To appear in an IEEE VGTC sponsored conference (LDAV 2018).

SpRay: Speculative Ray Scheduling for Large Data Visualization

Hyungman Park*‡ Donald Fussell† Paul Navrátil‡

*Electrical and Computer Engineering †Computer Science ‡Texas Advanced Computing Center
The University of Texas at Austin

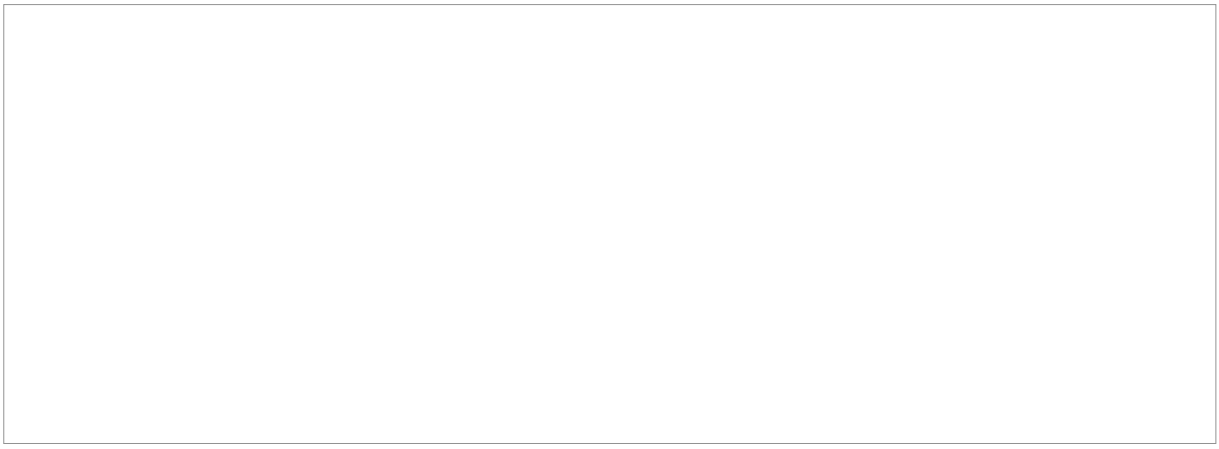


Figure 1: All images rendered with our speculative ray tracing technique. First three columns: a massive channel-flow turbulence DNS dataset. Last two columns: an RM fluid instability dataset and an Enzo Astrophysics AMR dataset (left and right). The number of triangles from left to right: DNS2 (1.8 billion), DNS1-side and DNS1-back (0.9 billion), RM (108 million), and Enzo (8 million). Five images on the bottom row show ambient occlusion shading, and the rest show three-bounce path tracing. For all datasets, 32 samples per pixel were used to render images at 1024×1024 resolution, and one diffuse ray and 16 shadow rays were generated at every hit point. Using Stampede2 Skylake at the Texas Advanced Computing Center, each node with 192 GB memory, at least eight nodes are required to render the DNS dataset with speculation enabled.

Abstract

With modern supercomputers offering petascale compute capability, scientific simulations are now producing terascale data. For comprehensive understanding of such large data, ray tracing is becoming increasingly important for 3D-rendering in visualization due to its inherent ability to convey physically realistic visual information to the user. Implementing efficient parallel ray tracing systems on supercomputers while maximizing locality and parallelism is challenging because of the overhead incurred by ray communication across the cluster of compute nodes and data loading from storage. To address the problem, reordering rendering computations by means of ray batching and scheduling has been proposed to temporarily avoid inherent dependencies in the rendering computations and amortize the cost of expensive data moving operations over ray batches. In this paper, we introduce a novel speculative ray scheduling method that builds upon this insight but radically changes the approach to resolving dependencies by allowing redundant computations to a certain extent. To evaluate the method, we measure the performance of different implementations for both out-of-core and in situ rendering setups. Results show that compared to a well-known scheduling method, our approach on ambient occlusion and path tracing achieves up to 2.3× speedup for the scenes comprising up to billions of triangles extracted from terascale scientific data.

Index Terms: Human-Centered Computing—Visualization—Visualization Techniques; Computing Methodologies—Computer Graphics—Rendering—Ray Tracing

*e-mail: hyungman@utexas.edu

1 Introduction

As data size increases, traditional post-processing methods for visualization, where the data is communicated through storage, are becoming impractical owing to limited I/O bandwidth. Alternative methods such as in situ and co-processing visualization that bridge the data between simulation and visualization either through memory or interconnect networks have shown a promising path forward. Simultaneously, ray tracing is becoming increasingly important for 3D-rendering in visualization because of its inherent ability to produce physically-based images. Hence, the goal of this paper is to develop an efficient parallel ray tracing system and algorithms capable of rendering high-fidelity images for such enormous data on supercomputers, while facilitating the aforementioned visualization methods.

For rendering such complex scenes at scale, the data is subdivided into a grid of domains such that each domain can fit into memory and domains are assigned to distinct nodes of the cluster for parallel computations. As such, ray tracing in this context requires managing the rays and data in each subdivided region, which essentially makes it a ray scheduling problem. Specifically, prior to performing rendering computations, either the relevant data needs to be loaded into where the rays are, or the rays need to be moved to where the data is in the cluster. With this constraint, exploiting the massively parallel compute resources on supercomputers while achieving locality (i.e., reducing data I/O, both between nodes and between a node and the filesystem) is not only an important problem but also one of the key challenges in improving performance.

In order to achieve this goal, a wide spectrum of methods have been studied on many different computing platforms. One set of methods reorder rendering computations by means of ray batching and scheduling. This reordering technique temporarily avoids inherent dependencies in the rendering computations and amortizes

expensive data moving operations across ray batches. However, according to our preliminary studies, frequent reordering may incur costly I/O operations. Moreover, because reordering does not completely avoid rendering dependencies, they may further hinder parallelism, compromising overall performance.

This paper addresses such problems by introducing a new ray scheduling system and algorithms capable of rendering terascale data on supercomputers. Our system, unlike previous ones, leverages trade-offs between work efficiency, locality, and parallelism. Our system promotes an amortization of expensive data loading costs and parallel ray processing while allowing redundancy more in a controlled way and compromising communication time. This is specifically achieved by aggressive ray work creation via speculations of visibility queries. The end result is two variants of the speculative ray scheduling system, each specialized for out-of-core and in situ rendering. Our implementation, compared to a well-known method, gains up to 2.3× speedup, enabled by an increased level of parallelism as well as a significant amortization of expensive data moving operations.

2 Related Work

When rendering large scenes, the data, such as geometry, materials, and textures, may not all fit into memory. Such rendering jobs can be done using a cluster of compute nodes with the scene data stored either in memory or out-of-core, depending on the cluster size.

As the scene becomes complex, efficiently managing millions of rays traveling across different domains of the scene, along with the local and remote data, easily becomes a non-trivial task. To overcome the overhead incurred by file I/O and the forwarding of different kinds of intermediate data, it is useful to frame as a scheduling problem rather than a rendering problem. The following sections provide an overview of such perspectives for both out-of-core and distributed-memory ray tracers.

2.1 Out-of-Core Ray Tracing

Out-of-core ray tracing typically seeks to amortize expensive I/O costs over many rays in-flight, which essentially means batching rays to improve data locality. Rays are batched together based on criteria distinct from one method to another, but the common goal is to maximize the spatial coherence of ray batches.

Pharr et al. [19] designed an out-of-core ray tracing system capable of rendering large complex scenes by reordering the rendering computation on the ray batch in a domain, based on cost and benefit values associated with each domain. Budge et al. [5] implemented a path tracer on a CPU/GPU hybrid system, which separates its entirety into three distinct layers, each responsible for scheduling, data management, and rendering computations. Moon et al. [14] applied ray reordering to their path tracing and photon mapping system but unlike others, their approach is cache-oblivious. They newly defined a hit point heuristic which allows the spatially close rays to be scheduled together. Eisenacher et al. [9] reduced loading costs by deferring the shading of all hit points. Their method is highly optimized for texture mapping in their production-level path tracer.

Like the aforementioned approaches, our method attempts to improve coherence by utilizing ray batching and scheduling. However, our method maximizes the amortization of expensive I/O operations even further by addressing the problem of inherent sequential dependencies in the variants of Pharr's rendering system [19]. Our approach defers hit points similarly to Eisenacher's [9], but we cast speculative rays where they do not. Therefore, our system aggregates many more rays before resolving all hit points, potentially leading to a better amortization of costly I/O.

Another important distinction of our method is our scheduling strategy. We use a ray distribution-based front-to-back scheduling in which ray batches are scheduled in such a way that file I/O as well as the processing of redundant rays is minimized. To this end, our system collects the statistics of individual rays and evaluates metrics to determine the order in which ray batches are scheduled.

2.2 Distributed-Memory Ray Tracing

In general, both software and hardware rendering systems have two different spaces to work on: screen space and data space. In the seminal paper written by Molnar et al. [13], depending on how rendering work is sorted and distributed to processing elements between these two spaces, rendering can broadly be classified as sort-first, sort-middle, and sort-last. In order to distribute subsets of work to individual nodes, sort-first and sort-middle decompose screen space, whereas sort-last decomposes data space.

The concept of this sorting classification has equally been applied to distributed-memory ray tracing. In screen-space decomposition, each node takes a tile of the entire image plane and traces the camera rays independently. As the rays bounce around in the scene, they may touch different regions of the scene. Thus, the assignee node must, for exactness, read associated data from either memory or storage before processing them, depending on how large the scene is. Majority of distributed-memory ray tracing systems to date have followed this approach [2, 4, 7, 8, 16, 17, 22]. Our method uses this sort-first screen-space decomposition for out-of-core rendering.

In data-space decomposition, each node is assigned one or more domains, and it processes the rays in a domain until they either terminate inside the domain or travel across the bounds of the domain without intersecting any of the primitives in the domain. In this case, the node holding the rays must send them to the data responsible for the next domain that they enter. Many ray tracers, including volume renderers, have employed this approach in order to fit large datasets into aggregate memory of the cluster as well as to utilize abundant parallel compute resources available [6, 10, 16, 18].

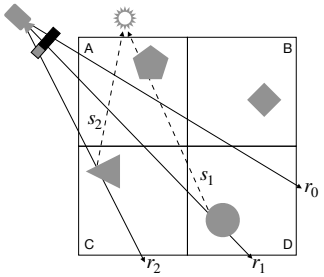
In addition, hybrid approaches combining these two decomposition methods have been studied [3,5,11,20,21]. Our method follows this approach where subsets of screen space are initially distributed across the cluster and newly spawned rays are forwarded to owner nodes based on data-space decomposition.

The latest distributed-memory ray tracer is the one by Son and Yoon [21]. Their system, targeted for a hybrid CPU/GPU cluster, employs a time line-based scheduling, which uses a dependency graph and a timing model to reduce the makespan (i.e., the running time of processing all the tasks) of rendering and data sending.

Additionally, in the context of this paper, it is worth mentioning the Kilauea ray tracer [11, 12] because our method shares similar insight in terms of generating redundant rays. Both Kilauea and our system leverage speculation as a means to exploit parallel compute resources. However, Kilauea's main drawback is that it associates a state with a ray and maintains it in the owner node responsible for the ray. Thus, the owner has to wait for a ray's test result to arrive from one or more nodes, which not only incurs a round trip latency, compromising overall performance, but also requires additional ray communication between nodes for redundantly spawned secondary rays. By contrast, each node in our system maintains a set of local speculation buffers, and the cluster collectively resolves physically true hit points by compositing them in logarithmic time. Moreover, Kilauea, unlike our method, does not provide ray scheduling via speculation for out-of-core rendering.

3 Problems in Baseline Methods

In this section, we review the concept of ray batching and scheduling based on prior work, and we identify the problems using a simple rendering example. Specifically, we study the work of Pharr et al. [19] for out-of-core ray tracing and the work of Navrátil et al. [15, 16] for distributed-memory ray tracing. Pharr et al. established a groundwork for ray scheduling and the concept has been well applied to both in-core and out-of-core ray tracers to date. Navrátil



Out-of-Core								In Situ					
Domain	t_0	t_1	t_2	t_3	t_4	t_5	t_6	Domain	t_0	t_1	t_2	t_3	t_4
A	(r_0,r_1,r_2)		(s_2)				(s_1)	A	r_0,r_1,r_2		s_2		s_1
B		r_0	r_0	(r_0)				B		r_0			
C		(r_1,r_2)				(s_1)		C		r_1,r_2		s_1	
D			r_1	r_1	(r_0,r_1)			D			r_0,r_1		
Load-Process	A	C	A	B	D	C	A	Process	A	B, C	A, D	C	A
								Send	r_0,r_1,r_2	r_0,r_1,s_2	s_1	s_1	
Color			s_2					Color			s_2		

Figure 2: An example scene of four domains, each with a geometric object (left). A table showing the evolution of each domain and the operations required to render the scene in two distinct rendering scenarios (right). For the out-of-core scenario, we assume a single node, and the data for each domain is fetched either from in-memory cache or from storage. Ray batches within the parentheses are the ones scheduled at each time step. For the in situ scenario, we assume four nodes, each with the preloaded data of a distinct domain. All four nodes concurrently process ray batches at each time step; thus, parentheses are implicit. Ray batches are forwarded to owner nodes as needed.

```
1  while domains are nonempty
2      choose a domain with most rays
3      for each ray in domain
4          if camera ray
5              do ray-triangle intersection queries
6              if closest intersection found
7                  compute ray's color contribution
8                  spawn shadow rays and place them in current domain
9              else
10                 place ray in next closest hit domain
11         else if shadow ray
12             do ray-triangle occlusion queries
13             if unoccluded
14                 if next closest hit domain exists
15                     place ray in next domain
16                 else
17                     update color with ray's contribution
```

Listing 1: A baseline method for scheduling ray batches in different domains.

extended the work to the realm of distributed-memory ray tracing for large data visualization.

We can abstract both methods as the pseudocode in Listing 1, which assumes processing only camera and shadow rays without spawning secondary rays. The basic idea is to find the closest domain of a ray and defer processing it until the next scheduling step where all other rays placed in the same domain are processed together at once. Such ray batching and deferring essentially amortize the costs for data loading and memory access. To briefly describe the algorithm, the ray batch in a domain is first chosen based on some heuristic (a domain with most rays in this case). Then, the rays are tested against geometry within the domain to determine a closest hit point for a camera ray and an occlusion for a shadow ray. If the ray crosses the domain bounds without any intersection, it is placed in the next closest domain for the next round of scheduling. This iterative process persists until all rays are terminated.

Given the method, we render the scene in Fig. 2 having four geometric shapes in separate domains and a light source and a camera placed next to domain A. Two camera rays, r_1 and r_2, respectively hit on the two shapes of triangle and circle in domains C and D, from which two shadow rays, s_1 and s_2, are subsequently spawned. The table in Fig. 2 considers two rendering scenarios: out-of-core rendering on a compute node and in situ rendering of distributed in-core data on four compute nodes.

For simplicity, we first illustrate the out-of-core scenario by showing how each domain evolves with ray batches using the baseline algorithm, which runs to completion until all rays in the scene are processed. We choose a domain with most rays for the heuristic and for a tie breaker, the priority is given in the order of domains A through D. At t_0, all three camera rays are in domain A and the remaining domains are empty. Thus, the ray batch in domain A is processed and since none of the rays intersect the pentagon, they are moved to next domains (i.e., r_0 to domain B and r_1 and r_2 to domain C). At t_1, domain C is scheduled over domain B based on the heuristic, and in domain C, only r_2 intersects the triangle so a shadow ray s_2 is spawned and moved to domain A without an occlusion. r_2 is terminated as it travels beyond the scene. r_1, however, is moved to domain D without a hit. At t_2, domain A is scheduled again according to the tie breaker. Since s_2 is an unoccluded shadow ray moving out of the scene, it is terminated after the pixel is updated with its color contribution. The scheduling of remaining ray batches repeats similarly as described.

Next, for the in situ scenario, we parallelize the processing of ray batches using four compute nodes. As shown in Fig. 2, the behavior diverges at t_1 since the two ray batches in domains B and C are processed in parallel. The processing of the ray batch in domain C is identical as it is done in the out-of-core scenario. r_0 in domain B, however, is sent to domain D without a hit. At t_2, the processing of ray batches in domains A and D updates the pixel of s_2 with its color and spawns a shadow ray s_1, which is eventually advanced to domain A (via domain C), where it is terminated owing to an occlusion by the pentagon.

For the out-of-core scenario, we assume that all four domains are processed on a single node, assuming only one domain can fit in memory. Whenever a ray batch is scheduled, the domain is read either from in-memory cache or from storage. Thus, the scheduling of same domains for consecutive scheduling steps reduces expensive loading time. However, this example exhibits no such occurrences but incurs the overhead of three additional loads compared to the required four loads, a 75% additional performance impact. For the in situ scenario, each node owns one distinct domain of the four. Therefore, ray batches are sent to domain owners at each time step. Once all rays are sent to owner nodes, ray batches on different nodes are processed in parallel, thus enhancing the ray processing time. One problem here is that as the render progresses, the number of active ray batches shrinks and processors become underutilized. As implied by Fig. 2, the node utilization is 35% on average (i.e., only one or two nodes are active at any given time).

The data loading overhead and the decreased level of parallelism are caused by two factors. First, partitioning the scene into domains compromises the scheduling coherence of ray batches, particularly in the context of this algorithm. That is, a ray that travels across domain boundaries may lead to expensive storage access. A domain already visited may be revisited in a later scheduling step due to newly created shadow and secondary rays moving into it. Second, inherent dependencies of the algorithm hinder the sustainability of parallelism due to the sequential nature of ray-domain traversal. Specifically, a ray-to-ray dependency requires that spawning a new

ray is deferred until after the parent ray's hit point is found. A ray-to-domain dependency requires that a domain is processed only after a ray has entered it. With these two observations, we address the problems in the next section by introducing our speculative ray scheduling approach.

4 Speculative Ray Scheduling

As a way to improve performance, our method exploits data locality and parallelism in the same spirit as prior work. However, the main idea of our approach adds the concept of speculate-first-and-resolve-later to leverage underutilized processing resources when few active rays are traced. Our method thus completely discards dependencies by scheduling rays speculatively at the earliest and deferring resolution of the speculation until the processing of all speculatively generated rays is finished. In the following two sections, we elaborate on this concept using two rendering variants: one tailored for out-of-core and one tailored for in situ.

4.1 Out-of-Core Rendering

For out-of-core rendering, we assume the sort-first screen-space decomposition, which means that each node owns the entire domains of a scene to render a subset of the image plane, which is often called a tile. Therefore, even if multiple nodes are used, ray communication between nodes is unnecessary since each node can independently render the tile. Communication only occurs when the pixel values of each node are composited into the frame buffer of a display node at the end.

As shown in Fig. 3, our rendering system takes camera rays as the input and generates rendered pixels at the end. Each rendering job undertakes four distinct phases: domain intersection, domain scheduling, domain traversal, and visibility resolution. We now elaborate on each of the four phases.

4.1.1 Domain Intersection

Ray-domain intersection tests are performed on incoming camera rays and newly generated secondary rays. As a result, a set of *ray batch queues* are used to speculatively place each ray in all of its hit domains, and statistics are updated as subsequently described.

4.1.2 Domain Scheduling with a Front-to-Back Heuristic

The main role of the domain scheduling is to promote an early rejection of redundant ray-primitive queries performed on the speculative rays that are highly probable of having closer hits or occlusions in the domain other than the current domain scheduled. This is achieved by reordering domains such that the ray batch in each domain is likely to traverse most of the hit domains in front-to-back order. Such rejection tests ultimately eliminate the unnecessary data loading caused by speculatively queued rays.

A schedule, defined as the order in which domains are processed in the subsequent phase of domain traversal, is evaluated. Briefly explaining, as depicted in Fig. 4, we use the ray distribution statistics collected for each domain to evaluate what we call a *traversal score* and sort the scores in descending order to get the schedule.

Specifically, whenever a ray is speculatively queued into a domain, we increment the counter value corresponding to its domain ID and the distance between the ray origin and the hit point on the domain boundary, defined as a *domain hit depth*. Given the ray distributions collected in this way, we evaluate the traversal score of each domain by taking a weighted sum of the ray counts at different domain hit depths and finally sort them to get the order in which ray batches are processed. Here is a procedure in more detail for estimating the order.

1. Initialize an array of $Counter[N][M]$ to zeros. N is the number of domains in the scene and M is the maximum number of hit domains that the array can hold for each domain.

2. For each ray, perform ray-domain intersection tests to get a list of the domains that it intersects, and sort them by distance from the ray origin. Iterating the list, increment the value at $Counter[i][j]$ by one, where i is a domain ID and j is a domain hit depth with zero being the nearest and $d-1$ being the farthest, if d ($\leq M$) domains are hit by the ray.

3. For each domain i, estimate the traversal score by evaluating a weighted sum of the counter values, $Score[i] = \sum_{j=0}^{M-1} w_j Counter[i][j]$, where w_j is a weight used to put emphasis on closer domains. We simply use $w_j = M - j$ to take relatively higher weights on the domains near the majority of ray origins in the scene.

4. Sort the array $Score[0 \ldots N-1]$ in descending order to get a traversal order. The domain at $Score[0]$ is the first one to process.

4.1.3 Domain Traversal

Once a domain schedule is available, we can start processing ray batches in the given order, namely a *ray-domain traversal*. During the traversal, a ray may intersect many surfaces, and each potential intersection increases the number of speculative rays needed. This not only requires more memory space but also requires computing more ray-primitive tests. Worse, if the speculation of a ray is ultimately discarded (e.g., an earlier termination point was found) and the domain data needs to be brought into memory from storage, the redundant data loading may significantly increase overall running time. Therefore, prior to performing such expensive operations, ray batches undertake a filtering step where only the rays that are more likely to be a correct speculation are sorted out for further processing.

Given the ray batch of a scheduled domain, the domain data is loaded either from a domain cache in memory or from storage, depending on the cache status. Then, ray-primitive tests are performed on the ray batch. If the closest hit is found for a radiance ray, speculative secondary rays are spawned according to the surface material. A shadow ray is then inserted into a *commit queue* with an occlusion flag, and a radiance ray is inserted into a *spill queue* only if its ray depth exceeds the maximum speculation ray depth, defined as a *history depth*. In addition, for both types of secondary rays except the radiance rays in the spill queue and occluded shadow rays, ray-domain intersection tests are performed, the rays are speculatively placed in all of their hit domains, and statistics are updated accordingly. As a result, the ray batch queues may or may not be empty at this point.

To manage all the speculations, we allocate a *history buffer* which maintains a history of hit distances (i.e., t values) for every screen-space sample location, and its state is constantly updated as a new hit point is found. In addition, each ray locally maintains its own history of the ancestor rays so it can be compared against the globally managed history buffer for ray filtering or visibility resolution in the subsequent phase. If all t values up to the current ray depth are identical to each other, the speculations are valid. If any of the older t values in the history buffer (i.e., earlier bounces preceding the ray being tested) are less than the associated t values in the local history, the speculations are invalid. If the converse is true, the history buffer is updated with the ray's history.

4.1.4 Visibility Resolution

As one round of domain traversal terminates, all the speculative shadow rays inside a *retire queue* are resolved using the history buffer and the color contributions of physically valid shadow rays are accumulated into the pixel values. Then, the empty retire queue is swapped with the commit queue in double buffering fashion. The reason for the swap is that a domain traversal may not be complete

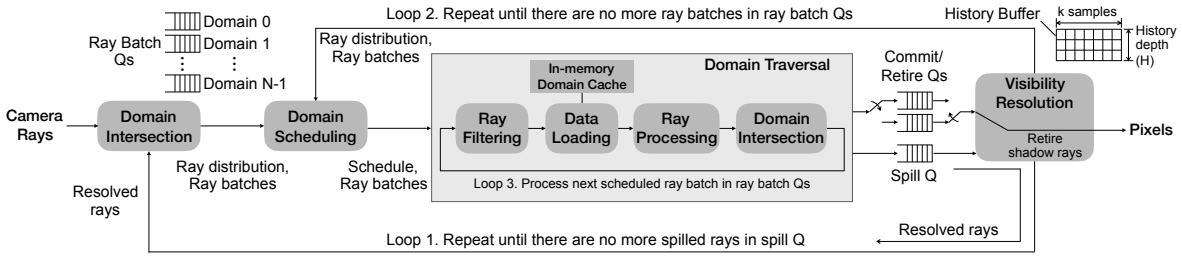


Figure 3: An abstract view of our speculative ray scheduling system for out-of-core rendering. A set of ray batch queues for N domains are used to speculatively place a ray in all the hit domains. A commit queue and a retire queue are used in double buffering fashion. The commit queue is used to insert the speculative shadow rays newly created in the ray processing step. The retire queue is used to resolve all the speculative shadow rays previously committed and retire their pixel values. A history buffer, which can accommodate k screen-space samples with a ray depth of H, is used to maintain a speculation history of the t values of each ray. An in-memory domain cache loads domain data from storage.

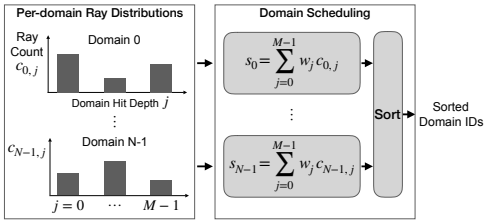


Figure 4: Ray distribution-based front-to-back scheduling for out-of-core rendering. A domain scheduler uses the ray distributions of N domains to estimate the traversal scores and sort them in descending order. A schedule is merely the sorted domain IDs.

for the shadow rays spawned in the middle of the traversal. Thus, only the ones committed in the previous round of traversal are safely taken for retiring.

If any of the ray batch queues and the retire queue are nonempty, another round of domain traversal is initiated with a new schedule (Loop 2 of Fig. 3). Otherwise, the system restarts to process the rays in the spill queue after resolving their visibility using the history buffer (Loop 1 of Fig. 3).

4.1.5 Implementation

Listing 2 shows the main function. During the initialization (Lines 2-3), the ray depth and ray counters are reset, camera rays are speculatively queued into radiance queues, and the corresponding ray counters, each indexed by the domain ID and the domain hit depth, are accordingly incremented. The outermost loop (Lines 4-27) repeats until the spill queue becomes empty. The spill queue is populated with all the radiance rays that overflow the history buffer. They wait there until the inner `while` loop (Lines 11-25) retires shadow rays to update pixel values. The main purpose of the outermost loop is to offer the system an ability to adjust speculation by allowing the while loop to spawn new radiance rays only up to a certain number of ray bounces and limit the size of the history buffer. At the end of the loop, the ray depth is incremented by the history depth.

The `for` loop (Lines 6-9) iterates the spilled rays to resolve their visibility by comparing their local history against the up-to-date history buffer and if visibility is correct, it speculatively places the resolved ones into radiance queues and increments the ray counters. Then, the history buffer is reset before the next round of ray scheduling begins (Line 10).

As long as there is a ray to process, the `while` loop finds a new schedule (Line 12) and feeds it into the `for` loop (Lines 13-22) that performs domain traversal. Once one round of domain traversal for a given schedule is done, it resolves visibility for all the shadow rays in the retire queue and updates the pixel values of the resolved ones (Line 23). It then swaps the commit and retire queues as well as the

```
1  function oocSpray()
2     ray_depth = 0, statistics.reset(), generate camera_rays
3     domainQueuing(camera_rays, statistics /* ray counters */)
4     repeat
5        // *_RQ: radiance queue, *_SQ: shadow queue, hbuf: history buffer
6        for each ray in qs(SPILL_RQ).get()
7           if hbuf.visibility(ray) is success
8              domainQueuing(ray, statistics)
9              ray.reset()
10       hbuf.reset()
11       while any queues of {RQ, SQ_IN, RETIRE_SQ} are nonempty
12          schedule = statistics.schedule()
13          for i = 0 to N-1
14             domain = schedule.get(i)
15             filter(RADIANCE, domain, RDATAQ, RQ)
16             filter(SHADOW, domain, SDATAQ_IN, SQ_IN)
17             filter(SHADOW, domain, SDATAQ_OUT, SQ_OUT)
18             if any filtered queues are nonempty
19                domain.load()
20                processRQ(domain, RQ, COMMIT_SQ, ray_depth)
21                processSQ(domain, SQ_IN, RETIRE_SQ)
22                processSQ(domain, SQ_OUT, COMMIT_SQ)
23          retire(RETIRE_SQ, hbuf)
24          swap(COMMIT_SQ, RETIRE_SQ)
25          swap(SQS_IN, SQS_OUT)
26       ray_depth = ray_depth + H
27    until !qs(SPILL_RQ).empty()
```

Listing 2: The main function for out-of-core rendering.

input and output shadow queues (Lines 24-25).

Following a schedule, the `for` loop (Lines 13-22) first filters out the rays unnecessary for ray-primitive tests (Lines 15-17), and while the filtered queues are nonempty, it loads a domain either from the domain cache or from storage and performs ray-primitive tests on the input rays (Lines 20-22).

Listing 3 shows how input rays are sorted out for ray filtering. Notice that there are two different data structures for ray queuing: a ray itself and a ray wrapper that includes a metadata having a domain hit depth and a distance between the ray origin and the domain. The metadata is needed for both filtering and gathering statistics for ray distributions. Given a ray data, the `filter` function decrements the ray counter and moves the ray to the output queue only if its predicate is true. The `predicate` function checks for valid visibility (Line 8). Furthermore, for a radiance ray, it checks whether a ray's hit point is farther than the current domain processed (Line 10), and for a shadow ray, it checks whether a ray is unoccluded (Line 12).

Listing 4 shows ray processing. The `processRQ` function performs ray-primitive intersection tests on filtered radiance rays. If a ray has a hit and successfully updates the history buffer with valid visibility, the function runs a shader to spawn secondary rays and

```
1  function filter(raytype, domain, in_qtype, out_qtype)
2     for each raydata in qs(in_qtype, domain.id).get()
3        statistics.decrement(domain.id, raydata.domain_depth)
4        if predicate(raytype, raydata) is true
5           qs(out_qtype, domain.id).push(raydata.ray)
6  function predicate(raytype, raydata)
7     // t_domain: distance between ray origin and current domain
8     if hbuf.visibility(raydata.ray) is success
9        if raytype is RADIANCE
10          return hbuf.fartherThan(ray.sample, raydata.t_domain)
11       if raytype is SHADOW
12          return not raydata.ray.occluded
```

Listing 3: Ray filtering for out-of-core rendering.

```
1  function processRQ(domain, qin_type, commitq_type, ray_depth)
2     for each ray in qs(domain, qin_type).get()
3        hitinfo = domain.intersect(ray)
4        if hitinfo.hit and hbuf.update(ray, hitinfo) is success
5           // rs: radiance rays, ss: shadow rays, spill queue populated internally
6           rs, ss = shade(domain, hitinfo, ray, ray_depth)
7           processSecondary(domain, rs, ss, commitq_type)
8  function processSecondary(domain, rs, ss, commitq_type)
9     for each r in rs
10       domainQueuing(r, statistics)
11    for each s in ss
12       if not domain.occluded(s)
13          no_hit_on_other_domains = domainQueuing(s, statistics)
14          if no_hit_on_other_domains
15             s.committed = true
16             qs(commitq_type).push(s)
17 function processSQ(domain, qin_type, commitq_type)
18    for each ray in qs(domain, qin_type).get()
19       if domain.occluded(ray)  ray.occluded = true
20       else if not ray.committed
21          ray.committed = true
22          qs(commitq_type).push(ray)
```

Listing 4: Ray processing for out-of-core rendering.

calls the `processSecondary` function, which either conditionally or unconditionally places the spawned rays to their hit domains and increments the ray counters (Lines 10 and 13). If an unoccluded shadow ray in the current domain does not intersect any other domains, it is pushed into the commit queue with its `committed` flag set. Likewise, the `processSQ` function performs ray-primitive occlusion tests on filtered shadow rays. If a ray has an occlusion, its `occluded` flag is set; otherwise, it is pushed to the commit queue with the `committed` flag set only if it has not been committed.

4.2 In Situ Rendering

For in situ rendering, we assume a sort-last data-space decomposition, modeling tightly-coupled in situ where data remains as distributed by the simulation. In this case, each node in the cluster owns a portion of the domains. Therefore, if a ray leaves the spatial extent of local data, but not of the dataset overall, the ray must be sent to the remote node that contains the next domain to traverse. For each node, we create a ray queue for each domain stored there and cycle through the queues round-robin to trace rays (in contrast to the out-of-core case where we actively schedule the next domain to process).

To improve performance, our in situ rendering system, shown in Fig. 5, provides processors with an ample amount of ray work through speculation, which is intended to increase available parallelism in the system. At a high level, each node performs ray-domain tests on input rays; speculatively places them in all hit domains, assuming they may or may not have a hit in a domain; sends each of the speculative ray batches to the node holding the data; performs

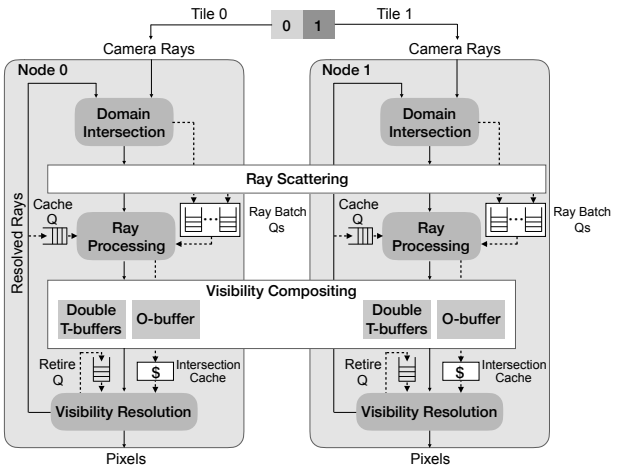


Figure 5: A speculative ray scheduling system for in situ rendering. Each node allocates a set of ray batch queues to buffer both local and remote rays. An intersection cache is used to buffer speculative secondary rays and their ray-primitive test results. A retire queue is used to buffer resolved shadow rays until they are retired. Resolved rays with cached test results are bypassed to a cache queue before the next round of ray scheduling begins.

ray-primitive tests on incoming ray batches; and locally caches speculatively spawned secondary rays along with the ray-primitive test results. Then, all nodes collectively perform visibility compositing to resolve the speculations, and the whole process repeats with ray-domain tests performed on the resolved secondary rays.

4.2.1 Domain Intersection and Ray Scattering

We initially divide the image plane into tiles to assign them to distinct nodes. Each node then creates independent camera rays for the assigned tile and performs ray-domain tests to speculatively place them in the ray batch queues. Since individual domain data are mapped to different nodes, ray batches are subsequently sent to owner nodes through point-to-point communication.

4.2.2 Ray Processing and Visibility Compositing

Finishing the ray communication, each node starts performing ray-primitive tests on ray batches. As a result, if secondary rays are spawned, each node immediately performs speculative ray-primitive tests on them and saves them into an *intersection cache* along with the test results. Once the processing of all rays including the speculative ones is finished, the cluster synchronizes to resolve all the accumulated speculations. Consequently, we allow the system to exploit as much parallelism as possible early on, without any node-to-node dependencies. Moreover, after all speculations are resolved, each node can fast move onto the speculative queuing step with the help of cached test results.

To resolve visibility, for every ray sample, each node maintains a t value and a single-bit occlusion flag after performing ray-primitive tests. For this purpose, each node utilizes a *distance buffer (t-buffer)* and an *occlusion buffer (o-buffer)*. Since many speculation rays may exist for the same ray sample location, each node constantly updates the buffers with test results as necessary.

Each node allocates two t-buffers which allow for processing both incident and outgoing rays in the same iteration. That is, one of them is used for populating t values for the current rays being processed and the other is used for resolving visibility of the rays generated in the previous iteration. Sending rays is expensive, especially when the ray count grows large under aggressive speculation. Hence, in order to minimize the communication cost, we use a history depth of one, in contrast to the out-of-core case. Furthermore, the o-buffer is

```
1  function insituSpray()
2     ray_depth = 0, partition = getPartition(process_id)
3     partition.loadAllData(), generate camera_rays
4     domainQueuing(camera_rays)
5     while true
6        work_counter.allreduce(SUM)
7        if work_counter.allDone() break
8        send and receive rays based on partition
9        for each domain in partition
10          processQ(CACHE, domain, tbuf_out, ray_depth)
11          processQ(LOCAL, domain, tbuf_out, ray_depth)
12          processQ(RECV, domain, tbuf_out, ray_depth)
13       if ray_depth < MAX_DEPTH  tbuf_out.allreduce(MIN)
14       if ray_depth > 0
15          obuf.allreduce(BITWISE_OR)
16          retire(qs(RETIRE).get(), tbuf_in)
17          obuf.reset()
18       tbuf_in.reset()
19       swap(tbuf_in, tbuf_out)
20       resolveVisibility(tbuf_in)
21       work_counter.populate()
22       ray_depth = ray_depth + 1
```

Listing 5: The main function for in situ rendering.

required since the results of occlusion tests on speculatively created shadow rays are scattered across the cluster.

After ray processing, the system composites the buffers across all nodes to achieve coherent global buffer state.

4.2.3 Visibility Resolution

With the t-buffer and o-buffer in consistent state, each node starts resolving visibility for the speculative rays in the intersection cache. A resolved radiance ray with a hit is pushed to a *cache queue* with its hit information for the next iteration of ray scheduling.

For a radiance ray, if a hit is found in the domain where it is spawned, we can possibly eliminate further ray-domain tests for the speculative queuing step if other domains are located farther than the hit point. Likewise, for an occluded shadow ray, redundancy caused by speculation can be reduced greatly. Therefore, upon caching the radiance ray, each node conditionally performs ray-domain tests to place it in the domains other than the current one.

If a resolved shadow ray is occluded, the single-bit occlusion flag in the o-buffer is set. Otherwise, each node commits the ray to a *retire queue*, maintains it until its visibility is determined in the next iteration for color updates, and unconditionally places it in all the domains intersected by the ray besides the current one. Notice that the shadow rays in the retire queue from the previous iteration must be retired before committing newly resolved ones.

4.2.4 Implementation

Listing 5 is the main function. The initialization resets the ray depth, loads all the domains mapped to each node, and speculatively queues camera rays into the hit domains (Lines 2-4). The `while` loop repeats until there is no more work available at the cluster level (Lines 6-7 and 21) and increments the ray depth at every iteration. If a node has one or more rays to work on, they are redistributed according to the data-domain mapping (Line 8). Then, each node starts processing different types of ray batches (Lines 9-12). `CACHE` refers to the rays with cached intersection results obtained from previous iterations, `LOCAL` and `RECV` refer to the rays not communicated and communicated, respectively. Then, the values of the o-buffer and output t-buffer are composited using `MPI_Allreduce` with the minimum and bitwise-or operators, respectively (Lines 13 and 15). Notice that the conditional statements (Lines 13-14) avoid unnecessary compositing for the first and last ray bounces. Then, the shadow rays in the retire queue, which are not speculative any more

```
1  function processQ(qtype, domain, tbuf_out, ray_depth)
2     for each ray in qs(qtype, domain, SHADOW).get()
3        if not obuf.occluded(ray) and domain.occluded(ray)
4           obuf.set(ray)
5     for each ray, hitinfo in qs(qtype, domain, RADIANCE).get()
6        if qtype is not CACHE  hitinfo = domain.intersect(ray)
7        if hitinfo.hit
8           if tbuf_out.update(hitinfo, ray) is success
9              rs, ss = shade(domain, hitinfo, ray, ray_depth)
10             for each r in rs
11                hitinfo2 = domain.intersect(r)
12                isect_cache.push(r, hitinfo2)
13             for each s in ss
14                hitinfo2 = domain.occluded(s)
15                isect_cache.push(s, hitinfo2)
```

Listing 6: Ray processing for in situ rendering.

```
1  function resolveVisibility(tbuf_in)
2     for each ray, hitinfo in isect_cache
3        if tbuf_in.visibility(ray) is success
4           if ray is RADIANCE
5              if hitinfo.hit
6                 qs(CACHE, hitinfo).push(ray, hitinfo)
7              domainQueuing(ray, hitinfo)
8           else if ray is SHADOW
9              if hitinfo.occluded  obuf.set(ray)
10             else
11                qs(RETIRE).push(ray)
12                domainQueuing(ray, hitinfo)
```

Listing 7: Visibility resolution for in situ rendering.

since their speculations were previously resolved, are used to update the pixel values, followed by an o-buffer reset (Lines 16-17). For the double buffering of t values, the output t-buffer is updated within `processQ` calls, and the input t-buffer is reset and swapped with the output t-buffer at the end of each iteration (Lines 18-19). Finally, visibility for all secondary rays is resolved, and the resolved rays are either conditionally or unconditionally queued into the domains depending on the ray type (Line 20).

Listing 6 performs ray-primitive tests on different types of ray batches. If an occluded shadow ray is found, the single-bit flag in the o-buffer, given by the screen-space sample ID and light ID, is set (Lines 2-4). It is safe to update the o-buffer here since visibility for the shadow ray was previously resolved. For a radiance ray with a hit, implicitly true for a cached ray, if it successfully updates the output t-buffer (i.e., the hit is the closest one so far), we run a shader function, perform ray-primitive tests on the secondary rays, and insert the rays and test results into the intersection cache (Lines 5-15).

Listing 7 resolves visibility for the secondary rays in the intersection cache by comparing each ray's local history (i.e., the parent ray's t value) against the input t-buffer. A resolved radiance ray with a hit is cached for reuse (Lines 4-6). Furthermore, the radiance ray is speculatively queued into the domain that is closer than the ray's hit point on a surface (Line 7). For a resolved shadow ray, if it is occluded, the corresponding bit in the o-buffer is set. Otherwise, the shadow ray is buffered into the retire queue and it is unconditionally queued into the hit domains (Lines 8-12).

5 Evaluation

To evaluate our speculative rendering system, we developed both the baseline and our methods starting from scratch. For out-of-core rendering, we implemented Navrátil's `LoadAnyOnce` dynamic scheduling algorithm [16] as a baseline. The baseline algorithm allows each process to schedule any domains based on a scheduling policy that chooses a domain with most rays. Therefore, to enable

Table 1: Benchmark datasets. Enzo: Astrophysics AMR. RM: Richtmyer-Meshkov fluid instability. DNS: Channel-flow turbulence.

Dataset	Enzo	RM	DNS1	DNS2
# Triangles (# Domains)	6 M (16)	108 M (52)	0.9 B (720)	1.8 B (1427)
Raw Data Size	940 GB	8 GB	483 GB	483 GB
Data Partitioning	Non-uniform rectilinear grid		Uniform rectilinear grid	

in situ rendering, where domains are statically assigned to distinct process, we slightly had to modify the original algorithm such that each process applies the scheduling policy to its own domains only.

Our system employs two-tier hierarchical bounding volume hierarchies (BVHs) for acceleration structures. An upper-level BVH for ray-domain queries is built on top of the bounding boxes representing domain boundaries, and a lower-level BVH for ray-primitive queries is built on top of each domain's surface data. The light-weight top BVH is built only once at start-up time prior to initiating a rendering job, whereas the lower-level BVH is built on a demand basis, as the associated domain data is brought into the domain cache from storage. For both BVH build and traversal, we use APIs offered by Embree v2.17.1 [23].

Upon loading a domain data, the domain cache starts building a BVH and saves it to an available cache entry. If all entries are full, it uses a least-recently used (LRU) eviction policy to free the BVH and domain data of the victim entry.

We measured all of the results by running our rendering system on the Skylake partition of the Stampede2 supercomputer at the Texas Advanced Computing Center. Each node has two 24-core Intel Xeon Platinum 8160 processors, each with 2-way hyperthreading, and 192 GB of system memory. We use up to 128 nodes that communicate via a 100 GB/sec Intel Omni-Path interconnection network. We compiled our code using v18.0.2 of the Intel C++ compiler and the Intel MPI library.

To verify rendering results and measure performance, we used real datasets obtained from scientific simulations. Table 1 summarizes the dataset characteristics. Fig. 1 shows images of the visualized datasets using our rendering system. We believe that the number of triangles for Enzo is immensely small compared to its raw data size because we extracted isosurfaces from the low-res first level of the AMR data. To obtain a model file for each domain, we partitioned each raw data into a 3D grid of domains and extracted isosurfaces using ParaView v5.4.1 [1]. For out-of-core rendering, each process owns all domains. For in situ rendering, we evenly assigned a set of adjacent domains to each process along the z-order curve.

To study behavior when rendering visually compelling images, we tested both path tracing (PT) and ambient occlusion (AO). For both PT and AO, we used 32 samples per pixel and generated 16 shadow samples for each hit point on a surface. We limited the maximum number of bounces to three for PT and one for AO. For out-of-core rendering, we used aggressive speculation: we set the history depth to the same value as each targeted ray depth. For materials, we assumed purely diffuse surfaces and used the Blinn-Phong shading model for direct illumination. For all datasets, we rendered images at 1024×1024 resolution.

We test our in situ system using the DNS dataset, as the originating simulation can operate at scales requiring in situ analysis, and our tests serve both as evaluation and prototyping for eventual in situ integration with the simulation. We test out-of-core rendering using the Enzo and RM datasets, where we incrementally shrink the domain cache size to emulate increasing memory pressure.

5.1 Results

5.1.1 Out-of-Core Rendering

Fig. 6 compares the performance of our method to the baseline for rendering Enzo and RM in out-of-core mode. We varied the size of domain caches from 25% to 100% of the total number of domains

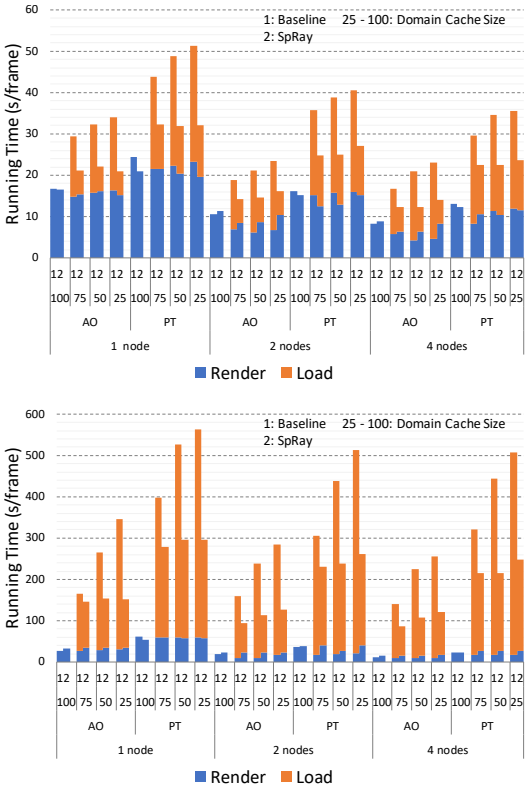


Figure 6: Performance comparison of out-of-core rendering for Enzo and RM (top and bottom). Domain cache sizes are calculated as a percentage of the total domain count. Our speculative method outperforms the baseline. Maximum speedup for total running time and loading time, respectively: (Enzo) 1.7×, 3.2×; and (RM) 2.3×, 2.7×.

for each dataset and used up to four compute nodes. Since out-of-core rendering requires frequent access to storage, we launched only one task and enabled 24 OpenMP threads that process rays in each domain in parallel. Data loading is run sequentially.

For Enzo, our method overall achieves 1.7× speedup compared to the baseline. However, if we compare the loading time, our method achieves up to 3.2× speedup. So our method has improved the loading time at the expense of compromised rendering time, which is due to aggressive speculation.

Likewise, for RM, our system achieves 2.3× speedup for overall running time and 2.7× speedup for loading time. As shown in the plot, the performance improvement comes from reducing the number of data loads, each of which is up to an order of magnitude more costly than the associated speculative rendering computation.

As we decrease the cache size, RM benefits greatly from speculation, since loads are more frequent with less cache data. Enzo shows less of this effect, since each domain is smaller and there are fewer domains overall. In addition, PT shows more benefit to ray speculation since its three bounces cause more data accesses and potentially access more of the data overall. AO and PT cast the similar number of rays per hit (i.e., 16 shadow rays for AO and one diffuse ray and 16 shadow rays for PT). However, AO has only one bounce and each ray travels a limited distance (i.e., shadow rays terminate on any hit), whereas PT has one additional diffuse ray per hit that travels until a hit is found or it leaves the data extent.

5.1.2 In Situ Rendering

For in situ rendering, we were only able to launch two MPI tasks for DNS1 and one MPI task for DNS2, due to the total size of domain data mapped to each process. If we launch more MPI tasks,

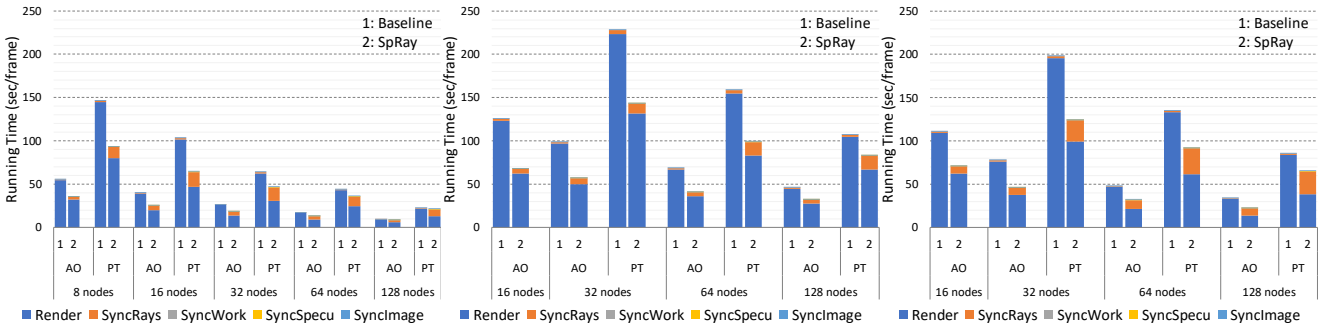


Figure 7: Performance comparison of in situ rendering for DNS1-side, DNS1-back, and DNS2 (from left to right). Our speculative method mostly outperforms the baseline. Maximum speedup for total running time and rendering time, respectively: (a) 1.6×, 2.2×; (b) 1.9×, 2.0×; and (c) 1.7×, 2.3×. Our method due to speculation particularly compromises ray communication time. The ratio of ray communication time for our method and the baseline: (a) 4 - 18; (b) 2 - 6; and (c) 7 - 21. Other types of communication are relatively negligible for both methods. The average percentage of ray communication time compared to total communication time for the baseline and our method, respectively: (a) 94%, 95%; (b) 99%, 96%; and (c) 95%, 98%.

the system quickly consumes most of the memory on Skylake and throws a memory allocation error. Our code currently does not support threading for the in situ rendering, but the current setup is sufficient for comparing the two methods. Because of the large data size, we used more nodes than in the out-of-core tests (i.e., 8 nodes for DNS1-side and 16 nodes for DNS1-back and DNS2).

For some test cases (e.g., PT on 16 nodes for DNS1-back and DNS2), our system ran out of memory due to excessive ray speculation. However, as the baseline system exhibited similar behavior on tests with fewer nodes (e.g., AO on 8 nodes or less for DNS2), the issue appears not to be unique to our system. For a quick test, reducing the image resolution to 512×512 on our speculative system alleviated memory pressure, leading to successful rendering jobs. This implies that we could apply blocking via tile scheduling so our method would still be able to render without a memory explosion. We will explore improved ray memory controls in future work.

Fig. 7 shows the results for the DNS dataset. DNS2 is the largest one that we have rendered in this paper. As in the out-of-core case, we compare both AO and PT performance with the baseline method and ours. In addition to the rendering time, we also measure the time for various types of synchronization across the cluster, including communication for ray batches (SyncRays), work counts (SyncWork), image compositing (SyncImage), and visibility compositing for our method only (SyncSpecu).

Overall, our speculative method achieves similar performance gains for both AO and PT. The running time is mostly dominated by the rendering time, which would greatly be reduced if we parallelize ray processing with threading and vectorization. Unlike the baseline, our method incurs longer ray communication time (i.e., 2 - 21× worse) because of all the speculative rays generated. For the same reason, the running time tends to approach that of the baseline with increase in the number of nodes, especially prominent with PT runs. Nevertheless, our method still achieves up to 1.8× speedup for overall running time and up to 2.3× speedup for rendering time.

The first two plots in Fig. 7 show the results for the DNS1 data from different view points. In both cases, our method mostly outperforms the baseline. The results for DNS1-back generally show longer running time for both the baseline and our method. This is mainly because there are more domains aligned along the viewing direction, requiring additional rendering computations.

6 Discussion

Fig. 8 implies that we can control speculation to improve performance. The loading time decreases as the history depth grows (i.e., $1 \leq H \leq B$) but starts to level off as the history depth becomes large enough (i.e., $H > B$). Another takeaway is that we need a mechanism

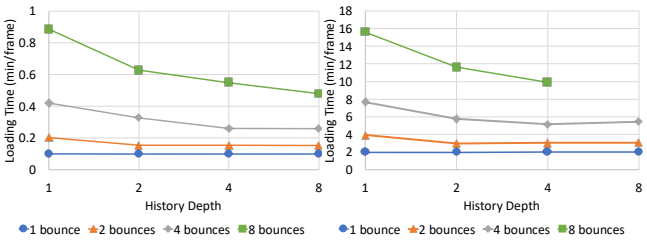


Figure 8: The effects of varying the history depth (H) and the number of ray bounces (B) when our path tracer was run on a node to render images of the Enzo and RM datasets (left and right) in out-of-core mode with 50% cache sizes (8 and 26 domains each). The test case of 8 bounces on RM fails for H>4 due to a memory explosion.

to stop performing speculation as it may exhaust memory space.

Although our current in situ renderer casts only a bounce worth of speculative rays, we could potentially take a similar approach to adjust speculation by having a larger t-buffer and o-buffer. This would greatly increase communication time. However, we believe that this challenge can be overcome by overlapping the rendering computation and ray communication.

Since our current in situ renderer can perform a simple round-robin scheduling, we can possibly incorporate our out-of-core scheduling capability to make it a hybrid of the two. However, having separate modes in our system serves a reasonable purpose for practicality. In situ visualization is preferred when abundant resources are available to a user. However, many practical reasons in a supercomputing environment (e.g., limited job queues, a long node acquisition time, etc.) suggest that one might prefer conducting visual analysis post hoc using fewer nodes.

7 Conclusion

In this paper, we have shown that speculatively producing and processing rays can correctly render large-scale data having billions of triangles. More importantly, we have experimentally demonstrated that our speculative system outperforms traditional methods for ray batching and scheduling in both out-of-core and in situ rendering scenarios. To verify both use cases, we have introduced two different system variants sharing the same spirit of the speculate-first-and-resolve-later concept. This new concept in ray scheduling for large data visualization has been enabled by defining different kinds of buffers and queues along with scheduling policies and algorithms to efficiently manage them.

ACKNOWLEDGMENTS

We thank Nick Malaya and Bob Moser at the Institute for Computational and Engineering Sciences at UT-Austin for providing the DNS data. We also thank the anonymous reviewers for their insightful comments. This work was funded in part by National Science Foundation grant ACI-1339863 and an Intel Visualization Center of Excellence award through the IPCC program.

REFERENCES

[1] U. Ayachit. *The ParaView Guide: A Parallel Visualization Application*. Kitware, Inc., 2015.

[2] J. Bigler, A. Stephens, and S. G. Parker. Design for parallel interactive ray tracing systems. In *IEEE Symposium on Interactive Ray Tracing*, pp. 187–196, Sept 2006. doi: 10.1109/RT.2006.280230

[3] R. Binyahib, T. Peterka, M. Larsen, K. L. Ma, and H. Childs. A scalable hybrid scheme for ray-casting of unstructured volume data. *IEEE Transactions on Visualization and Computer Graphics*, 2018. doi: 10.1109/TVCG.2018.2833113

[4] C. Brownlee, T. Ize, and C. D. Hansen. Image-parallel ray tracing using OpenGL interception. In *Proceedings of the 13th Eurographics Symposium on Parallel Graphics and Visualization*, EGPGV '13, pp. 65–72, 2013. doi: 10.2312/EGPGV/EGPGV13/065-072

[5] B. Budge, T. Bernardin, J. A. Stuart, S. Sengupta, K. I. Joy, and J. D. Owens. Out-of-core data management for path tracing on hybrid resources. *Computer Graphics Forum*, 2009. doi: 10.1111/j.1467-8659.2009.01378.x

[6] H. Childs, M. Duchaineau, and K.-L. Ma. A scalable, hybrid scheme for volume rendering massive data sets. In *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '06, pp. 153–161, 2006. doi: 10.2312/EGPGV/EGPGV06/153-161

[7] D. DeMarle, C. Gribble, S. Boulos, and S. Parker. Memory sharing for interactive ray tracing on clusters. *Parallel Computing*, 31(2):221–242, Feb. 2005. doi: 10.1016/j.parco.2005.02.007

[8] D. E. DeMarle, C. P. Gribble, and S. G. Parker. Memory-savvy distributed interactive ray tracing. In *Proceedings of the 5th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '04, pp. 93–100, 2004. doi: 10.2312/EGPGV/EGPGV04/093-100

[9] C. Eisenacher, G. Nichols, A. Selle, and B. Burley. Sorted deferred shading for production path tracing. In *Proceedings of the Eurographics Symposium on Rendering*, EGSR '13, pp. 125–132, 2013. doi: 10.1111/cgf.12158

[10] M. Howison, E. W. Bethel, and H. Childs. MPI-hybrid parallelism for volume rendering on large, multi-core systems. In *Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV'10, pp. 1–10, 2010. doi: 10.2312/EGPGV/EGPGV10/001-010

[11] T. Kato. "Kilauea"–parallel global illumination renderer. *Parallel Computing*, 29(3):289 – 310, 2003. Parallel graphics and visualization. doi: 10.1016/S0167-8191(02)00247-8

[12] T. Kato and J. Saito. "Kilauea": Parallel global illumination renderer. In *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, EGPGV '02, pp. 7–16, 2002. doi: 10.2312/EGPGV/EGPGV02/007-016

[13] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, July 1994. doi: 10.1109/38.291528

[14] B. Moon, Y. Byun, T.-J. Kim, P. Claudio, H.-S. Kim, Y.-J. Ban, S. W. Nam, and S.-E. Yoon. Cache-oblivious ray reordering. *ACM Transactions on Graphics*, 29(3):28:1–28:10, July 2010. doi: 10.1145/1805964.1805972

[15] P. A. Navrátil. *Memory-Efficient, Scalable Ray Tracing*. PhD thesis, The University of Texas at Austin, 2010.

[16] P. A. Navrátil, H. Childs, D. S. Fussell, and C. Lin. Exploring the spectrum of dynamic scheduling algorithms for scalable distributed-memory ray tracing. *IEEE Transactions on Visualization and Computer Graphics*, 20(6):893–906, June 2014. doi: 10.1109/TVCG.2013.261

[17] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan. Interactive ray tracing for isosurface rendering. In *Proceedings of the Conference on Visualization*, VIS '98, pp. 233–238, 1998. doi: 10.1109/VISUAL.1998.745713

[18] T. Peterka, H. Yu, R. Ross, and K.-L. Ma. Parallel volume rendering on the IBM Blue Gene/P. In *Proceedings of the 8th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '08, pp. 73–80, 2008. doi: 10.2312/EGPGV/EGPGV08/073-080

[19] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pp. 101–108, 1997. doi: 10.1145/258734.258791

[20] E. Reinhard, A. Chalmers, and F. W. Jansen. Hybrid scheduling for parallel rendering using coherent ray tasks. In *Proceedings of the IEEE Symposium on Parallel Visualization and Graphics*, PVGS '99, pp. 21–28, 1999. doi: 10.1145/328712.319333

[21] M. Son and S.-E. Yoon. Timeline scheduling for out-of-core ray batching. In *Proceedings of High Performance Graphics*, HPG '17, pp. 11:1–11:10, 2017. doi: 10.1145/3105762.3105784

[22] I. Wald, C. Benthin, and P. Slusallek. Distributed interactive ray tracing of dynamic scenes. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pp. 77–85, Oct 2003. doi: 10.1109/PVGS.2003.1249045

[23] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst. Embree: A kernel framework for efficient cpu ray tracing. *ACM Transactions on Graphics*, 33(4):143:1–143:8, July 2014. doi: 10.1145/2601097.2601199