Timeline Scheduling for Out-of-Core Ray Batching

Myungbae Son
KAIST
nedsociety@kaist.ac.kr

Sung-Eui Yoon
KAIST
sungeui@kaist.edu

Figure 1: This figure shows some of the sample views of our tested scenes, *Boeing777* and *SponzaMuseum* consisting of up to 500 M triangles. In handling multiple view requests, our renderer robustly allocates heterogeneous computing resources to reduce the idle time, achieving high horizontal scaling.

ABSTRACT

We present a timeline based scheduling method for Monte Carlo ray tracing of out-of-core models on distributed memory clusters. We abstract different setups of various compute and memory devices into a graph-based representation, and estimate the time for job execution and data transfer in a simple timing model. Our scheduler allocates not only jobs to processors, but also data transfers to memory channels. This approach allows us to control the I/O overload, which is the principal bottleneck in rendering massive-scale scenes. To manage dependencies of data transfers and data intensive jobs, each job and data transfer is arranged on the timeline with dependency relations. Based on this model, our scheduler aims to increase data locality by allocating a job that takes the least time to fetch required data on a given compute device. This goal is achieved by optimizing the data transfer path to maximize latency hiding effects. We have implemented a path tracer on our framework and tested massive models up to 500 M triangles. Compared to prior state-of-the-art scheduling techniques, our renderer achieved higher horizontal scalability on flexible device configurations.

CCS CONCEPTS

•**Computing methodologies** →**Ray tracing;** *Distributed algorithms;*

KEYWORDS

path tracing, out-of-core, cluster, hybrid

ACM Reference format:
Myungbae Son and Sung-Eui Yoon. 2017. Timeline Scheduling for Out-of-Core Ray Batching. In *Proceedings of HPG '17, Los Angeles, CA, USA, July 28-30, 2017,* 10 pages.
DOI: 10.1145/3105762.3105784

1 INTRODUCTION

Monte Carlo ray tracing techniques have been widely adopted for generating high-quality photorealistic rendering results. In its application, the ever-increasing demands on high-quality results, and thus on increasing model complexity, has brought a serious challenge upon achieving high performance in a scalable way [Yoon et al. 2008]. Especially, it has been known to be a hard problem to realize high performance on Monte Carlo ray tracers for out-of-core models which cannot be stored in main memory [Moon et al. 2010].

For Monte Carlo ray tracing techniques that deal with out-of-core data, the bandwidth of memory hierarchy has been a common bottleneck, due to their inherently random data access patterns over the scene data. In this case, such low data locality often causes expensive I/O transfers between different memory devices. Improving locality of out-of-core rendering at scheduling level has been well studied [Silva et al. 2002]. Along with it, various scheduling methods have been studied for supporting a diverse set of device configurations and heterogeneous computing resources [Berman et al. 1996; Kim et al. 2013].

Nonetheless, it has remained as a challenging problem to design robust scheduling techniques that achieve a high throughput of Monte Carlo ray tracers. Specifically, it has been understudied to consider both locality and utilization in complex device configurations, which can occur in modern cloud computing environments and remote rendering systems. Considering both factors at the same time is challenging when the rays are incoherent, and the overhead of data fetching is considerable. Besides, the scheduler must operate without a complete set of ray paths at any moment, making it even hard to schedule the jobs optimally.

Main contributions. In this paper, we address the problem of achieving the high performance, i.e., the smallest makespan, the end-to-end running time of processing all the tasks, of performing Monte Carlo ray tracing in a diverse set of device configurations. For accomplishing the objective, our method manages the schedule consisting of data fetches and the actual rendering job executions. Our timeline-based approach considers data locality in a given device configuration while resolving dependency of jobs. Our contributions are summarized as follows:

- We formulate our problem specification to consist of a device connectivity graph (DCG) and a timing model that describes the time of executing a job and transferring a data block from one memory device to another. Our formulation is general enough to consider data locality, different I/O bandwidths, and data dependency.
- We present a simple, iterative algorithm, Greedy Makespan Balancing (GMB) algorithm, to schedule and distribute jobs from the initial workload. Our GMB algorithm attempts to pull out maximum utilization while seeking the opportunity of hiding the data transfer latency as much as possible.
- We apply our scheduling approach to a distributed out-of-core path tracer, to show the robustness of our framework. It optimizes the schedule on a device configuration consisting of eight heterogeneous nodes, and shows better horizontal scalability than the state-of-the-art schedulers, up to about 40 % higher throughput of ray processing.

2 RELATED WORK

Given more than three decades of research for MC ray tracing, many scheduling algorithms have been proposed for both domains of out-of-core rendering [Silva et al. 2002] and distributed rendering [Chalmers and Reinhard 1998].

At a high level, any techniques from both fields focus on achieving at least one of two scheduling goals: improving utilization and data locality. Unfortunately, both issues are well known to be in a trade-off relation in distributed memory rendering systems [Salmon and Goldsmith 1988], as reducing the data fetch overhead narrows the flexibility of job distribution to exploit load balance, and vice versa. Such effect summarizes that achieving near-optimal throughput remains a challenging problem to the date.

2.1 Distributed-memory ray tracing

The study of distributed-memory ray tracing aims at realizing parallelization over the distributed memory systems. In such systems, most techniques focus on achieving high utilizations of homogeneous devices (e.g., same CPU cores), while maintaining data locality reasonably well. These methods can be classified as image space and domain space decomposition techniques.

The *image space decomposition* strategy lets each process to take a set of initial samples in the image space and to handle all the subsequent rays originated from those samples [DeMarle et al. 2005, 2004, 2003; Keller et al. 2017; Parker et al. 1998; Wald et al. 2003b,a, 2004, 2001]. This approach benefits from natural load balancing and shows a high ray coherence on primary rays, but the potential incoherence of secondary rays limits the data locality.

On the contrary, the *domain decomposition* strategy [Childs et al. 2006; Howison et al. 2012, 2010; Kobayashi et al. 1988; Reinhard et al. 1999; Reinhard and Jansen 1997] allocates a domain, which is a set of scene data, e.g., a part of a scene space subdivision, to each process. Each ray is paired with the domain to be intersected, and if a ray enters a domain, the process who owns that domain takes ownership over the ray to process. This method takes an advantage of data locality over the image space decomposition strategy by minimizing the number of loads, but suffers from distribution issues such as load imbalance and high communication overhead.

Reinhard et al. [1999; 1997] proposed hybrid approaches utilizing components of the two strategies above. This hybrid approach allows a process to be the process generating samples or the process holding the data domain, by considering various information such as ray types. Large-scale volume rendering techniques [Childs et al. 2006; Howison et al. 2012] have been proposed along this line, mainly for primary rays. Navrátil et al. [2014] inspected benefits of various scheduling methods including conservative hybrid techniques in homogeneous clusters . Their analysis has provided insight on which scheduling techniques are better for specific applications. Our framework incorporates such preference in our timing and bandwidth models.

While our method also belongs to a class of hybrid approaches, our method optimizes a scheduling function considering data locality, network bandwidth, and utilization, by using an iterative scheduling algorithm.

2.2 Out-of-core rendering

Out-of-core rendering [Silva et al. 2002] is another field of study on scheduling techniques for rendering out-of-core data such as large-scale geometries or shadings [Eisenacher et al. 2013]. Unfortunately, the random access generated by MC rendering on such data can cause enormous I/O overheads. As the bottleneck is located on the fetch cost between the memory hierarchies, scheduling for such jobs has been studied to maximize the data availability, leading to improving the overall rendering performance.

A family of *ray batching* techniques for Monte Carlo rendering techniques has been successfully established with explicitly controlling the data locality over the memory hierarchy [Budge et al. 2009; Moon et al. 2010; Pharr et al. 1997; Steinhurst et al. 2005]. Pharr et al. [1997] presented a decomposition of workloads such as ray intersections into smaller ones depending on a unit data block and reordered them using the cost-benefit function. Budge et al. [2009] applied this technique to CPU/GPU hybrid workstation with simple priority-based heuristics. Moon et al. [2010] proposed a similar approach with rays ordered by the space-filling curve based on approximate hit points. Our framework smoothly integrates the ray batching technique with global scheduling and distribution to reduce total makespan over the entire distributed memory system.

2.3 Specification techniques

Ray batching has some similarities to the domain decomposition strategy described in distributed memory techniques, in a sense that rays are sorted to the data domain and processed. There are, however, differences between them. For example, distributed memory renderers allocate data blocks to computing resources and move jobs between them to match the jobs and required data. On the other hand, the out-of-core renderer dynamically fetches the data block from a memory device to process the jobs.

To generalize strategies mentioned above, different characteristics of them must be specified within a problem instance. Specifically, a device configuration and work dependency must be encoded, and each of them has been separately studied in different fields: distributed heterogeneous schedulers [Berman et al. 1996; Kim et al. 2013; Potts 1985; Shchepin and Vakhania 2005] and data-parallel execution frameworks [Isard et al. 2007; Larsen et al. 2016; Loques et al. 1998], respectively.

Berman et al. [1996] and Kim et al. [2013] schedule unstructured jobs in heterogeneous device configurations by formulating the transfer and execution time, and optimize the schedule with linear programming to minimize the makespan accordingly. Their algorithm plans a semi-optimal schedule with a setup cost of issuing jobs, but the data availability is not considered, and its scheduling cost becomes intractable in the case of multiple devices. Data-parallel frameworks such as P-RIO, Dryad, and Legion [Bauer et al. 2012; Isard et al. 2007; Loques et al. 1998] can specify dependent parallel jobs in a graph-based manner, and schedule them to maximize the throughput. However, the communication channels are either homogeneous or predefined, which is not enough to determine the best data transfer path between the memory hierarchy.

Unfortunately, most aforementioned prior methods do not consider our problem of global illumination, in a sense that only one of these two concepts of device configurations and job dependency are focused. We propose a general formulation considering both of them for addressing our problem.

3 PROBLEM SPECIFICATION

In this section, we describe our approach to the scheduling problem. We argue that out-of-core rendering techniques for Monte Carlo ray tracing (e.g., path tracing) have characteristics of dynamic/dependent tasks in Sec. 3.1. We then formalize a specification for such applications, using a *device connectivity graph (DCG)* and the execution timing model of jobs in Sec. 3.2. A combination of them forms a problem instance for our scheduler to work on. We explain our scheduling method to solve the problem in Sec. 4.

3.1 Characteristics of Monte Carlo Ray Tracing

In terms of parallelism, the job structure of Monte Carlo ray tracing has two interesting performance characteristics that a robust scheduler must consider: the incoherence among generated rays, and the dynamic and dependent nature of ray generation. While recent studies have focused on the nature of ray incoherency, we focus on the dynamic nature of the task.

In the stochastic process of Monte Carlo ray tracing methods, new rays (e.g., secondary rays) can be spawned from tracing other rays. As a result, neither the distribution of future jobs (e.g., processing rays) nor their required data can be deterministically known at a time that we run our scheduling module. Therefore, it limits the exploration of the scheduling space until dependent rays are processed, forming *job-job dependencies*. This characteristic is known as *dynamic tasks* in the field of scheduling, which poses challenges to achieving a schedule with high quality.

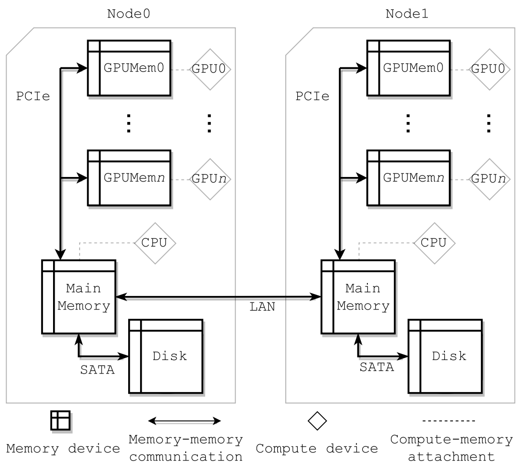


Figure 2: An illustration of a device connectivity graph, representing a multi-GPU cluster with two nodes.

Such dependency relation is further expanded in out-of-core rendering. Before the jobs are executed, the required data must be fetched to the main memory, which forms *data-job dependencies*. As the cost of data fetches is dominant in out-of-core rendering, these dependencies need to be handled by the scheduler as well.

To consider both kinds of dependencies, the timing when each job or data fetch is done must be analyzed and encoded into the schedule. In next section, we present various tools to address them.

3.2 Device Connectivity Graph (DCG) and Timing Model

Given a set of tasks, the goal of our scheduler is to create a schedule that minimizes the makespan. In our work, the scheduling result is to assign jobs to available computing resources, while considering data locality and data dependency among jobs for achieving the highest throughput. In the case of out-of-core rendering, the schedule consists of data transfer between memory devices and the actual job execution in compute devices, once the required data has been gathered and transferred to the attached memory device.

The communication cost between memory devices (e.g., disk and network I/O costs) for massive models usually dominates the running time of the overall rendering process. It is thus important to consider such connectivity constraints and timing behavior into the scheduler. To consider such connectivity, we propose a device connectivity graph with timing models. Based on these models, we expect the running time of tasks in compute devices.

Device Connectivity Graph (DCG).. We propose to use a novel device connectivity graph to estimate the expensive data transfer costs better. This graph describes the connectivity of memory devices and thus communication time behavior between them (Fig. 2). Each vertex of the graph is a memory device, and each edge between two vertices represents a communication channel between memory devices. Each compute device is associated with a memory device that is directly accessible from the compute device.

The scheduler controls which data should be present at which memory device, to ensure that the attached compute device can properly access the data on its job execution. A required data can be transferred along a path from a source to a destination in multiple hops. For example, in a case where the central scene database is physically separated from worker nodes, the scene data can be transferred along a network path where multiple nodes participate.

Our device connectivity graph is inspired by a network graph. Nonetheless, this concept was not used in prior out-of-core scheduling methods. Furthermore, the network connectivity graph considers only the node-level communication between nodes (e.g., the LAN connection between two nodes in Fig. 2), not the communication level of memory devices (e.g., PCIe, SATA, and LAN in Fig. 2). Instead, we consider different types of memory devices and allow data transfers through any paths between two memory devices (e.g., from the disk to GPU memory), resulting in higher throughput (Sec. 4.4).

Timing model. To minimize the actual makespan, a performance model is required to define a timing relationship between jobs and compute devices. Specifically, we want to estimate the execution time of jobs and the transfer time of required data. The transfer time depends on the latency and bandwidth of a communication channel, and the size of data. The execution time of a job in a compute device depends on the type of the job, the size of input data, and the compute device that processes it. Note that a job may take multiple input data – e.g., ray intersection job requires both scene data and ray data to be processed.

Ideally, precise modeling techniques are preferred for producing high-quality schedules, but there are too many factors and high overheads to obtain such models. To minimize the scheduling overhead, we formulate a simple linear timing model, while avoiding taking complex, hardware-specific internal parameters into consideration. Our timing model defines the processing time and transfer overhead, as a linear function of the size of workload. More specifically, the transfer overhead, $T_{TRANS}(d_i \rightarrow d_j,\, w)$, of moving a data block w from a memory device d_i to another memory device d_j is defined as the following:

$$T_{TRANS}(d_i \rightarrow d_j,\, w) = T_{LAT}(d_i \rightarrow d_j) + \frac{|w|}{T_{BW}(d_i \rightarrow d_j)}, \quad (1)$$

where $T_{LAT}(d_i \rightarrow d_j)$ and $T_{BW}(d_i \rightarrow d_j)$ represent two parameters of the communication channel, the setup cost of the channel, i.e., the network latency, and the bandwidth, respectively.

Similarly, the processing time, $T_{EXEC}(d, j, W)$, of a job of type J_j taking a set of input data blocks $W =< w_1, w_2, \ldots >$ on a compute device d is defined as the following:

$$T_{EXEC}(d, j, W) = \begin{cases} 0, & if\ W = \emptyset \\ T_{SETUP}(d, j) \\ \quad + T_{RATE}(d, j) & , \quad otherwise \\ \quad \cdot (|w_1|, |w_2|, \ldots) \end{cases} \quad (2)$$

where $T_{SETUP}(d, j)$ is the constant setup cost (e.g., GPU kernel launch overhead) for launching a job of type J_j in the d compute

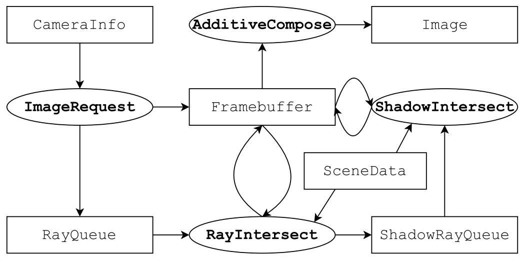


Figure 3: An illustration of dependencies for the path tracer. The circular shapes with the bold font are jobs, while the rectangular ones are associated data.

device, and $T_{RATE}(d, j)$ represents the weight vector describing how each input data $w_1, w_2, \ldots$ contributes to the execution time based on its size.

This model assumes that jobs with a payload n take $O(n)$. For the transfer overhead, this is straightforward in most network connections. For processing time, most ray intersection and sampling jobs scale linearly over the number of rays to be processed, as long as the complexity of each scene subdivision is uniformly distributed. For a more sophisticated processing time model for varying complexity, see [Larsen et al. 2016]. To verify our timing model empirically, we have measured the execution time for each job with varying input sizes, and found that observed execution times align well with our linear formulation; see the plot of the measured execution times available in the sup. report.

Our formulation for the timing model is similar to the linear approximation that Kim et al. used [2013] in proximity query jobs, but separates the data transfer term from the processing term. This separation enables our scheduler to compute the data transfer overhead separately from the cost of job processing itself. This property is critical for out-of-core ray batching techniques, which need to consider the data availability on a compute device.

4 GREEDY MAKESPAN BALANCING (GMB) SCHEDULER

Given the specification of DCG and timing model, we are ready to discuss our scheduling method, greedy makespan balancing (GMB) algorithm. Before we explain our method, we first explain how to handle jobs with dependency.

4.1 Handling Dependency

The scheduler must meet the constraint of various dependencies between job execution and data fetching. For example, a job can only be executed after all the required data associated with the job are fetched first to the corresponding memory device. Another example is in-between dependencies of jobs themselves (e.g., accumulating the image, generating rays, and intersection tests).

Our scheduler first takes a dependency graph of jobs to know dependencies between jobs. Fig. 3 shows an example of a dependency graph of jobs to perform path tracing. To simplify the process of handling the dependencies of jobs and data fetching within our scheduler, we treat a data transfer as a separate job, run by a channel between memory devices, i.e. an edge in DCG. As a result,

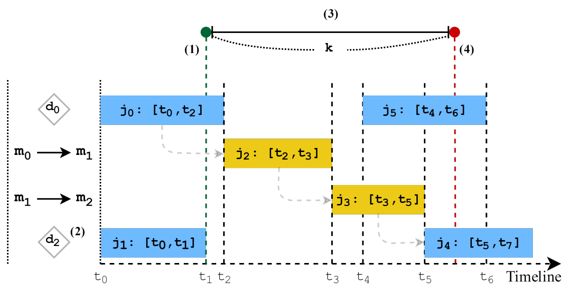


Figure 4: This figure shows an overview of our timeline based scheduling. In this example, the scheduler maintains four timelines of compute devices (d_0 and d_1) and channels ($m_0 \rightarrow m_1$ and $m_1 \rightarrow m_2$). The schedule has four E-jobs (blue intervals, j_0, j_1, j_4 and j_5) and two T-jobs (orange intervals, j_2 and j_3). Each job holds interval as well as dependency relation to other jobs. For example, an E-job, j_4, executed by the compute device d_2, requires data from another compute device d_0, which also performs a prior E-job j_0. The scheduler considers the dependency and computes a scheduling result by adding T-jobs, j_2 and j_3, through the memory channels between the corresponding memory devices of d_0 and d_2.

we design our scheduler to allocate jobs to both compute devices and memory channels, where the former takes executable jobs, henceforth E-jobs, and the latter takes data transfer jobs, henceforth T-jobs. This approach allows us to manage jobs and data fetch canonically, simplifying the design and implementation; additional justifications of our choice is available at the sup. report.

Scheduling dependent jobs in a timeline. Given dependency constraints, our scheduler generates series of E-jobs and T-jobs for each device or channel. For that, our scheduler generates a schedule, i.e., scheduling result, on the timelines. Each device or channel maintains its own timeline, and we estimate an interval of a job. We also record a dependency of execution orders for jobs. The main goal of our scheduler is then reformulated to compute the timelines for compute devices and memory channels. The execution of a schedule is done by following the timeline in order, while waiting for dependent jobs to be finished. An example of a schedule timeline is illustrated in Fig. 4.

4.2 Creating a Schedule

Given a specification consisting of DCG and timing models, obtaining an optimal schedule is NP-hard. Specifically, the problem of minimizing the makespan for a simple unrelated parallel system and its ρ-approximation (where $\rho < \frac{3}{2}$) are known to be NP-hard [Graham et al. 1979; Lenstra et al. 1990]. This problem can be reduced from ours by assuming the complete connection between devices and the independence of jobs. We thus strive to find an approximate, but efficient scheduling approach for our problem.

Our scheduling method is based on the wisdom of previous approaches. A common approach of many ray batching techniques is to schedule blocks, whose job granularity is high and fetch cost is low, aiming to reduce both setup and fetch time [Pharr et al. 1997]. In the field of distributed rendering, many systems prioritize on load balancing, while trying to hide latency, effectively reducing the fetch time and increasing the portion of job processing of the final makespan [Wald et al. 2004].

Specifically, we aim to minimize makespan by reducing the idle time of compute devices, which are composed of the following parts in the decreasing priority:

(1) **True idle time**. Idling of a device can be caused by many different reasons. If a compute device is not scheduled nor waiting for a data, we define it to be the true idle state. Load balancing reduces the time for this state.
(2) **Fetching time**. When a compute device is allocated for an E-job whose required data is unavailable at the time, it must wait for required data to arrive, i.e. waiting for T-jobs on input data to be completed. An example of reducing such time is latency hiding based on a prefetching.
(3) **Setup time**. A device has started an E-job, and is expected to use the constant setup time corresponding to the term $T_{SETUP}(d, j)$ in Eq. 2. This cost can be minimized by increasing the granularity of work.

We now explain the strategy to reduce the idle time consisting of the aforementioned three parts. Whenever a new job is created, we push it to a global job queue. We run our scheduler based on the timeline concept, and invoke our scheduler when one of the compute devices exhausts every job in its timeline and enters the idle state. Alternatively, a dedicated scheduler process may run in one of the compute devices and manage the timeline, to hide the scheduling latency at higher synchronization cost.

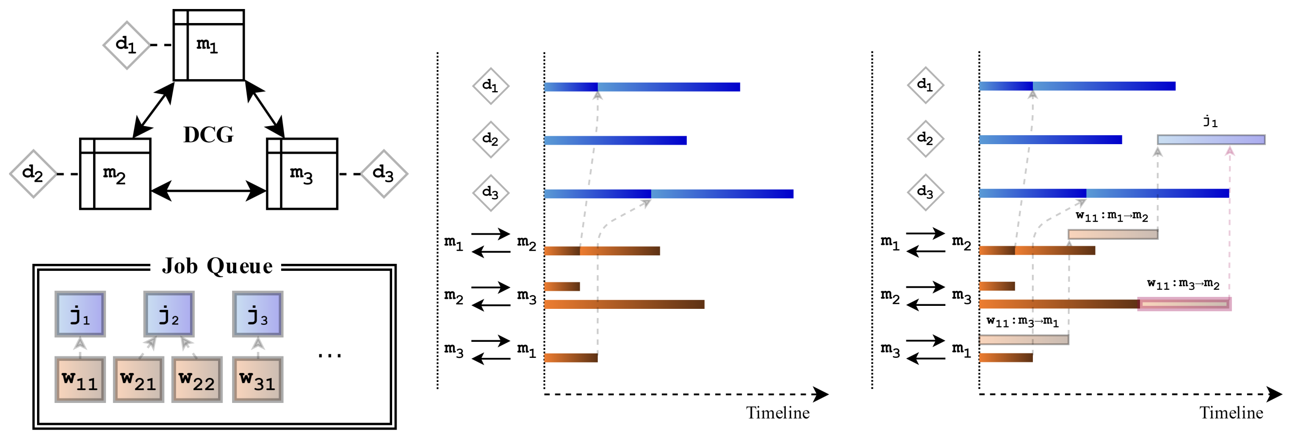
Given this context, our scheduler finds a compute device d that has the minimal execution time, and then assigns an E-job j retrieved from the global job queue to the device d. Choosing the E-job j is described in Sec. 4.3. When the scheduler assigns the E-job to the device, the scheduler also assigns its dependent T-jobs to channels before the E-job. The assigning process of dependent T-jobs is described in Sec. 4.4.

This scheduling algorithm iterates until all available jobs are exhausted or each compute device has been scheduled to proceed to a certain execution time threshold k (e.g., 2000 ms). The reason why we schedule devices up to the execution time threshold is that there could be many newly generated jobs due to the dynamic nature of our application, and it is thus beneficial for the scheduler to reschedule with those newly generated jobs for achieving the better scheduling quality.

As an example in Fig. 4, assume that the schedule initially had two E-jobs, j_0 and j_1. **(1)** When one of the compute devices enters the true idle state (in the example, d_2 enters the idle state first at t_1), **(2)** it executes the scheduling module. **(3)** The scheduling module iteratively allocates the job onto the schedule of all compute devices, up to the duration of execution time threshold k (that is, to the point of $t_1 + k$). **(4)** The resulting schedule is broadcasted and synchronized among the compute devices and channels. The pseudocodes of the scheduling module are enlisted in the sup. report.

4.3 Job Selection

Job selection is performed once a compute device d is chosen. In this context, an E-job j is selected based on its smallest fetching



(a) DCG and job queue **(b) Current schedule** **(c) Adding j_1 to the schedule**

Figure 5: This figure illustrates an example of adding a job onto the current schedule for a cluster consisting of three connected pairs of compute and memory devices. The scheduler tries to assign a job, j_1, to d_2, which is the least occupied compute device in the current schedule as illustrated in (b). To evaluate the fetch overhead of dependent data, w_{11}, of j_1, the scheduler performs analysis on which transfer path is the most efficient for each job to reduce the fetch time. Suppose that the data w_{11} is located in m_3. In (c), the scheduler considers two choices: 1) sending the data in the direct path of $m_3 \rightarrow m_2$, and 2) taking an indirect path of $m_3 \rightarrow m_1 \rightarrow m_2$. It turns out that the indirect path shows a faster fetch time over the direct path.

overhead; i.e. a job that can be started earliest is chosen at the moment (Fig. 5). If multiple jobs have the same fetching overhead, the scheduler chooses the job with the smallest setup time over the total execution time, i.e. $\frac{T_{SETUP}(d, j)}{T_{EXEC}(d, j, W)}$, as a tie-breaker. In other words, we prefer a job with higher granularity, i.e. the load of work.

The main technical challenge for the job selection is to estimate fetching overheads of E-jobs located in the global job queue. To estimate such fetching overheads at a compute device d, it is critical to compute possible data transfer paths to load the dependent T-jobs into the memory device associated with the compute device d. This is explained in the subsequent section.

Possible starvation and workarounds. A job is considered *fetch-free* if it has zero fetching overhead. If there are fetch-free jobs, the one with the highest granularity is chosen. Note that the fetching overhead is measured from the moment of adding a job in d. If all dependency chain of T-jobs for j can be resolved before d enters an idle state in the current timeline, then j is considered fetch-free, avoiding starvation if it has the highest granularity.

If the execution time threshold k is short enough, then current scheduling window of k may get filled by fetch-free E-jobs before any dependent T-jobs can be resolved for other E-jobs. In such case, those E-jobs may get forced to wait indefinitely until fetch-free jobs are exhausted, leading to the starvation. In our implementation, we have avoided the problem by setting the high value for execution time threshold k, long enough to ensure any E-job has a chance to become fetch-free within the scheduling window of k. Alternatively, we may introduce the concept of aging for each job, if the application has a more strict constraint on its responsiveness.

4.4 Finding a Fetch Path

Before launching an E-job j on a compute device d, its associated memory device, m_d, must have all required data for the job j. For each input data block w of the job j, we check if it is available in m_d. For this process, the scheduler maintains which data blocks are in which memory device at the point when each timeline of related channels is finished.

If all input data blocks exist in m_d, the fetch cost for the E-job j is zero and it can be scheduled at the end of the timeline of d. Otherwise, our scheduler attempts to schedule T-jobs to fetch missing data from other devices to m_d by constructing a transfer path between them, before the E-job j is scheduled.

Construction of a transfer path. In a simple case where the data is available from a source at the beginning and the schedule is empty, the problem of finding a transfer path is simplified into finding the shortest path in DCG, where the weight of each edge, i.e. channel, represents a transfer time (T_{TRANS}) of the requested data block. As it is preferable to fetch the data as soon as the channels are ready, generated T-jobs can be directly chained into each other as illustrated in Fig. 4, i.e., j_2 and j_3.

Based on this intuition, we propose *Smallest Datapath Contention (SDC)* algorithm based on Dijkstra's algorithm [1959], which also considers other jobs that are already scheduled in channels. Given a DCG, a current schedule snapshot p, a requested data block w, and a target compute device m_d, our SDC algorithm finds a transfer path within DCG that fetches w into m_d at the earliest time.

Considering available time of data. Given a schedule p, the time when data block w is available to each source memory device may differ. We thus handle it by adding a virtual source vertex S onto the DCG. We add an edge between S and each of memory device d that has w. The weight of each new edge is set to the time when a device is ready for sending the data, i.e. the time when the memory device has held or received w, and is ready to send the data. Once we introduce the virtual source vertex S, the problem transforms into obtaining the transfer path from S to m_d.

Availability of channels. On each step of searching for candidates to expand the path, Dijkstra's algorithm evaluates the accumulated path length from a vertex p to a candidate vertex q by simply adding the edge cost, w_{pq}, to the shortest path. Such accumulation process corresponds to the case where it is possible to send the data block from p to q as soon as the data in p is available.

For our case, however, the channel, c_{pq}, between two memory devices p and q should be available, to make the transfer through c_{pq}. If the channel c_{pq} is busy, the transfer must be delayed to the point when the channel is available. As a result, the shortest cost toward p can be superseded by the waiting time of c_{pq}, i.e., $FinishTime(c_{pq})$. The final cost update equation to evaluate the extended path length, $l_{S \rightsquigarrow q}$, is thus defined as the follows:

$$l_{S \rightsquigarrow q} = max(l_{S \rightsquigarrow p}, FinishTime(c_{pq})) + w_{pq}. \quad (3)$$

A pseudocode of finding a fetch path is available in the sup. report.

5 GMB SPECIALIZED FOR PATH TRACING

In this section, we describe how to apply our GMB scheduler to accelerate the path tracer, one of well-known Monte Carlo ray tracing methods. We choose path tracing for the demonstrative purpose and its simplicity, but we believe that many other ray tracing methods such as bidirectional path tracing [Lafortune and Willems 1993] and photon mapping [Hachisuka et al. 2008]) can be supported in a similar manner.

5.1 Jobs of Path Tracing

Fig. 3 illustrates processing jobs with related input and output data of path tracing in the data flow diagram format. Our decomposition of path tracing tasks is similar to that of Budge et al. [2009] in a sense that each job can be efficiently run also in SIMD compute devices such as GPUs, with some differences that enable a higher level of parallelization. Especially, the method of Budge et al. lets each cluster node handle a part of jobs defined by image plane decomposition and then schedules those local jobs, i.e., rays generated from a part of the image plane over each node. On the other hand, our job formulation supports parallelization of all those jobs over different nodes by distributing and scheduling jobs globally. Specifically, our scheduler strives to reduce the makespan by transferring jobs and their dependent data from a node to another through memory channels. It is a novel feature that prior methods did not consider. We also add common job structures of remote rendering such as image requesting and additive composition of framebuffers, which is detailed in the sup. report.

In the perspective of the GMB scheduler, each data block is independent and can be transferred as described in Sec. 4.4, with the following specifics: 1) `Framebuffers` are only allowed to be moved for **AdditiveCompose**, and 2) if a `(Shadow)RayQueue` is transferred to a memory device that already has a queue with the same type and associated `SceneData`, their rays are merged to the recipient queue. For detailed management technique of `(Shadow)RayQueues`, see [Laine et al. 2013].

5.2 Dynamic Job Prediction

As we have discussed in Sec. 3.1, the unpredictable nature of Monte Carlo rendering casts many schedulers including ours to work on

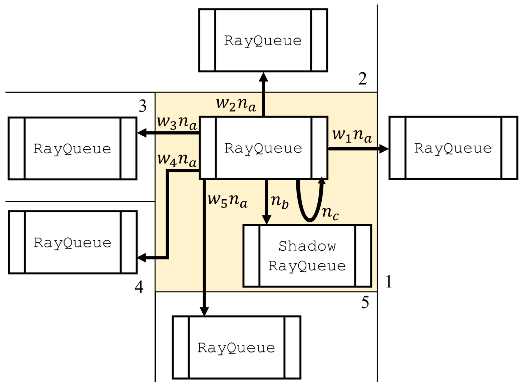


Figure 6: This figure shows an example of future job predictions generated from processing the `RayQueue` of the center cell shown in yellow. For rays located in `RayQueue` of the center cell, we predict how many rays should be appended to $\texttt{RayQueue}_i$ for each adjacent cell i. The distribution weight w_i is determined by the relative ratio of the surface area from the current cell to that of the cell i. As predicted results of ray hits, n_b shadow rays and n_c secondary rays are predicted to be generated and appended to corresponding ray queues.

a snapshot of jobs at a particular time. If future events that are likely to happen can be estimated well, we can consider such jobs in addition with the current snapshot. This in turn realizes better-populated data blocks and higher performance.

To achieve this goal, we predict those future jobs by inspecting data blocks associated with jobs that are currently under processing. We then infer which data block can be accessed in the near future and prefetch them at the scheduling stage for higher performance. In particular, during the scheduling phase, we anticipate jobs of **RayIntersect** and **ShadowIntersect** that might be generated in future. We focus on these two jobs as they are dominant and the most I/O intensive jobs. We then attempt to schedule them along the rest of jobs that were already requested, allowing the scheduler to prefetch the dependent data and process those future jobs.

For utilizing future events in the scheduler, we need to predict job types and their amount, i.e., the number of rays per ray queue associated with each data block. As a result, jobs of **(Shadow)-RayIntersect** are expected in the following three cases (Fig. 6):

No intersection. A ray may propagate into one of the neighboring `SceneDatas`, e.g., an adjacent cell in the uniform grid, if it does not intersect with geometry contained in the current one. For this, we first estimate the hit ratio, α, of a ray that is updated per `SceneData`. This value is maintained by averaging the hit ratio over prior intersection results on the corresponding `SceneData`. This is a reasonable assumption for temporally coherent scenes, as the ray distribution on such scenes tends not to change drastically. n_a, the number of those rays that do not intersect is then estimated as $n_a = (1 - \alpha)|R|$, where $|R|$ is the number of rays in the current ray queue. Since these rays are assumed to be distributed uniformly, we assign these rays uniformly to those neighboring cells to the current cell. Note that we predict future jobs without accessing individual ray information that would incur a high computational

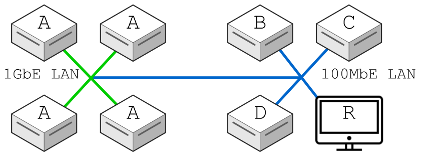


Figure 7: An illustration of node connection setup for our experimental hardware setup. Nodes of type A, B, C, D are GPU workstations, while node R is a commodity desktop. Nodes with type A are interconnected with 1GbE LAN (green line), while others are connected with 100MbE LAN (blue line).

overhead. Our chosen strategy is simple yet conservative because it tends to anticipate the worst case scenario, where the rays are equally scattered to all those neighboring cells, which is usually the most time-consuming case.

Shadow ray generation. When the direct illumination sampling is adopted, an intersection of a non-shadow ray guarantees to generate a new shadow ray. The number of these shadow rays, n_b, is expected by $n_b = \alpha|R|$. These shadow rays are expected to be processed with the current `SceneData`, as its origin lies on the hit point of the parent ray. We thus assign them to the ray queue of the current cell.

Secondary ray generation. An intersection of a non-shadow ray can create a non-shadow ray, when the path is survived, i.e. the ray passes the Russian roulette test. Those survived rays are simply determined by the expectation with current ray properties (e.g., attenuation). We maintain the average of the pass possibility, β, over the current `RayQueue`, so that we can predict how much rays would survive the Russian roulette process on average. The number of such secondary rays, n_c, is then expected as $n_c = \alpha\beta|R|$, and its job is assigned to the current cell.

Incorporating predicted jobs into the scheduler. When our scheduler is initiated as described in Sec. 4, it also predicts future jobs, $j_{i1}, \ldots, j_{im}$, for each job j_i in the input job queue, as mentioned above. We let each predicted job j_{ik} to have a link to its preceding job j_i. This prediction process can be applied recursively to even predicted jobs. We thus also introduce the maximum depth of prediction as a user parameter. At the end of executing a job of **(Shadow)RayIntersect**, we inspect the output data block (e.g., the size of RayQueue) to determine whether its future jobs are predicted correctly or not. When the prediction turns out to be unable to execute, e.g., an expected E-job is not generated, we discard the predicted job and its whole dependent jobs such as its T-jobs.

Overall, our prediction has up to 75 % accuracy on average with the tested benchmarks, albeit its simplicity. Thanks to the high prediction accuracy, we have observed up to 85 % overall throughput improvement. More detailed analysis is available in the sup. report.

6 EXPERIMENTAL RESULTS

We have implemented our framework and tested its performance over a cluster of GPU workstations as illustrated in Fig. 7 along with its memory channel bandwidths. The testing system is composed of eight nodes, consisting of four homogeneous nodes (type A: i7-4770, 8-core, Titan), three heterogeneous workstation nodes (type B: i7-4790, 8-core, Titan / type C: E5-2690, 16-core, Titan / type D: E5-2690, 16-core, TitanX) and one commodity desktop node (type R: i7-3770, 4-core, GTX980). Each type of node has one SATA storage, 8GB main memory and different type of CPU and GPU. Type A nodes are interconnected with 1GbE LAN, while others are connected with 100MbE LAN. In total, heterogeneous 8-node consists of 84 compute devices and 24 memory devices. The exact specification for each node is noted in sup. report.

	Boeing777	***SponzaMuseum***
Num. of triangles	496 M	245 M
Model mem. footprints	26.5 GiB	12.3 GiB
Num. of data blocks	129	65
Image spec.	512^2, 4*spp*	2048^2, 128*spp*

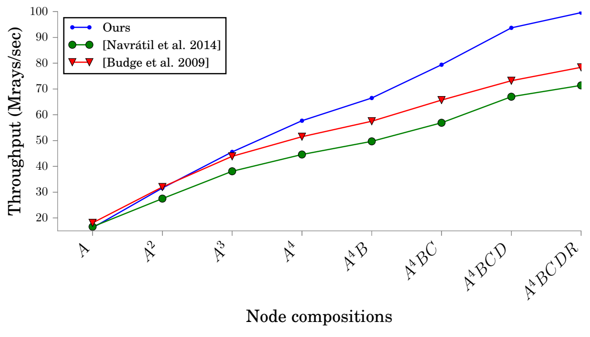
Table 1: This table shows various statistics of each tested scene. The memory footprint includes input scene data and acceleration data structure for Monte Carlo ray tracing. ***spp*** **represents Sample Per Pixel.**

For comparative study, we have implemented two prior methods from Budge et al. [2009] and Navrátil et al. [2014] on the same hardware configuration. The first method [Budge et al. 2009] distributes jobs in the image-space decomposition to each node and performs case-by-case analysis for allocating jobs to CPU and GPU. The second method [Navrátil et al. 2014] is a dynamic ray scheduling framework for distributed memory parallel systems. On implementing [Navrátil et al. 2014], we choose *LoadAnyOnce* policy as it exhibits high performance on global illumination scenes. Originally, this method runs on homogeneous CPU clusters. We have thus modified it to run on GPU by allocating additional worker processes that share GPU ownership on round-robin for each step.

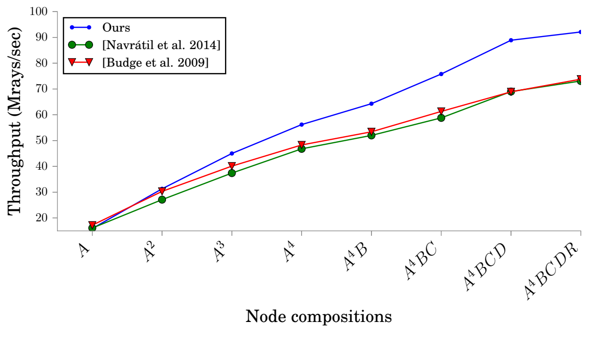
For the implementation of ray intersection kernels in each method, we use modified kernels from an optimized rendering system [Kim et al. 2014] for CPU, and from NVIDIA OptiX for GPU. To obtain the timing model for E-jobs, a sample run has been performed to measure individual job durations. Then, the architecture-specific model constants – $T_{SETUP}(d, j)$ and $T_{RATE}(d, j)$ – are fitted using the least square method.

As the execution time threshold, k, grows, we have higher chance to utilize the latency hiding capabilities due to wider planning space. On the other hand, increasing k might increase the scheduling overhead and prediction error. Nonetheless, our inspection on a range of different k values does not show a high variation on the performance in a range of $[200ms, 10000ms]$ in the tested scenes. As a result, we simply set this value $k = 2000ms$ in our experiment.

Benchmarks Scenes. We have tested two different scenes rendered by path tracing (Fig. 1, Table 1). The *Boeing777* scene of a highly complex CAD model consists up to about 500 million triangles. The *SponzaMuseum* scene is a virtual museum scene consisting of Crytek Sponza, St. Matthew, two Lucy, and two David models featuring scanned models (up to about 250 million triangles) with a high amount of diffuse reflections. A detailed per-bounce performance analysis of incoherent rays on this scene is provided in the sup. report. Each model is subdivided in the stackless kd-tree with



(a) Boeing777



(b) SponzaMuseum

Figure 8: This figure shows the throughput of each method as more computational resources are added to the system. X-axis represents the composition of nodes, where Y-axis represents the throughput in the total number of rays shot per second.

rope [Popov et al. 2007] for efficient traversal in both CPU and GPU. A median-split strategy is used to subdivide the model into approximately even-sized data blocks.

A number of client processes run on a separate node, generating image requests. For testing different methods in a uniform setting, we set each client to request a scene along predefined camera movements, feeding an image request in a frame-by-frame manner. To see the steady-state behavior of different methods, we increase the number of clients until each method reaches its peak performance, and then compare their peak performances.

6.1 Scalability Analysis on Peak Performance

We have evaluated the horizontal scalability by measuring throughputs of different methods on varying number of nodes. The peak throughput is measured with the total number of rays shot per unit time. Fig. 8 shows measured peak throughputs.

In specific, we increasingly add the nodes as follows. We first attach the homogeneous node *A* one-by-one to see how the system scales as the computational resources scales linearly. Then we add node *B*, *C*, and *D* in increasingly powerful performance to see if the scheduler captures increasing performance boosts. Finally, we add the commodity desktop node *R* to the system to represent the scenario of remote rendering where the client also participates. Each node has a full copy of the model dataset on its HDD.

As the number of nodes increases, the tested method shows higher throughputs. However, the performance per each node degrades as various overheads such as communication cost and scheduling overhead increases for all three methods. Nonetheless, our method scales better horizontally over the prior methods as our method utilizes available channels better, thanks to locality-aware job distribution, and optimizes the data transfer path and prefetching via such available channels. For example, the average utilization for LAN bandwidth on heterogeneous 8-node setup for [Budge et al. 2009], [Navrátil et al. 2014], and ours are 37.1%, 39.2% and 87.8%, respectively.

For a single node case (*A*), our method shows slightly lower performance compared to others. In this case, there are not much divergence on paths as the bandwidth of the SATA channel is the dominant bottleneck here, leaving not much space to optimize the data transfer path or to plan to prefetch ahead of time. As a result, this case exhibits only the scheduling overhead (up to about 20 % of the total runtime) of our method to search for data transfer paths.

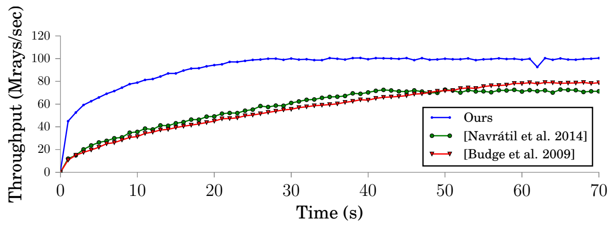


Figure 9: This figure shows how throughputs of different methods increase as time advances on the central scene database scenario for *Boeing777* scene on heterogeneous 8-node setup. Required data is fetched and cached to each node on-demand. Our method reaches to its peak performance earlier than other tested methods.

Note that the throughput achieved by our method with eight nodes is about 100 M rays per second. This throughput may be interpreted low compared to throughputs of recent ray tracers. However, our reported throughput is achieved with massive models that cannot be stored in main memory. A recent heterogeneous rendering method utilizing CPU/GPU with approximated geometry for the Boeing model reported about 10 M rays per second in a single node, and its full version using the original geometry reported lower than 1 M rays per second [Kim et al. 2014].

In addition to our cluster setup, we have also tested a case of rendering using a single workstation with multi-GPUs, but without using network data transfers. While our method is not mainly designed for this case, our method shows a minor improvement, i.e., up to 8% overall performance improvement over [Budge et al. 2009], which is mainly designed for this case, and 26% improvement over [Navrátil et al. 2014]. The detailed results and analysis are given in the sup. report.

6.2 Efficiency of Data Fetching

To verify the efficiency of data fetching of different methods, we have measured how fast they can handle for a scenario using the central scene database. In this scenario, only one node (*master*) has a full copy of the scene data in its HDD, and others (*slaves*) fetch necessary data via network, and cache into its local HDD. Specifically, we have measured how fast each method reaches to its peak performance. In all experiments, the master node is set to be the node of type *D*.

As shown in Fig. 9, our method reaches to the peak performance earlier than previous methods, thanks to the optimization of transfer paths. In prior methods, all slaves directly request data from the master, and thus suffer from the LAN bottleneck on the outgoing link of the master node. On the other hand, our method is able to determine the availability of each data block on each memory device, and constructs a transfer path from a slave to another slave node, offloading the data fetches from the master node. This enables our approach to converge to the peak performance much faster than previous methods. In the first 10 seconds, our system utilizes 31% of total bandwidth for slave-to-slave channels for *SceneData* transfer, while other methods do not.

7 CONCLUSION AND FUTURE WORK

We have proposed a novel scheduling method for ray tracing out-of-core data. Our approach aims to minimize the makespan by considering data locality and expensive I/O bandwidths for various device configurations, which are encoded in DCG. Our main contribution is the iterative scheduling method, greedy makespan balancing method that considers different data transfer paths to minimize the makespan. To demonstrate benefits of our method, we have applied our method to path tracing of two massive models.

Interesting future research directions lie ahead. The complex shading systems for large-scale textures and volumetric scattering can utilize our approach with different formulation of job schemes. For example, the job structure can encode the dependencies of deferred rendering structures [Eisenacher et al. 2013]. We might also explore larger scheduling space (e.g., considering multiple steps on job allocation) for achieving better performance.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their detailed and constructive comments and suggestions.

This work was supported by ICT R&D program of MSIP/IITP. [R0126-17-1108, Proximity computing and its applications to autonomous vehicle, image search, and 3D printing]

REFERENCES

Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *SC'16*. 66.

Fran Berman, Rich Wolski, Silvia Figueira, Jennifer Schopf, and Gary Shao. 1996. Application-level scheduling on distributed heterogeneous networks. In *Supercomputing, ACM/IEEE*. 39–39.

Brian Budge, Tony Bernardin, Jeff A. Stuart, Shubhabrata Sengupta, Kenneth I. Joy, and John D. Owens. 2009. Out-of-core data management for path tracing on hybrid resources. *Computer Graphics Forum (EG)* 28, 2 (2009), 385–396.

Alan Chalmers and Erik Reinhard. 1998. Parallel and distributed photo-realistic rendering. *Course notes for SIGGRAPH* (1998), 425–432.

Hank Childs, Mark A Duchaineau, and Kwan-Liu Ma. 2006. A scalable, hybrid scheme for volume rendering massive data sets. In *EGPGV*. 153–161.

David E. DeMarle, Christiaan P. Gribble, Solomon Boulos, and Steven G. Parker. 2005. Memory sharing for interactive ray tracing on clusters. *Parallel Comput.* 31, 2 (2005), 221–242.

David E. DeMarle, Christiaan P. Gribble, and Steven G. Parker. 2004. Memory-savvy distributed interactive ray tracing. In *EGPGV*. 93–100.

David E. DeMarle, Steven Parker, Mark Hartner, Christiaan Gribble, and Charles Hansen. 2003. Distributed interactive ray tracing for large volume visualization. In *PVG*. IEEE Computer Society, 12.

Edsger W Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.

Christian Eisenacher, Gregory Nichols, Andrew Selle, and Brent Burley. 2013. Sorted deferred shading for production path tracing. In *Computer Graphics Forum*, Vol. 32. 125–132.

Ronald L Graham, Eugene L Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. 1979. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics* 5 (1979), 287–326.

Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. 2008. Progressive photon mapping. *ACM Transactions on Graphics (TOG)* 27, 5 (2008), 130.

Mark Howison, E Bethel, and Hank Childs. 2012. Hybrid parallelism for volume rendering on large-, multi-, and many-core systems. *IEEE Transactions on Visualization and Computer Graphics* 18, 1 (2012), 17–29.

M. Howison, E. W. Bethel, and H. Childs. 2010. MPI-hybrid parallelism for volume rendering on large, multi-core systems. In *EGPGV*. 1–10.

Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, Vol. 41. 59–72.

Alexander Keller, Carsten Wächter, Matthias Raab, Daniel Seibert, Dietger van Antwerpen, Johann Korndörfer, and Lutz Kettner. 2017. The Iray Light Transport Simulation and Rendering System. *CoRR* abs/1705.01263 (2017). http://arxiv.org/abs/1705.01263

Duksu Kim, Jinkyu Lee, Junghwan Lee, Insik Shin, John Kim, and Sung-Eui Yoon. 2013. Scheduling in heterogeneous computing environments for proximity queries. *IEEE Trans. on Visualization and Comput. Graph.* 19, 9 (2013), 1513–1525.

Tae-Joon Kim, Xin Sun, and Sung-Eui Yoon. 2014. T-ReX: interactive global illumination of massive models on heterogeneous computing resources. *IEEE Transactions on Visualization and Computer Graphics* 20, 3 (2014), 481–494.

Hiroaki Kobayashi, Satoshi Nishimura, Hideyuki Kubota, Tadao Nakamura, and Yoshiharu Shigei. 1988. Load balancing strategies for a parallel ray-tracing system based on constant subdivision. *The Visual Computer* 4, 4 (1988), 197–209.

Eric P Lafortune and Yves D Willems. 1993. Bi-directional path tracing. In *Proc. of CompuGraphics*, Vol. 93. 145–153.

Samuli Laine, Tero Karras, and Timo Aila. 2013. Megakernels considered harmful: wavefront path tracing on GPUs. In *Proc. of HPG'13*. 137–143.

Matthew Larsen, Cyrus Harrison, James Kress, David Pugmire, Jeremy S Meredith, and Hank Childs. 2016. Performance modeling of in situ rendering. In *SC'16*. 24.

Jan Karel Lenstra, David B Shmoys, and Éva Tardos. 1990. Approximation algorithms for scheduling unrelated parallel machines. *Math. Program.* 46, 1-3 (1990), 259–271.

Orlando Loques, Julius Leite, and Enrique V Carrera. 1998. P-RIO: a modular parallel-programming environment. *Concurrency, IEEE* 6, 1 (1998), 47–57.

Bochang Moon, Yongyoung Byun, Tae-Joon Kim, Pio Claudio, Hye-Sun Kim, Yun-Ji Ban, Seung Woo Nam, and Sung-Eui Yoon. 2010. Cache-oblivious ray reordering. *ACM Transactions on Graphics (TOG)* 29, 3 (2010), 1–10.

Paul Navrátil, Hank Childs, Donald S Fussell, Chong Lin, and others. 2014. Exploring the spectrum of dynamic scheduling algorithms for scalable distributed-memory ray tracing. *IEEE Trans. on Vis. and Comput. Graph.* 20, 6 (2014), 893–906.

Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan. 1998. Interactive ray tracing for isosurface rendering. In *Proc. of Vis'98*. 233–238.

M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. 1997. Rendering complex scenes with memory-coherent ray tracing. In *ACM SIGGRAPH*. 101–108.

Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. 2007. Stackless KD-tree traversal for high performance GPU ray tracing. In *Computer Graphics Forum*, Vol. 26. 415–424.

C. N. Potts. 1985. Analysis of a linear programming heuristic for scheduling unrelated parallel machines. *Discrete Applied Mathematics* 10, 2 (1985), 155–164.

Erik Reinhard, Alan Chalmers, and Frederik W Jansen. 1999. Hybrid scheduling for parallel rendering using coherent ray tasks. In *PVG*. 21–28.

Erik Reinhard and Frederik W Jansen. 1997. Rendering large scenes using parallel ray tracing. *Parallel Comput.* 23, 7 (1997), 873–885.

J. Salmon and J. Goldsmith. 1988. A hypercube ray-tracer. In *Proc. of Hypercube Concurrent Computer and Applications*. 1194–1206.

Evgeny V. Shchepin and Nodari Vakhania. 2005. An optimal rounding gives a better approximation for scheduling unrelated machines. *Operations Research Letters* 33 (2005), 127–133.

C. Silva, Y.-J. Chiang, W. Correa, J. El-Sana, and P. Lindstrom. 2002. Out-of-core algorithms for scientific visualization and computer graphics. In *Vis'02 Courses*.

Joshua Steinhurst, Greg Coombe, and Anselmo Lastra. 2005. Reordering for cache conscious photon mapping. In *Graphics Interface*. 97–104.

Ingo Wald, Carsten Benthin, Andreas Dietrich, and Philipp Slusallek. 2003b. Interactive ray tracing on commodity pc clusters. In *Euro-Par 2003*. Springer, 499–508.

Ingo Wald, Carsten Benthin, and Philipp Slusallek. 2003a. Distributed interactive ray tracing of dynamic scenes. In *PVG*. 77–85.

Ingo Wald, Andreas Dietrich, and Philipp Slusallek. 2004. An interactive out-of-Core rendering framework for visualizing massively complex models. In *EGSR*. 82–91.

Ingo Wald, Philipp Slusallek, and Carsten Benthin. 2001. Interactive distributed ray tracing of highly complex models. In *Rendering Techniques*. 277–288.

S.-E. Yoon, Enrico Gobbetti, David Kasik, and Dinesh Manocha. 2008. *Real-Time Massive Model Rendering*. Morgan & Claypool Publisher.