2012 IEEE International Conference on Cluster Computing

A Job Scheduling Design for Visualization Services using GPU Clusters

Wei-Hsien Hsu Chun-Fu Wang Kwan-Liu Ma Hongfeng Yu Jacqueline H. Chen
Department of Computer Science, University of California, Davis
Sandia National Laboratories, Livermore
whhsu@ucdavis.edu cfwang@ucdavis.edu ma@cs.ucdavis.edu hyu@sandia.gov jhchen@sandia.gov

***Abstract*—Modern large-scale heterogeneous computers incorporating GPUs offer impressive processing capabilities. It is desirable to fully utilize such systems for serving multiple users concurrently to visualize large data at interactive rates. However, as the disparity between data transfer speed and compute speed continues to increase in heterogeneous systems, data locality becomes crucial for performance. We present a new job scheduling design to support multi-user exploration of large data in a heterogeneous computing environment, achieving near optimal data locality and minimizing I/O overhead. The targeted application is a parallel visualization system which allows multiple users to render large volumetric data sets in both interactive mode and batch mode. We present a cost model to assess the performance of parallel volume rendering and quantify the efficiency of job scheduling. We have tested our job scheduling scheme on two heterogeneous systems with different configurations. The largest test volume data used in our study has over two billion grid points. The timing results demonstrate that our design effectively improves data locality for complex multi-user job scheduling problems, leading to better overall performance of the service.**

I. Introduction

Leveraging the power of large cluster systems, such as the ones enabled by cloud computing, users can now process and analyze their large data in a scalable and cost-effective manner. The success of large-scale systems is evidenced by the steadily increasing deployment of cloud computing for commercial web-based applications [1] and for scientific applications [2]. To further increase the computing capacity and reduce the power consumption of large systems, one of the most recent trends is to employ heterogeneous processing elements, such as GPUs, to work collaboratively with CPUs. For instance, Cluster GPU Instances have been recently introduced for Amazon's EC2 to deliver the heterogeneous processing power in the cloud [3]. This new system achieves a considerable performance improvement compared to the original cloud configuration [4].

Harnessing the power of these heterogeneous and specialized co-processors makes it possible to run certain conventional compute-intensive tasks at interactive rates. Consequently, apart from traditional batch jobs such as web indexing, the requirement to support interactive jobs has recently emerged for these large systems. In this use case, multiple users are allowed to share computing resources and run processing tasks possibly interactively over a set of large data. Such centralized heterogeneous facilities should enable users to accomplish their tasks with a short turnaround time, while fully utilizing the large-scale systems.

To share a large system between users with high efficiency is not a trivial task. Although many sophisticated job scheduling techniques have been developed to improve efficiency of systems with multi-user support, most of current approaches are not designed for running on large heterogeneous systems. Compared to conventional homogeneous systems, heterogeneous systems are characterized with deep memory hierarchies and high levels of concurrency, thus incurring an increased disparity between data transfer speed and compute speed. This makes data locality become more crucial for performance. We present efficient job scheduling strategies for placing computation closer to its input data. We select parallel visualization, specifically parallel volume rendering, as our target application. It is possible to achieve interactive framerates to render large volumetric data by exploiting multiple GPUs. This capability is very much desired by many scientific applications. Our work will thus benefit a wide class of application areas beyond what is demonstrated in this paper.

Much research has gone into the issue of improving interactivity, usability, and load balance of parallel volume rendering techniques. However, most conventional methods ignore data locality problem that occurs in the process of fetching data from disks to memory, particularly main memory and video memory, before carrying out the rendering. In a single user scenario, this one-time initialization can be omitted since a user usually explores one dataset at a time during the whole session. However, the situation can become more complex in a multi-user scenario: some users may simultaneously explore different datasets in interactive mode, and some users may submit batch rendering jobs for producing animation or visualizing time-varying data. Thus the system might be overloaded with a significant number of I/O and memory swapping operations. The data fetching step cannot be neglected since it is no longer a one-time operation, and its overhead can severely impact the overall performance on both the system and user sides.

In this work, we have developed an efficient job scheduling strategy for a parallel visualization server with multi-user support. While the job scheduling problem has been

978-0-7695-4807-4/12 $26.00 © 2012 IEEE
DOI 10.1109/CLUSTER.2012.63

extensively studied for supporting multi-tasking and multi-user systems, to the best of our knowledge, no work has been conducted to apply adequate scheduling algorithms for a parallel volume rendering service for supporting both interactive and batch modes.

We also present a cost model to assess the computational costs associated with the parallel volume rendering approach. With this model, the rendering cost in the visualization system can be quantified and predicted. We have designed a heuristic approach to achieve optimal scheduling of rendering jobs, where data locality and load balance are considered. We compare our method with various existing scheduling schemes in our experiments. Our method outperforms other methods and successfully optimizes visualization services that can serve multiple users to increase machine utilization while maintaining satisfactory responsive rates. The experimental study on our design demonstrates convincingly the plausibility of shared visualization services, and also suggest a few directions for further research.

II. Background and Related Work

A. Parallel Volume Rendering

Volume rendering is a well-established technique used to visualize volumetric datasets. Among various volume rendering methods, the ray-casting volume rendering [5] is the most popular method thanks to its effectiveness in producing high quality images where the internal structures of complicated volume data can be revealed. For each screen pixel on the plane, a single ray is traversed from the eye into the volume. During ray casting, a transfer function that maps scalar values to optical properties, including color and opacity, is applied at each sample point. These colors and opacities are then accumulated along a ray to composite a final color on the screen. The ray-casting volume rendering is computationally intensive since it requires the interpolation and shading calculations for every sample point along the ray. Krüger and Wetermann [6] proposed GPU-accelerated ray-casting rendering methods. By storing the volumetric data in the video memory, and utilizing GPU hardware including vertex and fragment shaders, these renderers achieve high quality rendering quality at interactive rates.

For large volumetric datasets where the visualization cannot be computed on a single machine, parallel volume rendering methods provide feasible solutions by distributing both data and rendering calculations to multiple compute nodes. Among the three basic parallel rendering approaches, including sort-first, sort-middle, and sort-last, defined by Molnar et al. [7], sort-last parallel rendering has been widely used by visualization researchers [8]–[11] due to its simple task decomposition for achieving load balancing. In the sort-last situation, the data is split between the nodes, and each node renders its own portion of the data. Then, image compositing takes the depth information into account to form a final image from each node's rendering. The image compositing stage in sort-last approaches demands interprocessor communication, which can become very expensive because of the potentially large amount of messages exchanged. Therefore, image compositing could become a bottleneck that affects the efficiency of the sort-last parallel rendering algorithms. Much research has gone into the issue of improving the performance of image compositing. Ma et al. [12] proposed the binary swap algorithm which uses a hierarchical communication scheme to improve the performance on large scale workloads while still involving all the processors in the compositing. Yu et al. [13] has extended the binary swap algorithm by exploiting a 2-3 swap mechanism which allows using an arbitrary number of compositing processors.

One important application of parallel volume rendering is remote visualization. Remote parallel rendering servers utilize remote computational resources to visualize full-resolution datasets. Ma and Camp [14] demonstrated several latency hiding techniques including overlapped I/O, rendering, compression and transmission in the context of remote visualization. The proposed system achieves near-interactive remote visualization of time-varying data on parallel computers. Remote volume visualization has received much attention due to the rapidly increasing size of simulation data generated by high performance computers. For interactive exploration of large datasets, Freitag and Loy [15] developed theoretical performance models for three different remote visualization strategies, including image-based rendering, parallel visualization servers, and subsampling techniques, and they also examined the performance characteristics of each strategy. Toyama et al. [16] introduced a resource management system for volume visualization which efficiently handles some issues due to high-cost data transfer. Gao et al. [17] proposed an adaptive data management scheme for large data visualization which emphasizes data sharing and access across the network.

B. Job Scheduling

The job scheduling problem has been shown to be NP-complete [18], and it has been extensively studied in the literature. In general, the job scheduling problem can be classified into two categories, *static scheduling* and *dynamic scheduling*. In static scheduling, the characteristics of parallel tasks (such as estimated execution time, data dependencies, and communication pattens) are known *a priori* before run time. A static scheduling problem typically can be modeled as *direct acyclic graph*, where the weight of a node represents the execution time of a task and the weight of an edge represents the dependency between tasks. Various heuristic approaches, such as clustering algorithms [19], list-scheduling algorithms [20], and guided random search methods [21], [22], have been proposed to find near optimal solutions with polynomial-time complexity.

In dynamic scheduling, *a priori* knowledge of parallel

tasks is typically not assumed before run time, and a scheduling solution is made on-the-fly according to the current state of systems. The main goal of dynamic scheduling algorithms is to balance the workload across the processors and thus to minimize the job completion time. In general, dynamic scheduling algorithms first gather the state information of each processor and then transfer the workload from heavily loaded processors to lightly loaded ones. Such load balancing can be classified as centralized [23] or decentralized [24]. In centralized load balancing, a dedicated processor is responsible for gathering system load information and assigning tasks to processors, showing a typical master-slave structure. In decentralized load balancing, each processor exchanges workload information with other processors, and tasks are transferred between arbitrary processors in a collaborative fashion. Since the scheduling is performed at run time, an essential property of a dynamic scheduling algorithm is to lower the scheduling overhead and make it only a small portion of the overall job completion time. Many studies have shown that simple heuristics, such as *First-Come First-Served (FCFS)*, *Round Robin (RR)*, and *Shorted First (SF)*, can achieve good performance in practice [25]. A recent research direction focuses on the fairness between users in large-scale clusters [26], in which a *Fair Sharing (FS)* scheduler aims to equally allocate computational resources to the jobs over time.

In parallel visualization community, researchers have proposed various methods to balance workload according to the properties of underlying visualization algorithms. Strengert et al. [27] partitioned the data into volume bricks in object space to achieve static load balancing. Wang et al. [28] partitioned and distributed volumetric data along a hierarchical space-filling curve, and achieved well-balanced rendering workload at run time. Marchesin et al. [29] proposed a method that split data into bricks of equal size and dynamically transferred the bricks among processors according to an approximation of the rendering cost. However, these prior methods were designed to facilitate particular visualization algorithms for single-user scenarios, and their approaches cannot be directly used for multi-user support.

III. Overview

Compared with transitional job scheduling, scheduling parallel visualization in heterogeneous systems has the following characteristics. First, scheduling calculations must not incur excessive costs. A parallel visualization job is corresponding to a user interaction, and a large amount of jobs can be generated simultaneously and instantaneously from multiple users during their exploring processes. To ensure users' interactive experiences, scheduling algorithms must assign and distribute many tasks at a speed comparable to the processing speed of GPUs.

Second, given the disparity between the I/O cost and the rendering cost, scheduling algorithms must place jobs close

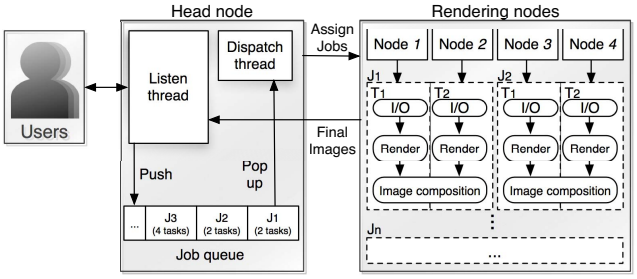


Figure 1. An overview of our system, where four rendering nodes are included for illustrative purposes.

to their data so as to minimize I/O overhead. Although there are usually no dependencies assumed between different rendering jobs, a series of interactions from the same user typically operate on the same data, thus exhibiting strong temporal coherence of data usage to guide job scheduling.

Third, scheduling algorithms need to exploit the temporal coherence of user interactions to improve job scheduling performance. A parallel visualization job can be dynamically generated, and its start time is difficult to predict. However, it is possible to approximate certain characteristics, such as the execution time, of a job from its precursors, which allows us to design an algorithm to combine the advantages of static scheduling and dynamic scheduling.

A. System Overview

Based on our observations, we design and implement our visualization system in a master-slave fashion as shown in Fig. 1. The head node is responsible for communicating with users, such as receiving rendering requests and returning final images to users. The head node is also in charge of managing the compute nodes for the rendering and image compositing operations. In general, the head node has two threads, a listening thread and a dispatching thread. When users send rendering requests to the head node, the listening thread converts the incoming requests to rendering jobs, and pushes these jobs to a job queue. At the same time, the dispatching thread pops up rendering jobs in turn from the queue. Based on a data decomposition policy and a job scheduling scheme, the dispatching thread decomposes each job into a set of independent tasks associating with data chunks, and distributes those rendering tasks to a group of rendering nodes for the rendering process. A rendering node processes the incoming rendering tasks on a *First-In-First-Out* basis. Each rendering node performs asynchronized parallel rendering to generate subimages. Then, a collective image composition operation is performed among the rendering group to generate a final image. Finally, the head node receives the final image and returns it to the user.

B. Parallel Volume Rendering Pipeline

The parallel volume rendering pipeline has three main steps, data I/O, rendering, and image compositing. The data

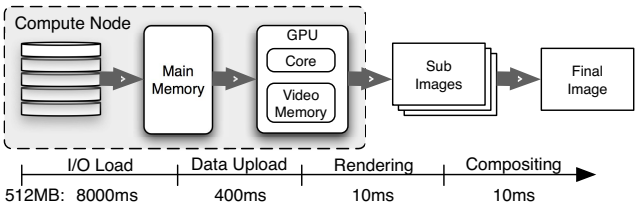


Figure 2. A typical visualization pipeline involves data I/O, rendering, and image compositing. I/O for a parallel system can be local disks or a network file server. Before the actual rendering calculation is carried out, data need to be prepared from I/O and uploaded to the GPU memory, and this initialization process usually takes several seconds and thus becomes a bottleneck in the rendering pipeline.

I/O includes data loading from the disk to the main memory, and uploading from the main memory to the video memory. Before the rendering calculation, if the associating data chunks are not loaded, the rendering nodes will make an initialization step to fetch the data chunks from the file system into the memory. Depending on the data size, the time for this step can be of the order of tens of seconds, whereas ray-casting and image compositing usually take a few miniseconds as shown in Fig. 2. In a multi-user parallel visualization system, a large amount of jobs can come from multiple users' interactions and other batch rendering requests, and thus the system can be overloaded. Without a proper job scheduling mechanism, the system may repeatedly perform I/O operations which can seriously impact the overall rendering performance. Hence, we focus on the cache locality in the main memory and develop an efficient job scheduler which minimizes the occurrences of I/O operations among all the rendering nodes, and maximize the system utilization to achieve well-balanced workload.

C. Data Decomposition

Given a data decomposition policy, a rendering job can be divided into a set of independent tasks, each with an associated data chunk. Here we discuss two decomposition strategies. The first strategy has been broadly applied in many conventional parallel visualization systems, especially for volume rendering. In this strategy, every data set is evenly partitioned into m equal-size block-shape chunks, where the number of data chunks, m, is equal to the number of rendering nodes, n. By simply distributing all the rendering tasks to all the rendering nodes, we can easily achieve balanced rendering workload and maximize system utilization without using any advanced job scheduling methods. However, using all the rendering nodes to process one rendering job at a time might be inefficient because of the unnecessary transmission overheads over the network. And due to the limited graphics memory on a GPU, the size of a data chunk must not exceed the size of graphics memory, and thus the maximal data size of the system is limited.

In order to more efficiently utilize data locality and provide high system scalability, we exploit a data decomposition strategy which divides the data into m chunks, $m = \lceil D_{size}/Chk_{max} \rceil$, where D_{size} is the data size and Chk_{max} determines the maximal chunk size. In this way, a data set is partitioned into a minimal number of chunks, where the size of each chunk is less than or equal to the maximal chunk size. By allowing more than one chunk assigned to the same rendering node, the system can potentially support an infinite size of data. Chk_{max} should not exceed the graphics memory of a rendering node. Chk_{max} should not be too small either because a small chunk size results in more chunks and transmission overheads. In our experiments, a moderate chunk size slightly less than the graphics memory usually results in satisfactory performance.

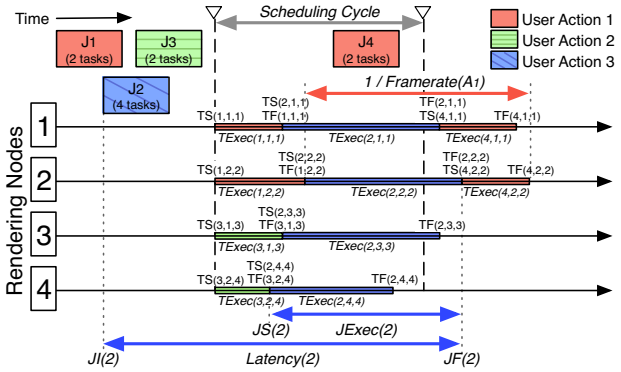


Figure 3. The illustration of our cost model. Different colors represent rendering jobs from different user actions. Jobs within a scheduling cycle are processed together to maximize the system utilization. *Latency* is the time between the job initial time and job finish time of a job. And *Framerate* is computed from successive job finish times of a user action.

IV. Cost Model and Objective

Based on our design in Section III, the system contains one head node and a set of rendering nodes φ, where $|\varphi| = p$. All the rendering nodes $(R_k, k = 1 \sim p)$ are interconnected together. The head node has a job queue consisting of a collection of independent rendering jobs $Q = \{J_1, J_2, J_3, ...\}$ that are dynamically pushed into the system and executed in parallel. Each job is associated with a dataset that can be decomposed into m chunks. We split the rendering job J_i into m tasks $\{T_{i,j}, j = 1 \sim m\}$, where each task is responsible for one chunk. Here we define a cost model to measure the performance of the parallel rendering, and quantify the efficiency of job scheduling methods.

Definition 1. Define $TS(i, j, k)$ to be the task start time of the task $T_{i,j}$ running on the rendering node R_k. The task execution time $TExec(i, j, k)$ of the task $T_{i,j}$ allocated to the node R_k can be expressed as follows:

$$TExec(i, j, k) = t_{io} + t_{render} + t_{composite}$$

where t_{io}, t_{render}, and $t_{composite}$ are times for I/O load, rendering, and image compositing, respectively. Since the

I/O time dominates the overall cost of the rendering process, the task execution time can be simplified as:

$$TExec(i,j,k) \cong t_{io} + \alpha,$$

where α is a constant that is much less than t_{io}. Notice that the I/O time can be omitted if the data chunk is already loaded in the main memory of the rendering node. The task finish time $TF(i,j,k)$ is the time when the node R_k finishes the rendering task $T_{i,j}$, and is defined as $TF(i,j,k) = TS(i,j,k) + TExec(i,j,k)$.

Definition 2. On completing execution, a task waits at a join point for sibling tasks in the same render group G to complete the whole execution. The job start time $JS(i)$ and job finish time $JF(i)$ of the job J_i are defined as the minimal $TS(i,j,k)$ and the maximal $TF(i,j,k)$ associated to J_i, respectively. The job execution time can then be expressed as $JExec(i) = JF(i) - JS(i)$

Definition 3. Define $JI(i)$ to be the job initial time of the job J_i, i.e., $JI(i)$ is the time when J_i is being issued and queued). The job latency $Latency(i)$ of the job J_i is then defined as:

$$Latency(i) = JF(i) - JI(i)$$

The latency represents the actual delay that can be noticed at the user's end. A long latency can be due to long job execution time, e.g. a job that requires to load data chunks from I/O. As a result, utilizing cache locality to minimize I/O overhead can tremendously decrease the latency. Long latencies can also happen in a busy system in which it can take time for a queued job to start being processed if the system still works on other prior jobs. In such a case, load balance is especially important so as to fully utilize the computing capacity of the cluster.

Definition 4. Given a set of interactive jobs $\mathbf{J_I} = \{J_i, i = 1 \sim n\}$ which correspond to a sequence of continuous user interactions, the frame rate for this job set is given by:

$$Framerate(\mathbf{J_I}) = (n-1) \,/\, \left(\sum_{i=1}^{n-1} JF(i+1) - JF(i) \right)$$

In our rendering service, interactive jobs associate with user actions from interactive visualization sessions, and the frame rates directly affect the user's experience and are usually used to assess the system's capability for handling interactive jobs.

In Fig. 3, we use a simple example to illustrate our cost model. In order to handle a large amount of rendering requests from multiple users and maintain the interactivity for large volumetric data visualization, an efficient scheduling method should be able to minimize the job latencies, and maximize the framerates and system throughput.

V. Scheduling Algorithm

Job scheduling for multiple machines, also known as the job shop scheduling problem, is an NP-complete problem. The exact solutions demand considerable computational power and time. In our study, a cluster has p rendering nodes and a job queue contains a total of N jobs where each job J_i is further divided into t_i independent tasks based on data decomposition. Given such a configuration, the number of ways that we can assign t_i tasks within the job J_i to p rendering nodes is p^{t_i}. Thus, the total number of mappings we can have for all the jobs is $\prod_{i=1}^{N}(p^{t_i})$. Furthermore, N jobs have the factorial $N!$ different permutations for execution. Omitting the permutations of the tasks that are parts of the same job and are assigned to the same rendering node, the total number of combinations is $N! \cdot \prod_{i=1}^{N}(p^{t_i})$. Hence it is prohibitive to perform such costly scheduling calculations in a multi-user parallel visualization system which is designed to provide interactive rendering services.

A. The Proposed Method

Instead of evaluating all the possible combinations to obtain an optimal scheduling solution, our method detects and traces the cache locality and the node availability in the system, and uses a heuristic method to perform scheduling by searching for a near optimal solution in a much reduced space. In order to trace the system status, the head node maintains three tables. The first table is a cached-data table that records the information of the data chunks resident in the main memory of each rendering node. The second table is an available-time table that records a predicted available time of each rendering node based on its current and scheduled workload. The third table is an estimated I/O-cost table that records the latest I/O time for each data chunk. These three tables are updated periodically by the head node at run time.

Unlike conventional job scheduling algorithms such as FIFO or SF, which immediately perform scheduling once a job enters the queue or the number of queued jobs reaches a scheduling window size, our method periodically perform scheduling in a constant short period of time to ensure the responsiveness for interactive user actions. The search space can be reduced based on the following heuristics:

- Tasks associated with a rendering job will be first decomposed and then scheduled individually.
- Interactive jobs within a scheduling cycle should be immediately scheduled, whereas batch jobs can be held until a rendering node becomes available.
- Interactive tasks that use the same data chunk within a scheduling cycle will be ideally scheduled to the same rendering node.
- A batch task that needs to reload the data chunk from the disk on a rendering node can be scheduled only if no interactive tasks have been assigned to this rendering node for a certain amount of time.

Table I
PARAMETERS USED IN OUR ALGORITHM

ω	Scheduling cycle
J_i	A rendering job
$T_{i,j}$	A task associated with J_i
t_i	Number of tasks within J_i
c	A data chunk
φ	A set of rendering nodes
R_k	A rendering node
ϵ	Idle time threshold for interactive tasks
$Available[R_k]$	Predicted available time of node R_k
$Cache[c]$	A set of nodes which contain caches of c
$Estimate[c]$	Estimated execution time of a task using c

Although most conventional methods perform scheduling on a job basis, our method decomposes a job into a number of tasks first and independently schedules individual tasks to optimize load balance and maximize data reuse. In order to ensure low response time for user actions, interactive jobs within a scheduling cycle need to be immediately scheduled. Batch jobs that enter the system are first held and would not be scheduled until a rendering node becomes available after processing interactive tasks.

We carefully choose the scheduling cycle ω so that interactive jobs can be scheduled timely with minimal scheduling overhead. With a proper ω configuration, the number of interactive jobs within a scheduling cycle should be controlled in a reasonable number. As a result, it is fair to assign all the tasks that use the same data chunks within the same scheduling cycle to the same rendering node. Then if more rendering jobs that use the same data are coming, the system can pick other rendering nodes in the following scheduling cycles to distribute the workload.

In our system, serving interactive rendering requests has the top priority, and the batch jobs are handled only when there are still rendering nodes available after all the interactive tasks being scheduled. For the batch tasks whose associated data chunks are cached on a particular node, the tasks are scheduled to the node until the predicted available time of the node exceeds the next scheduling time. For those tasks which require disk I/O to fetch data chunks, we use an idle time threshold ϵ to determine if a rendering node has processed an interactive task in recent past. Since disk I/O time is usually much longer than a scheduling cycle, such tasks should only be scheduled when the node has no interactive requests for a while. So the threshold can be defined as $\epsilon = Estimate[c]/2$, where $Estimate[c]$ is the estimated execution time of a task that uses the data chunk c. The pseudo code of our scheduling method is shown in Algorithm 1, in which the procedure Schedule(Q_J) is carried out every ω period of time, and Table I summarizes the notations used in the algorithm.

Algorithm 1 Schedule(an incoming job queue Q_J)

Ensure: a scheduled task queue Q_T
1: $\lambda \Leftarrow$ current time $+\omega$ # next scheduling time
2: $H_I[c \rightarrow Q_I]$ # a hash table that maps chunks to sub task queues for interactive tasks
3: $H_B[c \rightarrow Q_B]$ # a hash table that maps chunks to sub task queues for batch tasks
4: **for all** $J_i \in Q_J$ **do**
5: decompose J_i into $T_{i,j}$, $j = 1 \sim t_i$
6: push $T_{i,j}$ into $H_I[c]$ or $H_B[c]$ based on the job types
7: **end for**
scheduling interactive tasks
8: divide H_I into $H_{I,\text{Cached}}$ if $Cache[c] \neq \varnothing$, and $H_{I,\text{Non-Cached}}$ if $Cache[c] = \varnothing$
9: sort $H_{I,\text{Non-Cached}}$ based on $Estimate[c]$
10: **for all** $c \in H_{I,\text{Cached}} \cup H_{I,\text{Non-Cached}}$ **do**
11: $n \Leftarrow \min_{R_k} Available[R_k] + Estimate[c]$
12: assign all $T_{i,j}$ in $H_{I,*}[c]$ to n
13: update $Available[R_k]$
14: update $Cache[c]$ if $c \in H_{I,\text{Non-Cached}}$
15: **end for**
scheduling cached batch tasks
16: **for all** $R_k \in \varphi$ **do**
17: $Q_{R_k} \Leftarrow \forall T_{i,j} \in H_B$ if $R_k \in Cache[c]$
18: **while** $Available[R_k] < \lambda$ and $Q_{R_k} \neq \varnothing$ **do**
19: pop a $T_{i,j}$ from Q_{R_k} and assign it to R_k
20: update $Available[R_k]$
21: **end while**
22: **end for**
scheduling non-cached batch tasks
23: sort H_B based on # of nodes in $Cache[c]$
24: **for all** $R_k \in \varphi$ **do**
25: **while** $Available[R_k] < \lambda$ and $H_B \neq \varnothing$ **do**
26: **if** idle time for interactive tasks on $R_k > \epsilon$ **then**
27: pop a $T_{i,j}$ from H_B and assign it to R_k
28: update $Available[R_k]$ and $Cache[c]$
29: **end if**
30: **end while**
31: **end for**

B. Tables Update and Correction

The system keeps updating the three tables $Available$, $Cache$, and $Estimate$ at run time. The $Available$ table records the predicted available time of each node and is updated every time a task is scheduled. The available time is later corrected when a task is complete and the prediction time differs from the actual time. The $Cache$ table contains the caching information for each data chunk over the rendering nodes. The $Cache$ table on the head node is updated in the scheduling process when any of the rendering nodes is scheduled to load a new data chunk from I/O or to eliminate a cached chunk from the memory. Note that every rendering node has a system memory limit, and if a new chunk needs to be loaded but the available memory reaches its limit, the least recently used caches are released. To

perform prediction, we use the $Estimate$ table to evaluate task execution time. The $Estimate$ table is initialized via a test run and updated at run time to the latest I/O time of data chunks to reflect the state of the system.

C. Implementation

We implement our parallel visualization system in a sort-last fashion and use MPI for communication. The ray casting algorithm is implemented using OpenGL shading language on GPU [6]. The 2-3 swap method [13] is used in our system for parallel image composting. Similar to the work of Cavin et al. [30], we use three concurrent threads running on each rendering node: the rendering thread (responsible for ray casting on GPU), the compositing thread (responsible for parallel image compositing), and the communication thread (responsible for the communication with the head node). The head node periodically comminutes with the rendering nodes through MPI nonblocking function calls.

VI. Results and Discussion

A. System Specifications

We have tested our job scheduling algorithm on two heterogeneous systems with different configurations. The first system is an 8-node Linux cluster. Each node has one quad-core 3.0GHZ Intel Core 2 processor with 4 GB of memory, and one nVidia GeForce GTX 285 GPU. The second system is a 100-node GPU cluster at Argonne National Laboratory (ANL). Each node contains two quad-core 2.0GHZ Intel Xeon processors with 32 GB of memory, and two nVidia Quadro FX5600 GPUs. These two systems are representative visualization machines. In the past, they are typically employed to perform batch processing for huge volume data analysis and rendering.

B. Experiment Design

In our experimental study, we use simulation as the means for the performance evaluation of our proposed scheme. We design four scenarios that simulate real multi-user environments with mixed interactive and batch rendering requests to validate the effectiveness of our method in workload balancing and cache reusing. To evaluate the performance, we calculate the latency of every processed job, and for each series of user actions, we analyze the difference between the resulting framerate and its corresponding target value.

Scenario 1: We use 8 nodes in the first system. The memory quota of each node is constrained to 2GB, and thus there is 16GB of available memory in total. We use 6 datasets and the size of each one is 2GB, resulting in 12GB data in total. The maximal chunk size Chk_{max} is set to 512MB so that each rendering job yields 4 corresponding tasks. Since total data are smaller than total memory and thus can be completely cached, the purpose of this test is to measure the capability of workload balancing of a scheduling scheme.

Table II
Four different scenarios in our experiment

scenario	# of nodes	total memory	# of datasets	total size
1	8	16 GB	6	12 GB
2	8	16 GB	12	24 GB
3	64	512 GB	32	256 GB
4	64	512 GB	128	1 TB
scenario	**total length**	**# of batch jobs**	**# of inter-active jobs**	**target framerate**
1	60s	0	12006	33.33 fps
2	120s	2251	21011	
3	300s	9844	160633	
4	600s	35176	388481	

The target framerate of all the four scenarios is set to 33.33 fps (one request per 30ms for each action).

Scenario 2: Following Scenario 1, but we double the number of datasets and place additional batch jobs and interactive actions. The test is to verify the utilization of data locality.

Scenario 3: We use 64 nodes in the second system. The memory quota of each node is constrained to 8GB, and thus there is 512GB of available memory in total. We use 32 datasets and the size of each one is 8GB, resulting in 256GB data in total. The maximal chunk size Chk_{max} is set to 512MB so that each rendering job yields 16 corresponding tasks. The scenario simulates a light-load environment in a large-scale GPU cluster for interactive services.

Scenario 4: Following Scenario 3, this test uses 128 datasets with a total size of 1TB to simulate a heavy-load environment with numerous batch jobs and user actions.

The configurations of the four scenarios and the other profiling details are summarized in Table II. For comparison, we select the following five widely-used job scheduling policies, and modify them moderately for our application.

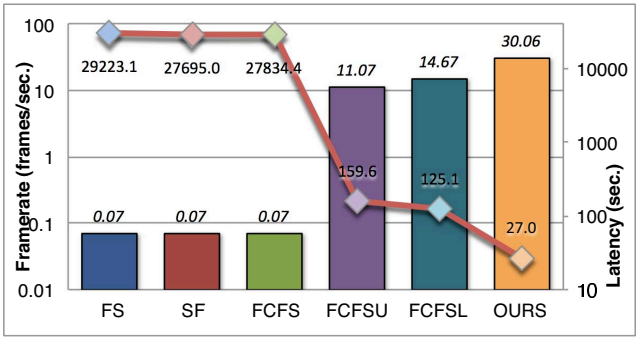
- *First-Come-First-Serve (FCFS)*: This policy schedules jobs in the order of their arrival time in the job queue. This policy also maintains an available-time table and applies the greedy strategy to assign tasks to nodes with the smallest values of available time.
- *First-Come-First-Serve* with data locality *(FCFSL)*: This policy follows the similar strategy as *FCFS*, but it takes data locality into account in its greedy search.
- *First-Come-First-Serve* with a uniform data partition and distribution *(FCFSU)*: This policy follows the same scheduling scheme as *FCFS*, but instead of partitioning data based on Chk_{max}, it uniformly partitions a dataset and distributs the associated tasks to all the nodes.
- *Shortest-First (SF)*: This policy sorts the jobs within a certain batch window based on the estimated execution time and schedules the jobs using the greedy strategy.
- *Fair-Sharing (FS)*: This policy allocates available computational resources to jobs based on estimated execution time such that each job gets an equal share of the resources on average over time. This method allows for certain levels of interactivity and is used by many distributed computing platforms such as Hadoop [26].

Figure 4. Scenario 1: This 60-second simulation is run on an 8-node Linux cluster. The bar charts and the marked lines represent the resulting interactive framerates and latencies of the six scheduling schemes, respectively. The result shows that our scheduler can effectively utilize the data locality and dynamically balance the system workload.

C. Results

Scenario 1 simulates six simultaneous user actions that periodically request rendering for six different datasets in a framerate of 33.33 fps on an 8-node Linux cluster. Fig. 4 shows the resulting framerates (bar charts) and latencies (marked lines) from the benchmarks of the six scheduling schemes. *FS*, *SF*, and *FCFS*, the methods which do not take data locality into consideration, frequently fetch data from the disk and result in poor framerates at less than 1 fps and long latencies. *FCFSU* uniformly distributes the rendering jobs and can achieve perfect data reuse because all the data can be cached in this test. However, since *FCFSU* partitions a dataset into twice the number of chunks than in other approaches, it also consumes twice as many computing resources for each single job and can only achieve nearly half of the target framerate. *OURS* and *FCFSL* dynamically allocate the tasks based on data locality and system workload, and have the best performance in both framerates and latencies. But *FCFSL* carries out the scheduling procedure on an every-job basis and thus produces slightly higher scheduling overhead than *OURS* in this test. With no batch jobs involved, this experiment shows that OURS can facilitate the interactive performance.

Scenario 2 simulates many short user actions mixed with a number of batch jobs. As shown in Fig. 5, *FS*, *SF*, and *FCFS* again result in poor interactive performance and have very high latencies for batch jobs. Since the total data size used in this test is larger than the system memory capacity, for *FCFSU* and *FCFSL*, data swapping is then required when a batch job with a different associated dataset is issued and needs to be immediately scheduled. In such a case, the interactive actions are then interrupted and require additional data swaps to resume the rendering tasks. As a result, the framerates for both *FCFSU* and *FCFSL* drop below half of the target value, and the latencies of interactive jobs tremendously increase. *OURS*, which utilizes the proposed heuristics, defers batch jobs while still handling interactive



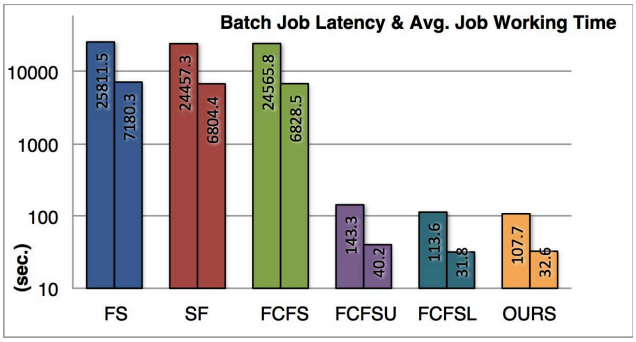


Figure 5. Scenario 2: This 120-second test simulates many short interactive user actions and a number of batch requests on an 8-node Linux cluster. The top graph illustrates the resulting interactive framerates and latencies of the six scheduling schemes. The bottom graph shows the latency (left bar) and the average working time (right bar) of each scheduling scheme for batch jobs. Although our method trades the batch throughput for a better interactive framerate, the result shows that by minimizing the total execution time, *OURS* can achieve the lowest batch job latency.

jobs and thus is able to maintain an acceptable framerate. In the bottom chart Fig. 5, the left and right bars of each schedule scheme show the average latencies and working time for batch jobs (shorter working time indicates higher throughput for batch jobs), respectively. Note that *OURS* has slightly higher average working time than *FCFSL* because *OURS* trades the batch throughput for a better interactive framerate. But by minimizing the total execution time, *OURS* still achieves the lowest batch job latency.

Scenario 3 and 4 use 64 rendering nodes on the GPU cluster at ANL. In such a system, *FCFSU* results in a considerable redundant processing overhead as the number of nodes increases. In Scenario 3, in which data can be fully cached, *FCFSU* only obtains 11.25 fps, whereas *OURS* can reach an almost-optimum framerate of 32.80 fps as shown in Fig. 6. And by deferring batch jobs, *OURS* can achieve the lowest latency of less than 1s for interactive jobs. The interactive framerate and latency of *FCFSL* are still affected by the interruption of batch jobs, although it has notably better performance on batch requests.

Scenario 4 simulates a heavy-load environment by creating a large number of rendering jobs from both interactive actions and batch requests. A total of 1TB data are accessed by 64

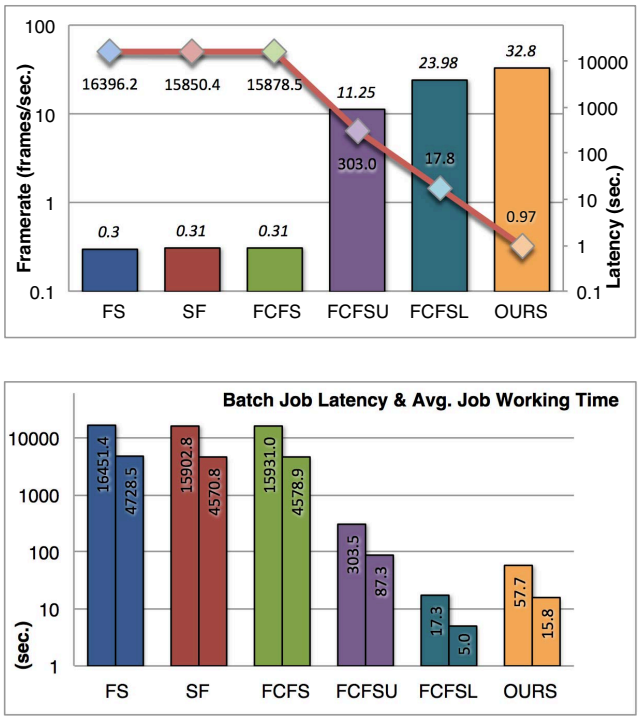


Figure 6. Scenario 3: This test uses 64 computing nodes on a 100-node GPU cluster at ANL to simulate a hybrid environment of interactive and batch requests on a large-scale cluster. The top graph illustrates the resulting interactive framerates and latencies of the six scheduling schemes. The bottom graph shows the batch latency (left bar) and the average working time (right bar). The result shows that by utilizing the proposed heuristics, which defers batch jobs, OURS achieves the lowest latency of less than 1s, and an almost-optimum framerate of 32.80 fps for interactive jobs.

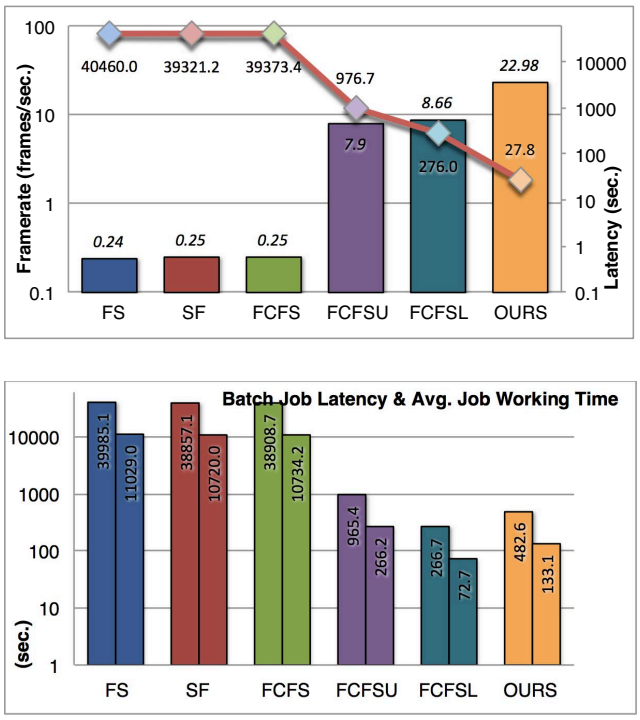


Figure 7. Scenario 4: This test simulates a heavy-load hybrid environment using 64 nodes on the GPU cluster at ANL. A total of 1TB data are requested by 423,657 rendering jobs in this 10-minute simulation. The top graph illustrates the resulting interactive framerates and latencies of the six scheduling schemes. The bottom graph shows the latency (left bar) and the average working time (right bar) of each scheduling scheme for batch jobs. The result shows that our method can achieve a high interactive framerate close to the target value, while maintaining a reasonable batch throughput.

rendering nodes in this test. *FCFSL* again suffers from a large amount of data swaps as in Scenario 2 and has a ten times longer interactive latency than *OURS*. Note that the latency of *OURS* soars up to 27.767s in this test because rendering jobs are unceasingly pushed into the system during this 10-minute simulation. But in a real scenario, users usually do not continuously make actions and would stop the interactions when they sense a lag. But even in this heavy-load simulation, *OURS* can still achieve 22.98 fps throughput for interactive rendering, which is 167.2% performance gain from FCFSL and 190.9% from FCFSU. The resulting framerates and the latencies are illustrated in Fig. 7.

Table III
DATA REUSE HIT RATES AND AVERAGE SCHEDULING COSTS

scenario		FS	FCFSU	FCFSL	OURS
1	hit rate	8.01%	99.95%	99.94%	99.94%
	avg. cost (μs)	32	60	65	33
2	hit rate	28.63%	99.86%	99.72%	99.91%
	avg. cost (μs)	36	72	74	53
3	hit rate	12.19%	99.97%	99.74%	99.91%
	avg. cost (μs)	677	2019	1002	1446
4	hit rate	10.67%	99.86%	99.51%	99.76%
	avg. cost (μs)	680	3459	1078	1392

D. Scheduling Costs and Performance

Table III shows the data reuse rates and the scheduling costs of the four schemes in each scenario. *OURS* and *FCFSU* can achieve a nearly perfect hit rate in all scenarios. *FCFSL* has a slightly lower hit rate in every case due to the data swapping between interactive and batch jobs. Other methods such as *FS* have low data reuse percentages. Note that even 0.1% of difference in the hit rates causes huge impact on the system performance because the I/O time is usually hundreds orders of magnitude larger than the rendering time. The avg. costs listed in the table represent the average time for scheduling a single job, and *OURS* has a much lower value compared to *FCFSU*. All the *First-Come-First-Serve* schemes schedule one job at a time, and the costs are independent of the number of user actions but linearly proportional to the size of the cluster. *OURS* and *FS* apply a constant scheduling cycle and can more efficiently process multiple jobs. Fig. 8 shows the comparison of the number of user actions to the scheduling costs between *OURS*, *FCFSL*, and *FCFSU*. We can see that, compared to the other methods, *OURS* can generally maintain a high hit rate while requiring a low scheduling cost to support multiple simultaneous user actions.

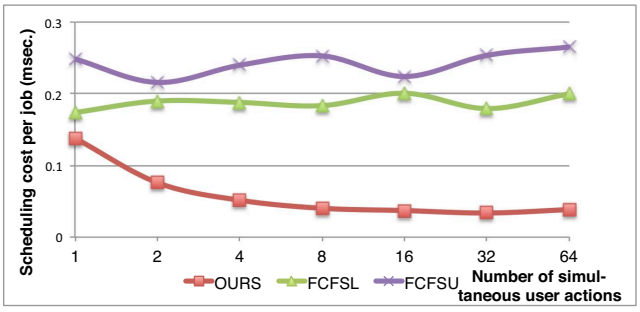


Figure 8. The comparison of the number of user actions to the scheduling costs. Our method carries out the scheduling procedure in a constant short time period and can more efficiently process incoming jobs as more simultaneous user actions are taking place. The test is run on 32 nodes on the GPU cluster at ANL, and uses 16 datasets (4GB per dataset).

However, when increasing the number and size of datasets, *OURS* requires a longer scheduling time due to the pre-processing for categorizing tasks by chunks. The complexity of our algorithm is $O(p \times m\log(m))$, where p is the number of nodes and m is the total number of data chunks. Such pre-processing is worthwhile because, first of all, the scheduling time is typically five to ten times shorter than the rendering time so such an overhead is negligible. Most importantly, the benefit gained by conducting the scheduling is so great, as shown in Fig. 9, that *OURS* can achieve the best overall performance to satisfy the target requirements.

Our scheduling method has a certain degree of fault tolerance when some of the nodes crash. By dynamically updating the $Estimate$ table to identify those unavailable nodes, the rendering can still carry on as long as the system has copies of the required data chunks on other rendering nodes. Such characteristics make our method also compatible with the distributed file systems in cloud computing. With our job scheduling scheme, multiple users are allowed to simultaneously render high resolution data at interactive or nearly interactive framerates. Fig. 10 shows three rendering examples generated by our parallel visualization system. It can be seen that our parallel visualization solution enables scientists to study fine details in large-scale data.

VII. Conclusion and Future Work

We have developed a new job scheduling method for interactive visualization of large volumetric data with multi-user support. We characterize the problem of scheduling rendering jobs on heterogeneous systems and effectively reduce the problem space, leading to a scheduling design with a much lower complexity. Our test results show that our method can schedule a large amount of jobs at interactive framerates while achieving an optimal solution for data locality and keeping the I/O overhead to a minimum. Our method obtains a well-balanced workload among the processors, giving us a highly scalable solution.

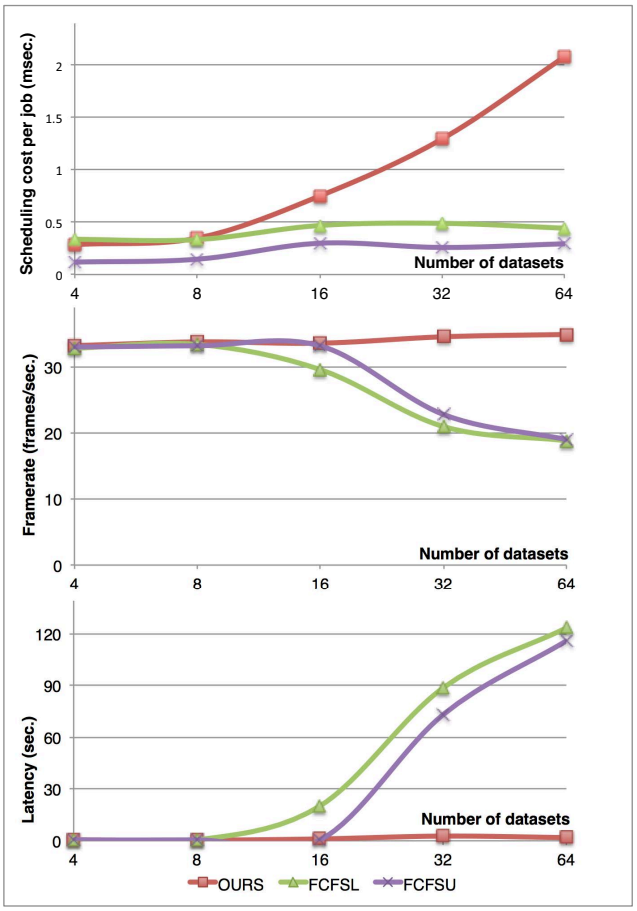


Figure 9. The comparison of the number of datasets in use to the scheduling costs and performance. As shown in the top graph, due to the pre-processing for categorizing incoming jobs by the associated data chunks, the scheduling cost of our method increases as more datasets are in use; however, this cost is negligible because it is typically two to three orders of magnitude smaller than the rendering time. In addition, *OURS* can maintain a stable interactive framerate (the middle graph) and low latency (the bottom graph) even when total data exceed the system memory capacity. The test is run on 16 nodes on the GPU cluster at ANL, and uses the datasets (8GB per dataset) with mixed interactive and batch jobs.

While this paper presents our study of job scheduling specific for the target application of parallel volume rendering, the job scheduling requirements and challenges are representative. Therefore, our design principle and scheduling scheme can be generally applicable to a wider class of interactive applications on large heterogeneous systems.

We have shown that with a set of simple heuristics, our scheduling design helps not only reduce scheduling cost but also achieve good data locality. Apart from the dominating I/O cost, we would also like to consider the associated computation and communication cost of parallel jobs in our future designs. Moreover, we will develop methods to minimize the data transfer between main memory and video memory and to fully exploit high levels of concurrency powered by heterogeneous systems.

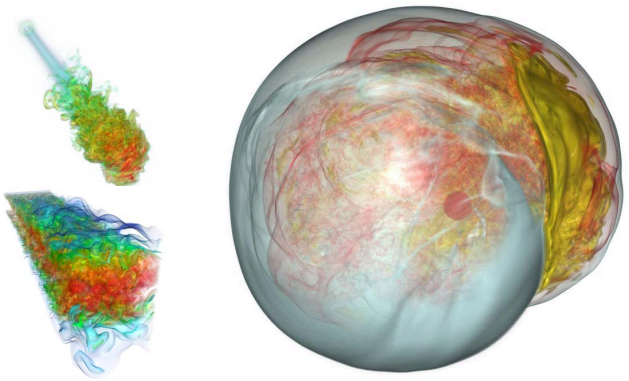


Figure 10. Three example images generated by our parallel visualization system. Top Left: rendering of a plume simulation dataset (data dimension: $252 \times 252 \times 1024$), Bottom Left: rendering of a combustion simulation dataset (data dimension: $2025 \times 1600 \times 400$), Right: rendering of a supernova simultion dataset (data dimension: $864 \times 864 \times 864$).

ACKNOWLEDGMENT

This work has been sponsored in part by the U.S. Department of Energy through grants DE-FC02-06ER25777, DE-FC02-10ER26039, and DE-FC02-12ER26072, program manager Lucy Nowell. This work has also been supported in part by the U.S. National Science Foundation through grants OCI-0749217, CCF-0811422, OCI-0749227, OCI-0850566, OCI-0950008, and CCF-0938114. Computational resources have been made available on Eureka at the Argonne National Laboratory.

REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *the ACM Communications*, vol. 53, pp. 51–58, 2010.

[2] A. Fox, "Cloud computing - what's in it for me as a scientist?" *Science*, vol. 331, pp. 406–407, 2011.

[3] High performance computing using Amazon EC2. [Online]. Available: http://aws.amazon.com/ec2/hpc-applications/

[4] Berkeley lab contributes expertise to new Amazon web services offering. [Online]. Available: http://www.lbl.gov/cs/Archive/news071310.html

[5] M. Levoy, "Display of surfaces from volume data," *IEEE Computer Graphics and Applications*, vol. 8, pp. 29–37, 1988.

[6] J. Kruger and R. Westermann, "Acceleration techniques for GPU-based volume rendering," in *IEEE Visualization*, 2003, pp. 287–292.

[7] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A sorting classification of parallel rendering," in *ACM SIGGRAPH ASIA 2008 courses*, 2008, pp. 1–11.

[8] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh, "A data distributed, parallel algorithm for ray-traced volume rendering," in *Proceedings of the ACM symposium on Parallel Rendering*, 1993, pp. 15–22.

[9] T.-Y. Lee, C. S. Raghavendra, and J. B. Nicholas, "Image composition schemes for sort-last polygon rendering on 2d mesh multicomputers," *IEEE Transactions on Visualization and Computer Graphics*, vol. 2, pp. 202–217, 1996.

[10] D.-L. Yang, J.-C. Yu, and Y.-C. Chung, "Efficient compositing methods for the sort-last-sparse parallel volume rendering system on distributed memory multicomputers," *Journal of Supercomputing*, vol. 18, pp. 201–220, 2001.

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

[12] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh, "Parallel volume rendering using binary-swap compositing," in *IEEE Computer Graphics and Applications*, vol. 14, no. 4, 1994, pp. 59–67.

[13] H. Yu, C. Wang, and K.-L. Ma, "Massively Parallel Volume Rendering Using 2-3 Swap Image Compositing," in *Proceedings of SC*, 2008.

[14] K.-L. Ma and D. M. Camp, "High performance visualization of time-varying volume data over a wide-area network status," in *Proceedings of SC*, 2000.

[15] L. A. Freitag and R. M. Loy, "Comparison of remote visualization strategies for interactive exploration of large data sets," in *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, 2001, pp. 181–183.

[16] T. Toyama, Y. Yamada, and K. Konishi, "A resource management system for volume visualization in desktop grid environments," in *Systems Modeling and Simulation*. Springer Japan, 2007, pp. 314–318.

[17] J. Gao, J. Huang, C. Johnson, and S. Atchley, "Distributed data management for large volume visualization," in *IEEE Visualization*, 2005, pp. 183–189.

[18] J. D. Ullman, "NP-complete scheduling problems," *Journal of Computer and Systen Sciences*, vol. 10, pp. 384–393, 1975.

[19] S. Kim and J. Browne, "A general approach to mapping of parallel computation upon multiprocessor architectures," in *Proceedings of ICPP*, 1988, pp. 1–8.

[20] A. Radulescu and A. van Gemund, "Low-cost task scheduling for distributed-memory machines," *IEEE Trans. on Parallel and Distributed Systems*, vol. 13, no. 6, pp. 648–658, 2002.

[21] D. Perkovic and P. Keleher, "Randomization, speculation, and adaptation in batch schedulers," in *Proceedings of SC*, 2000.

[22] S. Kim and J. Weissman, "A genetic algorithm based approach for scheduling decomposable data grid applications," in *Proceedings of ICPP*, 2004, pp. 406–413.

[23] Y.-C. Chow and W. H. Kohler, "Models for dynamic load balancing in homogeneous multiple processor systems," *IEEE Transactions on Computers*, vol. 36, pp. 667–679, 1982.

[24] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *Transactions on Software Engineering*, vol. 12, pp. 662–675, 1986.

[25] M. Drozdowski, *Scheduling for Parallel Processing*. Springer, 2009.

[26] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the European Conference on Computer Systems*, 2010, pp. 265–278.

[27] M. Strengert, M. Magallón, D. Weiskopf, S. Guthe, and T. Ertl, "Hierarchical visualization and compression of large volume datasets using GPU clusters," in *Proceedings of EGPGV*, 2004, pp. 1–7.

[28] C. Wang, J. Gao, and H.-W. Shen, "Parallel multiresolution volume rendering of large data sets with error-guided load balancing," in *Proceedings of EGPGV*, 2004, pp. 23–30.

[29] S. Marchesin, C. Mongenet, and J.-M. Dischler, "Dynamic load balancing for parallel volume rendering," in *Proceedings of EGPGV*, 2006, pp. 51–58.

[30] X. Cavin, C. Mion, and A. Filbois, "COTS cluster-based sort-last rendering: Performance evaluation and pipelined implementation," in *IEEE Visualization*, 2005, pp. 111–118.