for cosmological simulation. pp. 8 pp.–. doi:10.1109/IPDPS.2003.1213186. 17

[LLN*15] LARSEN M., LABASAN S., NAVRÁTIL P., MEREDITH J., CHILDS H.: Volume Rendering Via Data-Parallel Primitives. In *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)* (Cagliari, Italy, May 2015), pp. 53–62. 11

[LSP14] LU K., SHEN H., PETERKA T.: Scalable computation of stream surfaces on large scale vector fields. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Nov 2014), pp. 1008–1019. doi:10.1109/SC.2014.87. 6, 7

[Ma95] MA K.-L.: Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures. In *Proceedings of the IEEE Symposium on Parallel Rendering* (New York, NY, USA, 1995), PRS '95, ACM, pp. 23–30. URL: http://doi.acm.org/10.1145/218327.218333, doi:10.1145/218327.218333. 11

[MAWM11] MOLONEY B., AMENT M., WEISKOPF D., MOLLER T.: Sort-first parallel volume rendering. *IEEE Transactions on Visualization and Computer Graphics 17*, 8 (Aug 2011), 1164–1177. doi:10.1109/TVCG.2010.116. 9

[Max95] MAX N.: Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics 1*, 2 (June 1995), 99–108. doi:10.1109/2945.468400. 7

[MC] MA K.-L., CROCKETT T. W.: A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data. In *In Proceedings of 1997 Symposium on Parallel Rendering*, pp. 95–104. 9, 11

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications 14*, 4 (July 1994), 23–32. doi:10.1109/38.291528. 8, 13

[MCHG13] MÜLLER C., CAMP D., HENTSCHEL B., GARTH C.: Distributed parallel particle advection using work requesting. In *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)* (Oct 2013), pp. 1–6. doi:10.1109/LDAV.2013.6675152. 5, 6

[MDRP17] MOMMESSIN C., DREHER M., RAFFIN B., PETERKA T.: Automatic data filtering for in situ workflows. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)* (Sep. 2017), pp. 370–378. doi:10.1109/CLUSTER.2017.35. 18

[MKPH11] MORELAND K., KENDALL W., PETERKA T., HUANG J.: An image compositing solution at scale. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), SC '11, ACM, pp. 25:1–25:10. URL: http://doi.acm.org/10.1145/2063384.2063417, doi:10.1145/2063384.2063417. 12, 13, 14

© 2019 The Author(s)

[MLP*10] McLoughlin T., Laramee R. S., Peikert R., Post F. H., Chen M.: Over Two Decades of Integration-Based, Geometric Flow Visualization. *Computer Graphics Forum* (2010). doi:10.1111/j.1467-8659.2010.01650.x. 3

[MLZ09] McLoughlin T., Laramee R. S., Zhang E.: Easy integral surfaces: A fast, quad-based stream and path surface algorithm. In *Proceedings of the 2009 Computer Graphics International Conference* (New York, NY, USA, 2009), CGI '09, ACM, pp. 73–82. URL: http://doi.acm.org/10.1145/1629739.1629748, doi:10.1145/1629739.1629748. 3, 7

[MMD06] Marchesin S., Mongenet C., Dischler J.-M.: Dynamic load balancing for parallel volume rendering. In *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2006), EGPGV '06, Eurographics Association, pp. 43–50. URL: http://dx.doi.org/10.2312/EGPGV/EGPGV06/043-050, doi:10.2312/EGPGV/EGPGV06/043-050. 9, 10

[Mor11] Moreland K.: Icet users' guide and reference, version 2.0. technical report sand2010-7451, sandia national laboratories, January 2011. 13

[Mor13] Moreland K.: A survey of visualization pipelines. *IEEE Transactions on Visualization and Computer Graphics 19*, 3 (March 2013), 367–378. doi:10.1109/TVCG.2012.133. 17

[Mor18] Moreland K.: Comparing binary-swap algorithms for odd factors of processes. In *2018 IEEE 8th Symposium on Large Data Analysis and Visualization (LDAV)* (Oct 2018). 13, 14

[MP16] Morozov D., Peterka T.: Efficient delaunay tessellation through k-d tree decomposition. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Nov 2016), pp. 728–738. doi:10.1109/SC.2016.61. 5, 6, 18

[MPHK94] Ma K.-L., Painter J. S., Hansen C. D., Krogh M. F.: Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications 14*, 4 (July 1994), 59–68. doi:10.1109/38.291532. 9, 12

[MPS93] Montani C., Perego R., Scopigno R.: Parallel rendering of volumetric data set on distributed-memory architectures. *Concurrency: Practice and Experience 5*, 2 (1993), 153–167. 10

[MSE06] Müller C., Strengert M., Ertl T.: Optimized volume raycasting for graphics-hardware-based cluster systems. In *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2006), EGPGV '06, Eurographics Association, pp. 59–67. URL: http://dx.doi.org/10.2312/EGPGV/EGPGV06/059-066, doi:10.2312/EGPGV/EGPGV06/059-066. 9, 10, 11

[MWP01] Moreland K., Wylie B., Pavlakos C.: Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *Proceedings of the IEEE 2001 Symposium on Parallel and Large-data Visualization and Graphics* (Piscataway, NJ, USA, 2001), PVG '01, IEEE Press, pp. 85–92. URL: http://dl.acm.org/citation.cfm?id=502125.502141. 13

[MWSC03] Max N., Williams P., Silva C., Cook R.: Volume rendering for curvilinear and unstructured grids. In *Computer Graphics International, 2003. Proceedings* (2003), IEEE, pp. 210–215. 11

[Nag05] Nagle D.: Mpi – the complete reference, vol. 1, the mpi core, 2nd ed., scientific and engineering computation series, by marc snir, steve otto, steven huss-lederman, david walker and jack dongarra. *Sci. Program. 13*, 1 (Jan. 2005), 57–63. URL: http://dx.doi.org/10.1155/2005/653765, doi:10.1155/2005/653765. 18

[NBFF96] Nichols B., Buttlar D., Farrell J., Farrell J.: *PThreads Programming: A POSIX Standard for Better Multiprocessing*. A POSIX standard for better multiprocessing. O'Reilly Media, Incorporated, 1996. URL: https://books.google.com/books?id=oMtCFSnvwmoC. 18

[NLS11] Nouanesengsy B., Lee T. Y., Shen H. W.: Load-balanced parallel streamline generation on large scale vector fields. *IEEE Transactions on Visualization and Computer Graphics 17*, 12 (Dec 2011), 1785–1794. doi:10.1109/TVCG.2011.219. 4, 5, 6

[NOF15] Nonaka J., Ono K., Fujita M.: 234 scheduling of 3-2 and 2-1 eliminations for parallel image compositing using non-power-of-two number of processes. In *2015 International Conference on High Performance Computing Simulation (HPCS)* (July 2015), pp. 421–428. doi:10.1109/HPCSim.2015.7237071. 12, 13, 14

[NOF18] Nonaka J., Ono K., Fujita M.: 234compositor: A flexible parallel image compositing framework for massively parallel visualization environments. *Future Generation Computer Systems 82* (2018), 647 – 655. URL: http://www.sciencedirect.com/science/article/pii/S0167739X17302030, doi:https://doi.org/10.1016/j.future.2017.02.011. 12, 13

[PCG*09] Pugmire D., Childs H., Garth C., Ahern S., Weber G. H.: Scalable computation of streamlines on very large datasets. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (Nov 2009), pp. 1–12. doi:10.1145/1654059.1654076. 3, 6, 18

[PD81] Prince P., Dormand J.: High order embedded runge-kutta formulae. *Journal of Computational and Applied Mathematics 7*, 1 (1981), 67 – 75. URL: http://www.sciencedirect.com/science/article/pii/0771050X81900103, doi:https://doi.org/10.1016/0771-050X(81)90010-3. 3

[PF01] Pascucci V., Frank R. J.: Global static indexing for real-time exploration of very large regular grids. In *SC '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing* (Nov 2001), pp. 45–45. doi:10.1145/582034.582036. 2

[PGR*09] Peterka T., Goodell D., Ross R., Shen H. W., Thakur R.: A configurable algorithm for parallel image-compositing applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (Nov 2009), pp. 1–10. doi:10.1145/1654059.1654064. 12

[PKP*12] Peterka T., Kwan J., Pope A., Finkel H., Heitmann K., Habib S.: Meshing the universe : Identifying voids in cosmological simulations through in situ parallel voronoi tessellation. 18

[PMP14] Peterka T., Morozov D., Phillips C.: High-performance computation of distributed-memory parallel 3d voronoi and delaunay tessellation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Piscataway, NJ, USA, 2014), SC '14, IEEE Press, pp. 997–1007. URL: https://doi.org/10.1109/SC.2014.86, doi:10.1109/SC.2014.86. 18

[PNP*17] Patchett J. M., Nouanesengesy B., Pouderoux J., Ahrens J., Hagen H.: Parallel multi-layer ghost cell generation for distributed unstructured grids. In *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)* (Oct 2017), pp. 84–91. doi:10.1109/LDAV.2017.8231854. 17

[PRN*11] Peterka T., Ross R., Nouanesengsy B., Lee T. Y., Shen H. W., Kendall W., Huang J.: A study of parallel particle tracing for steady-state and time-varying flow fields. In *2011 IEEE International Parallel Distributed Processing Symposium* (May 2011), pp. 580–591. doi:10.1109/IPDPS.2011.62. 4, 5, 6

[PYR*09] Peterka T., Yu H., Ross R., Ma K., Latham R.: End-to-end study of parallel volume rendering on the ibm blue gene/p. In *2009 International Conference on Parallel Processing* (Sept 2009), pp. 566–573. doi:10.1109/ICPP.2009.27. 2

[Qui87] Quinn M. J.: *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, Inc., New York, NY, USA, 1987. 14

[RAC*10] Rasera Y., Alimi J.-M., Courtin J., Roy F., Corasaniti P. S., Fuzfa A., Boucher V.: Introducing the dark energy universe simulation series (deuss). *AIP Conference Proceedings 1241* (06 2010), 1134. doi:10.1063/1.3462610. 16, 17

© 2019 The Author(s)

[RPW*08] RUBEL O., PRABHAT, WU K., CHILDS H., MEREDITH J., GEDDES C. G. R., CORMIER-MICHEL E., AHERN S., WEBER G. H., MESSMER P., HAGEN H., HAMANN B., BETHEL E. W.: High performance multivariate visual data exploration for extremely large data. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Nov 2008), pp. 1–12. doi:10.1109/SC.2008.5214436. 2

[RT04] RABENSEIFNER R., TRÄFF J. L.: More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (Berlin, Heidelberg, 2004), Kranzlmüller D., Kacsuk P., Dongarra J., (Eds.), Springer Berlin Heidelberg, pp. 36–46. 12

[RTBS05] RHODES P. J., TANG X., BERGERON R. D., SPARR T. M.: Iteration aware prefetching for large multidimensional datasets. In *Proceedings of the 17th International Conference on Scientific and Statistical Database Management* (Berkeley, CA, US, 2005), SSDBM'2005, Lawrence Berkeley Laboratory, pp. 45–54. URL: http://dl.acm.org/citation.cfm?id=1116877.1116883. 5

[SAM05] SHIRLEY P., ASHIKHMIN M., MARSCHNER S.: *Fundamentals of Computer Graphics*. Ak Peters Series. Taylor & Francis, 2005. URL: https://books.google.com/books?id=0VOEjFmPB-0C. 7

[SFLS00] SAMANTA R., FUNKHOUSER T., LI K., SINGH J. P.: Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (New York, NY, USA, 2000), HWWS '00, ACM, pp. 97–108. URL: http://doi.acm.org/10.1145/346876.348237, doi:10.1145/346876.348237. 10, 11

[SH96] SUJUDI D., HAIMES R.: - integration of particle paths and streamlines in a spatially-decomposed computation. In *Parallel Computational Fluid Dynamics 1995*, Ecer A., Periaux J., Satdfuka N., Taylor S., (Eds.). North-Holland, Amsterdam, 1996, pp. 315 – 322. URL: http://www.sciencedirect.com/science/article/pii/B9780444823229500931, doi:https://doi.org/10.1016/B978-044482322-9/50093-1. 4

[SML96] SCHROEDER W. J., MARTIN K. M., LORENSEN W. E.: The design and implementation of an object-oriented toolkit for 3D graphics and visualization. In *VIS '96: Proceedings of the 7th conference on Visualization '96* (1996), IEEE Computer Society Press, pp. 93–ff. 2, 17, 18

[SPEP16] STEINER D., PAREDES E. G., EILEMANN S., PAJAROLA R.: Dynamic work packages in parallel rendering. In *Proceedings of the 16th Eurographics Symposium on Parallel Graphics and Visualization* (Goslar Germany, Germany, 2016), EGPGV '16, Eurographics Association, pp. 89–98. URL: https://doi.org/10.2312/pgv.20161185, doi:10.2312/pgv.20161185. 9

[SS94] SIEGELL B. S., STEENKISTE P.: Automatic generation of parallel programs with dynamic load balancing. In *Proceedings of 3rd IEEE International Symposium on High Performance Distributed Computing* (Aug 1994), pp. 166–175. doi:10.1109/HPDC.1994.340247. 3

[STNC10] SKORY S., TURK M. J., NORMAN M. L., COIL A. L.: PARALLEL HOP: A SCALABLE HALO FINDER FOR MASSIVE COSMOLOGICAL DATA SETS. *The Astrophysical Journal Supplement Series 191*, 1 (oct 2010), 43–57. 17

[SZF*99] SAMANTA R., ZHENG J., FUNKHOUSER T., LI K., SINGH J. P.: Load balancing for multi-projector rendering systems. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (New York, NY, USA, 1999), HWWS '99, ACM, pp. 107–116. URL: http://doi.acm.org/10.1145/311534.311584, doi:10.1145/311534.311584. 9, 10

[TIH03] TAKEUCHI A., INO F., HAGIHARA K.: An improvement on binary-swap compositing for sort-last parallel rendering. In *Proceedings of the 2003 ACM Symposium on Applied Computing* (New York, NY, USA, 2003), SAC '03, ACM, pp. 996–1002. URL: http://doi.acm.org/10.1145/952532.952728, doi:10.1145/952532.952728. 13

[UJK*89] UPSON C., JR. T. F., KAMINS D., LAIDLAW D. H., SCHLEGEL D., VROOM J., GURWITZ R., VAN DAM A.: The application visualization system: A computational environment for scientific visualization. *Computer Graphics and Applications 9*, 4 (July 1989), 30–42. 2

[Vit01] VITTER J. S.: External memory algorithms and data structures: Dealing with massive data. *ACM Comput. Surv. 33*, 2 (June 2001), 209–271. URL: http://doi.acm.org/10.1145/384192.384193, doi:10.1145/384192.384193. 18

[WB99] WOO M., BOARD O. A. R.: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Graphics programming. Addison-Wesley, 1999. URL: https://books.google.com/books?id=IwicQgAACAAJ. 7

[WCM12] WEBER G. H., CHILDS H., MEREDITH J. S.: Efficient parallel extraction of crack-free isosurfaces from adaptive mesh refinement (amr) data. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)* (Oct 2012), pp. 31–38. doi:10.1109/LDAV.2012.6378973. 15, 17

[WFM11a] WHITLOCK B., FAVRE J. M., MEREDITH J. S.: Parallel In Situ Coupling of Simulation with a Fully Featured Visualization System. In *Eurographics Symposium on Parallel Graphics and Visualization* (2011), Kuhlen T., Pajarola R., Zhou K., (Eds.), The Eurographics Association. doi:10.2312/EGPGV/EGPGV11/101-109. 2

[WFM11b] WHITLOCK B., FAVRE J. M., MEREDITH J. S.: Parallel in situ coupling of simulation with a fully featured visualization system. In *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2011), EGPGV '11, Eurographics Association, pp. 101–109. URL: http://dx.doi.org/10.2312/EGPGV/EGPGV11/101-109, doi:10.2312/EGPGV/EGPGV11/101-109. 18

[WHA*11] WOODRING J., HEITMANN K., AHRENS J., FASEL P., HSU C.-H., HABIB S., ADRIAN POPE A.: Analyzing and visualizing cosmological simulations with paraview. *The Astrophysical Journal Supplement Series 195* (06 2011), 11. doi:10.1088/0067-0049/195/1/11. 16

[WKL*01] WEBER G. H., KREYLOS O., LIGOCKI T. J., SHALF J., HAGEN H., HAMANN B., JOY K. I.: Extraction of crack-free isosurfaces from adaptive mesh refinement data. In *VisSym* (2001). 15

[YWM07] YU H., WANG C., MA K.: Parallel hierarchical visualization of large time-varying 3d vector fields. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing* (Nov 2007), pp. 1–12. doi:10.1145/1362622.1362655. 4, 5, 6, 7

[YWM08] YU H., WANG C., MA K.-L.: Massively parallel volume rendering using 2-3 swap image compositing. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Piscataway, NJ, USA, 2008), SC '08, IEEE Press, pp. 48:1–48:11. URL: http://dl.acm.org/citation.cfm?id=1413370.1413419. 12, 13

[YYC99] YANG D.-L., YU J.-C., CHUNG Y.-C.: Efficient compositing methods for the sort-last-sparse parallel volume rendering system on distributed memory multicomputers. In *Proceedings of the 1999 International Conference on Parallel Processing* (Sep. 1999), pp. 200–207. doi:10.1109/ICPP.1999.797405. 13

[ZBB01] ZHANG X., BAJAJ C., BLANKE W.: Scalable isosurface visualization of massive datasets on cots clusters. In *Proceedings of the IEEE 2001 Symposium on Parallel and Large-data Visualization and Graphics* (Piscataway, NJ, USA, 2001), PVG '01, IEEE Press, pp. 51–58. URL: http://dl.acm.org/citation.cfm?id=502125.502136. 14, 15, 18

[ZGH*18] ZHANG J., GUO H., HONG F., YUAN X., PETERKA T.: Dynamic load balancing based on constrained k-d tree decomposition for parallel particle tracing. *IEEE Transactions on Visualization and Computer Graphics 24*, 1 (Jan 2018), 954–963. doi:10.1109/TVCG.2017.2744059. 5, 6, 17

© 2019 The Author(s)

[ZWB*15] Zou X., Wu K., Boyuka D. A., Martin D. F., Byna S., Tang H., Bansal K., Ligocki T. J., Johansen H., Samatova N. F.: Parallel in situ detection of connected components in adaptive mesh refinement data. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (May 2015), pp. 302–312. doi:10.1109/CCGrid.2015.154. 16

© 2019 The Author(s)