Exploring the Spectrum of Dynamic Scheduling Algorithms for Scalable Distributed-Memory Ray Tracing

Paul A. Navrátil, *Member, IEEE,* Hank Childs, Donald S. Fussell and Calvin Lin, *Member, IEEE*

***Abstract*—This paper extends and evaluates a family of dynamic ray scheduling algorithms that can be performed in-situ on large distributed memory parallel computers. The key idea is to consider both ray state and data accesses when scheduling ray computations. We compare three instances of this family of algorithms against two traditional statically scheduled schemes. We show that our dynamic scheduling approach can render datasets that are larger than aggregate system memory and that cannot be rendered by existing statically scheduled ray tracers. For smaller problems that fit in aggregate memory but are larger than typical shared memory, our dynamic approach is competitive with the best static scheduling algorithm.**

***Index Terms*—Distributed memory, dynamic scheduling, parallel, ray tracing**

I. Introduction

Ray tracing is a well-known technique for rendering high-fidelity images. While originally designed for entertainment and artistic purposes, ray tracing is becoming increasingly important for scientific visualization, where its faithful adherence to the physics of light transport allows it to better express spatial relationships and realistic lighting than less computationally expensive rendering techniques.

To complicate matters, scientific visualization is now typically computed on the same hardware that produces the data. There are two reasons for this trend. First, most scientific simulations are performed on massively parallel supercomputing clusters and produce terabytes or more of data, so it is prohibitively expensive to ship this data to dedicated hardware for rendering and analysis. Second, it is impractical, and sometimes impossible, to pre-compute highly-tuned acceleration structures for these large datasets: The pre-processing would require significant additional machine time and disk space, and the resulting acceleration structure would consume significant additional DRAM, sometimes factors larger than the original dataset [1].

Thus, we now face the challenge of performing parallel ray tracing on distributed memory clusters, but with it comes the opportunity to construct visualization algorithms that can more-easily be used for *in-situ* and co-processing visualization, where an HPC simulation and interconnected visualization run on the same system simultaneously.

Whitted's traditional ray tracing algorithm is embarrassingly parallel, but when rendering secondary rays, the algorithm quickly loses ray coherence and exhibits poor memory system locality, as different rays touch different geometry and different parts of the acceleration structure. Thus, when the needed data are too large to fit in main memory—which is typical of scientific data—disk- and memory-efficient algorithms are needed for good performance.

To address the issue of poor ray coherence in the serial realm, Pharr, et al. [2] reformulate Whitted's algorithm to allow more flexible scheduling of ray-object intersection calculations. This formulation organizes rays and data into coherent work units, known as ray queues, which introduces a new tradeoff: Work units increase ray coherence—which in turn increases locality in the memory system—at the cost of increased memory state. Unfortunately, the use of the disk to cache excess ray state [2] can become intractable in a massively parallel environment due to I/O costs, specifically, file system contention from hundreds to thousands of processes performing extra I/O for rays both frequently and consistently throughout the rendering.

To create disk- and memory-efficient parallel algorithms, Navrátil et al. [3] generalize the flexible scheduling approach of Pharr, et al., to consider the issue of load balance, which is important for achieving good parallel efficiency. The basic idea is to consider locality when scheduling work units. This dynamic scheduling of rays and data can improve performance for large datasets where disk I/O and inter-processor communication limit performance.

The work by Navrátil et al. opens a rich space of dynamic scheduling, but their paper is limited in its evaluation, as it considers just one simple dynamic scheduling strategy. This paper is an extension of the work of Navrátil et al. that makes the following contributions:

- We provide a deeper exploration of their original dynamic scheduling policy (*LoadOnce*), and we explore two additional related dynamic scheduling policies: one that allows domain data to be replicated if there is sufficient demand for it on other processors (*LoadAnother*), and one that greedily processes the work unit that has the largest number of pending rays (*LoadAnyOnce*).
- We evaluate these schedulers on two parallel computers and across five dimensions that influence ray tracer performance: data set, data size, shading effects, camera

Paul A. Navrátil is with the Texas Advanced Computing Center, The University of Texas at Austin, email: pnav@tacc.utexas.edu.

Hank Childs is with the Department of Computer and Information Science, The University of Oregon and Lawrence Berkeley Nat'l Lab, email: hank@cs.uoregon.edu, hchilds@lbl.gov.

Donald S. Fussell and Calvin Lin are with the Department of Computer Science, The University of Texas at Austin, email: [fussell | lin]@cs.utexas.edu.

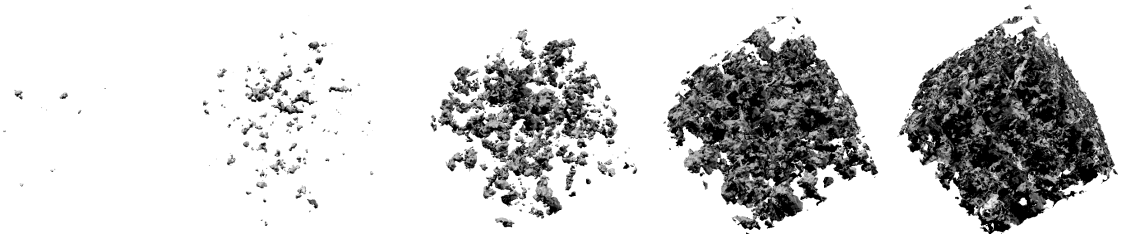


Fig. 1. The isosurfaces of the Perlin noise volumes used in our scaling study: Each is a 10% isovalue increment from 10% (far left) to 50% (far right).

position and number of processors.

- We empirically confirm Navrátil et al.'s result that when computing resources are limited, the *LoadOnce* dynamic scheduling policy outperforms static schedulers. Further, we demonstrate that the *LoadAnyOnce* policy performs best when shading generates many rays, such as with hemispheric sampling to compute diffuse reflections, and that allowing multiple processes to simultaneous load popular data domains using the *LoadAnother* policy provides little or no benefit over our other policies that map domains uniquely to processes.

The remainder of this paper proceeds as follows. We discuss related work in Section II. We then present ray tracing concepts essential to our discussion in Section III, followed by a description of our solution in Section IV. We describe our experimental methodology in Section V, an evaluation of the scheduling algorithms in Section VI, and a detailed scaling study on synthetic volumes in Section VII. We then conclude with a discussion of future work.

II. Related Work

To date, most parallel ray tracing research has been limited to either ray tracing on shared-memory machines [4], [5] or to ray casting (tracing only first-generation rays) on distributed memory architectures [6], [7], [8], [9]. Recent work on distributed-memory ray tracers [10], [11], [12], [13], [1], [14], [15] has demonstrated only modest scaling and has been hampered by various system limitations, including limited interconnect bandwidth and limited disk I/O bandwidth.

A. Shared-Memory Ray Tracing

Most parallel ray tracers assume a shared address space architecture [4], [16], [17], [5], [18], [19]. While these systems achieve impressive performance, the shared address space does not map to supercomputer clusters, and it tends to hide load balance concerns from the programmer. Explicitly out-of-core ray tracers [20], [21] also target shared memory systems, and their caching structures, if extrapolated to the distributed memory case, are similar to the distributed shared memory caching techniques described below.

B. Distributed-Memory Ray Tracing

On distributed memory computers, non-queueing ray tracers face a tradeoff: They can minimize data access by tracing ray groups that pass through contiguous pixels, or they can balance load by tracing disparate pixels in hopes of evenly distributing the rendering work [22]. These systems typically optimize performance by relying on expensive preprocessing steps, such as a low-resolution rendering pass to pre-load data on the processes [23] or an expensive pre-built acceleration structure to guide on-demand data loads [24], [12].

DeMarle et al. [1], [25], [13] use distributed shared memory to hide the memory complexities from the ray tracer. Their system achieves interactive performance for simple lighting models, but disk contention ruins performance if the scene does not fit in available memory. Moreover, their results rely on a preprocessing step to distribute the initial data, a step that typically takes several hours for a several gigabyte dataset. Ize et al. [14] update this approach using the Manta ray tracer [5] and modern hardware, but they experience similar memory and scaling limitations while retaining the expensive preprocessing step. Brownlee et al. [15] incorporate Ize et al.'s tracer into their OpenGL intercept framework using an explicit image-plane decomposition to achieve interactive performance for datasets that can fit in each process's available RAM.

Reinhard et al. [11], [10] distribute data across the cluster and assign tasks to processes based on load. This approach keeps camera and shadow rays on the originating process, while passing reflection and refraction rays to a process that contains the data required to process them. While the division of work is different, this technique is similar to that of DeMarle et al. in that it relies on data preprocessing and the caching of data in available system memory.

To balance load, the Kilauea system [26], [27] distributes the scene across all processes, but it replicates each ray on each process. This system requires scene data to fit entirely in aggregate memory, and it is unclear whether its small, frequent ray communication will scale beyond the few processes reported. It is also unclear whether the system can accommodate scientific data that does not have pre-tessellated surfaces.

To date, distributed memory ray tracers that queue and reorder rays have only been implemented on specialized hardware [28] and on a single workstation with GPU acceleration [29], [30], [31], solutions which are not feasible for the large datasets produced by supercomputing clusters.

C. Improving Memory Access Coherence

The importance of data coherence for improving rendering performance has been known for over thirty years [32]. Green and Paddon [33] propose three categories of coherence for ray tracing, and we use these categories to describe previous work in improving memory access coherence:

a) Image Coherence: Rays traced through adjacent pixels are likely to travel through the same regions of scene space, traverse the same acceleration structure nodes, and intersect

the same objects. Thus, image coherence can be achieved by tracing together primary rays that pass through contiguous pixels, such as in tiles [34] or along a space-filling curve [35]. Some parallel ray tracers sacrifice image coherence for the sake of load balance by tracing primary rays from widely separated pixels [23], [12].

b) Ray Coherence: Rays that travel a similar path are likely to require the same data for their traversal and intersection computations. Ray packeting [34], and similar techniques that trace a group of rays together, exploit ray coherence to achieve better performance. Pharr et al. [2] expand this concept to include rays that occupy the same region of scene space simultaneously, regardless of their origins or directions. We use this broader definition in our work. Secondary rays generated by ray-coherent primary rays often do not remain ray-coherent themselves, so ray reordering [36] and ray queueing [31], [30], [28], [37], [2], [38] are methods of building ray-coherent groups of secondary rays.

c) Data Coherence: Objects that are nearby in scene space are stored in nearby locations in machine memory. This relationship translates the coherent references of an algorithm into coherent requests to memory. Data coherence must exist for image- or ray-coherent traversals to maximize efficient use the memory system [33], [16], [2]. Otherwise, coherent accesses in the scene might result in random requests in memory, eliminating the coherence benefit.

From a parallelization perspective, the conditions for image and ray coherence are best served by algorithms that divide rays over processors and then load data as needed, while the conditions of data coherence are best served by algorithms that divide data over processors and pass ray data to the corresponding processor as it moves through the volume.

III. Background

This section explains—by summarizing the important issues in producing an efficient parallel ray tracing algorithm for large distributed memory computers—how parallel ray tracing is essentially a scheduling problem.

Consider a ray tracing problem with a fixed data set, camera location, and lighting properties. Over the entire computation, including both primary and secondary rays, this problem has some fixed number of calculations to perform, N, and each calculation matches a ray with the geometry that it intersects. Ideally, each calculation would take a fixed (and short) period of time, and ideally a ray tracing algorithm would simply divide the work evenly over P processes, leading to a P-fold speedup. Unfortunately, this ideal case rarely occurs: The amount of time needed to carry out a calculation often varies greatly due to cost of getting the ray and its intersecting geometry together on the same process.

Focusing on systems where individual processes own rays, there are three primary actions that a process P_i can make when considering ray R:

1) P_i calculates the next intersection of R.
2) P_i directs P_j to calculate the next intersection of R.
3) P_i takes no action with R.

For the first action, the bottleneck comes from fetching the geometry that R intersects, whether it has to be transferred from cache to registers or whether it has to be read from disk. These data movement costs often greatly exceed the cost of calculating the intersection itself. For the second action, the algorithm might be moving the ray intersection to a process that already contains the object data, where the calculation can be performed more efficiently, or it might be moving the ray intersection to a less heavily loaded process, in which case the calculation has the potential to be computed earlier. The third action simply introduces an additional delay, meaning that the computation is not being advanced.

In short, the efficiency of a parallel ray tracing algorithm depends on the time and location at which each calculation is performed, which introduces a scheduling problem whose goal is to minimize the overhead required to perform each individual calculation. The specific goals are to minimize disk reads, network communication, and load imbalance. As these costs are reduced, the performance of the algorithm increasingly mirrors that of an optimal one that carries out intersections as fast as it can perform floating-point arithmetic.

A. Static Ray Schedulers

Previous distributed memory ray tracers use static schedules that statically partition data among processes. The first, *image-plane decomposition*, partitions the screen space into a grid of domains and allocates to each domain any object whose projection overlaps that grid cell. The second, *data-domain decomposition*, partitions world space into a grid of domains, assigning to each domain all objects that overlap that domain.

```
01 RayQueue queue, all_queues[]
02 Domain d
03
04 ProcessQueue(RayQueue queue):
05  while (! queue.empty() ):
06    Ray r = queue.top()
07    queue.pop()
08
09    # Intersect ray r against domain d
10    # add any new rays to the queue
11    PerformRayOperations(d, r, queue)
12
13    if (! RayFinished(r) ):
14      Enqueue(all_queues, r)
15    else:
16      ColorFramebuffer(r)
17
18 GenerateRays():
19  if ( Image or Dynamic schedule ):
20   rays[] = create camera rays for my image tile
21  else: # Domain schedule
22   rays[] = create camera rays for my domain(s)
23  return rays[]
24
25 EnqueueRays(rays[]):
26  RayQueue queues[]
27  for r in (rays[]):
28    int q = r.queue_num
29    queues[q].push(r)
30  return queues[]
```

Fig. 2. Pseudocode for ProcessQueue(), GenerateRays() and EnqueueRays(). These are used in each schedule's pseudocode (see Figures 3 – 5). `PerformRayOperations()` includes the traversal, intersection, shading and spawning of new rays.

Static Image-Plane Decomposition: Perhaps the simplest and most common way to parallelize a ray tracer is to partition objects and primary rays among processes according to their

```
01 ImageTrace():
02  Ray rays[] = GenerateRays()
03  RayQueue queues[] = EnqueueRays(rays)
04
05  while (! queues.empty() ):
06    RayQueue q = FindQueueWithMostRays(queues)
07    Domain d = LoadDomain(q.domain_id)
08    ProcessQueue(q)
09    queues.delete(q) # q is now empty
10
11  MergeFramebuffers()
```

Fig. 3. Pseudocode for the Static Image-Plane Decomposition Schedule.

overlap of the image plane. See Figure 3. Initially each process is responsible for an equal-sized subset of the image plane comprising one or more grid cells; each grid cell maintains a queue of rays that have entered that domain; and data are loaded to each process on demand. The primary rays are evenly divided among these queues (and thus among the processes, see line 2). As rays are processed forward from their origin, they are re-queued whenever they leave one grid cell and enter another (line 8 and see Figure 2), which always occurs locally (i.e. rays are never sent to another process). When secondary rays are spawned, they are also enqueued and processed locally. At each scheduling step, each process selects for further processing its grid cell domain that has the largest number of local rays queued for further processing (line 6).

Note that primary rays always remain within the portion of the scene initially assigned to their local process. Secondary rays, however, can enter any region of the scene, including regions not initially allocated to their process. In such cases, a process will need to load partitions which were not initially assigned to it. In the worst case, all processes could have to load all portions of the scene. While primary rays are initially evenly distributed, this strategy can exhibit poor load balance as the computation proceeds because secondary rays can be extremely unevenly distributed among processes. It can also be difficult for the initial partition to evenly distribute the objects across processes for scenes that have a highly uneven spatial distribution.

This strategy has been commonly used [15], [5], [1], [25], [13], [23], [4], [39], [24], [12] and also corresponds to the demand-driven component of the schedule used by Reinhard et al. [11], [10]. It directly parallelizes a Pharr-like approach by using multiple serial instances run in parallel, where each instance is seeded with a subset of camera rays.

```
01 DomainTrace():
02  Ray rays[] = GenerateRays()
03  RayQueue queues[] = EnqueueRays(rays)
04
05  Domain last_d = NONE
06  boolean done = FALSE
07  while (! done ):
08    # each process only has rays for its domains
09    RayQueue q = FindQueueWithMostRays(queues)
10    if (q.domain_id != last_d):
11      Domain d = LoadDomain(q.domain_id)
12      last_d = q.domain_id
13
14    RayQueue new_rays = ProcessQueue(q)
15    queues.delete(q) # q is now empty
16
17    SendLiveRaysToNextProcess(queues)
18    done = AllProcessesReportEmpty()
19
20  MergeFramebuffers()
```

Fig. 4. Pseudocode for the Static Domain Decomposition Schedule.

Static Domain Decomposition: Another simple strategy is to spatially subdivide the dataset in world space and to distribute these domains among the available processes such that each process is responsible for a similar number of objects. See the pseudocode in Figure 4. A process can be assigned multiple domains if there are more domains than processes. Again, rays that have entered a domain are enqueued at each domain (line 3). As with the image-space approach, at each scheduling step each process selects the assigned domain with the largest number of local rays queued for processing (line 9), and rays are re-queued as they leave one domain and enter another (line 17). Note that all rays of the selected queue are processed (line 14) and the queue is emptied (line 15) before selecting the next queue. Unlike the image-space approach, however, the initial assignment of domains to processes is fixed throughout the computation. A ray that leaves its local process must be sent to the process responsible for the domain that it is entering.

This type of decomposition can more naturally balance the number of objects that each process is responsible for, regardless of how they project into screen space, but it can still exhibit poor load balance if the number of rays enqueued at each domain cell varies. Unlike the image-decomposition approach, even the primary rays may be poorly balanced. However, it does much less loading of data into memory in the global illumination case since all domain assignments are fixed. This is traded off for the need to communicate ray state among processes.

This strategy is used by several ray tracers [28], [40], [41] and in the data parallel component of the scheduling strategy in Reinhard et al. [11], [10]. It is a typical approach for large-scale volume renderers [6], [8], [9], [7]. Like the image-plane decomposition strategy, it parallelizes a Pharr-like approach by using multiple serial instances run in parallel, where each instance is assigned a set of domains and where rays are moved among processes.

IV. Our Algorithm

In this section, we present a dynamic scheduling framework and three dynamic scheduling policies that relax the static assignment of domains to processes to improve both locality and load balance. The pseudocode for this overall scheduling framework is shown in Figure 5. The basic idea is to start with a uniform volumetric world-space partitioning of the scene data into domains—as in a static domain decomposition—and to partition primary rays into queues for each process uniformly across the image plane—as in a static image plane decomposition. Thus, the ray queues are initially the same size for each process, as in the static image plane decomposition approach.

Each process loads on demand an initial data domain as the first ray in its queue enters the initial domain (lines 6–7). For all of our policies, we allow only one data domain to be loaded into the memory of a process at any given time, so initially each process will queue for later processing any of its rays that enter a different domain than the one initially loaded (line 3). Since the initial data domains need not contain the same amount of data (indeed, some may be completely empty),

```
01 DynamicTrace():
02  Ray rays[] = GenerateRays()
03  RayQueue queues[] = EnqueueRays(rays)
04  Schedule schedule
05
06  RayQueue q = FindQueueWithMostRays(queues)
07  Domain d = LoadDomain(q.domain_id)
08  Domain last_d = q.domain_id
09
10  ProcessQueue(q)  # traverse rays in queue
11  queues.remove(q) # q is now empty
12
13  boolean done = AllProcessesReportEmpty()
14
15  while (! done ):
16    # schedule next round of ray processing
17    case (schedule_policy):
18    :LoadOnce:
19       = LoadOncePolicy(last_d, queues)
20    :LoadAnyOnce:
21      q = LoadAnyOncePolicy(last_d, queues)
22    :LoadAnother:
23      q = LoadAnotherPolicy(last_d, queues)
24
25    ReceiveSchedule( schedule )
26    SendLiveRaysToNextProcess(queues)
27
28    d = schedule( my_process_id )
29    q = queues[d]
30
31    if (q.domain_id != last_d):
32      d = LoadDomain(q.domain_id)
33      last_d = q.domain_id
34
35    ProcessQueue(q)  # traverse rays in queue
36    queues.remove(q) # q is now empty
37
38    done = AllProcessesReportEmpty()
39
40  MergeFramebuffers()
```

Fig. 5. Pseudocode for Dynamic Scheduling Framework. ProcessQueue() is defined in Figure 2 and the policies are defined in Figures 6–8.

and since the rays in each process' queue can enter different domains, the initial computational load is not balanced any more than it is in the static image decomposition scheme, despite the fact that the ray queues are all the same size.

Our dynamic scheduling framework proceeds in stages, where each stage consists of a scheduling phase, a communication phase, and a computation phase. In the first stage, a special scheduling phase determines the initial workload of each process as described above and the communication phase is skipped (lines 2–13). In the scheduling phase of subsequent stages, a master process assigns to each slave process a data domain based on ray demand information that the master has received from the slave processes in previous stages (lines 16–23). By varying the criteria used by the master process for this assignment, different scheduling policies can be obtained.

Once the domain assignment has been determined, the master communicates the domain mapping to the slave processes (line 25), which then exchange rays in the communication phase (line 26). In this phase, each slave partitions its processed rays according to the domain that it is entering. It then exchanges rays with each of the other slaves to move rays to the appropriate domain. Once each slave has completed its exchange of rays with all other slaves, it enters its computation phase (lines 28–36). In this phase, it simply traces each ray in its next queue of unprocessed rays until it either intersects an object in its data domain or reaches the domain boundary.

If a ray intersects an object, child rays are spawned as needed and added to the local unprocessed ray queue for local

```
01 LoadOncePolicy( current_domain,
02                 local_ray_queues ):
03  foreach q in local_ray_queue:
04    queue_map[ q.domain_id ] = q.ray_count
05
06  if slave:
07    SendQueueInfoToMaster( current_domain,
08                           queue_map )
09
10  else if master:
11    current_schedule = {}
12    new_schedule = {}
13    foreach p in all_processes:
14      ReceiveQueueInfo( current_domain[p],
15                        queue_map[p] )
16
17      # note domains that have rays pending
18      foreach q in queue_map[p]:
19        # track total rays
20        # that request this domain
21        domains_to_schedule[ q.domain_id ]
22                             += q.ray_count
23
24      # note the domain (if any)
25      # loaded at each proc
26      if ( current_domain[p] != NONE ):
27        current_schedule[p]
28                         = current_domain[p]
29
30    # if a domain is already loaded, keep it
31    # and remove it from further processing
32    foreach d in domains_to_schedule:
33      proc_d = current_schedule.find( d )
34      if ( proc_d != NONE ):
35        new_schedule[proc_d] = d
36        domains_to_schedule.remove( d )
37      else:
38        victims.push( proc_d )
39
40    # sort remaining domains
41    # by number of waiting rays
42    domains_to_schedule.sort()
43
44    # for all victim procs,
45    # assign the next unassigned domain
46    # (with the next most rays requesting it)
47    foreach v in victims:
48      if ( domains_to_schedule.empty() ):
49        break
50      Domain d = domains_to_schedule.pop()
51      new_schedule[v] = d
52
53    SendScheduleToAllProcesses( new_schedule )
```

Fig. 6. Pseudocode for LoadOncePolicy(), called by the scheduling framework.

processing in the current stage. Note that all rays, whether primary or secondary rays, carry as part of their state the pixel to which they contribute illumination information. This pixel information is copied to the state of each spawned child ray. Thus, when a ray intersects an object, the partial illumination information for its pixel that can be determined at that ray object intersection is added to a local copy of the frame buffer maintained by each process. This parallel version of Pharr's ray deferral scheme allows the full illumination computation to be completed by merging all slave processes' frame buffers at the end of the computation.

If the domain boundary is reached, the ray is added to the set associated with the domain that it is entering for further processing in a subsequent stage. Once the local queue of unprocessed rays is empty, the slave communicates with the master the set of domains that its spawned rays are entering and the number of rays entering each domain. This information will be used by the master for domain assignments in the next stage. Once the master has received this information from all slaves, the stage ends. If all slaves indicate that no rays are

```
01 LoadAnyOncePolicy( current_domain,
02                    local_ray_queues ):
03   foreach q in local_ray_queue:
04     queue_map[ q.domain_id ] = q.ray_count
05
06   if slave:
07     SendQueueInfoToMaster( current_domain,
08                            queue_map )
09
10   else if master:
11     current_schedule = {}
12     new_schedule = {}
13     foreach p in all_processes:
14       ReceiveQueueInfo( current_domain[p],
15                         queue_map[p] )
16
17       # note domains that have rays pending
18       foreach q in queue_map[p]:
19         # track total rays
20         # that request this domain
21         domains_to_schedule[ q.domain_id ]
22                              += q.ray_count
23
24       # note the domain (if any)
25       # loaded at each proc
26       if ( current_domain[p] != NONE ):
27         current_schedule[p]
28                           = current_domain[p]
29
30     # every proc is a victim
31     victims = all_processes
32
33     # sort domains
34     # by number of waiting rays
35     domains_to_schedule.sort()
36
37     # for all victim procs,
38     # assign the next unassigned domain
39     # (with the next most rays requesting it)
40     foreach v in victims:
41       if ( domains_to_schedule.empty() ):
42         break
43       Domain d = domains_to_schedule.pop()
44       new_schedule[v] = d
45
46     SendScheduleToAllProcesses( new_schedule )
```

Fig. 7. Pseudocode for LoadAnyOncePolicy(), called by the scheduling framework.

pending, the final stage has completed and the illumination is computed by merging the slave processes' frame buffers as described above (line 40).

Unlike static approaches, our scheduling framework provides some latitude in assigning both ray state information and domain data to processes. Thus, we can experiment with different policies—used by the master process in the scheduling phase—that can be designed to optimize data reloading, ray state communication, load balance, or a combination of such factors to achieve the best performance for a given workload configuration and given set of machine parameters. Our dynamic scheduling system opens a wide range of possible assignment strategies. In previous work [3], we presented one possible dynamic scheduling policy, which we now dub *LoadOnce*, and compared it with the static policies described above. In this paper, we extend our analysis by adding two additional scheduling policies that help us better understand the tradeoffs involved.

Three Dynamic Scheduling Policies: We first describe how the *LoadOnce* policy makes domain assignments. Pseudocode for this policy is given in Figure 6. Recall that at the beginning of a stage, the master process has (1) a list of data domains that unprocessed rays are entering and (2) the number of such rays entering each such domain, aggregated from the results of

```
01 LoadAnotherPolicy( current_domain,
02                    local_ray_queues ):
03   foreach q in local_ray_queue:
04     queue_map[ q.domain_id ] = q.ray_count
05
06   if slave:
07     SendQueueInfoToMaster( current_domain,
08                            queue_map )
09
10   else if master:
11     current_schedule = {}
12     new_schedule = {}
13     foreach p in all_processes:
14       ReceiveQueueInfo( current_domain[p],
15                         queue_map[p] )
16
17       # note domains that have rays pending
18       foreach q in queue_map[p]:
19         # track total rays
20         # that request this domain
21         domains_to_schedule[ q.domain_id ]
22                              += q.ray_count
23
24       # note the domain (if any)
25       # loaded at each proc
26       if ( current_domain[p] != NONE ):
27         current_schedule[p]
28                           = current_domain[p]
29
30     # if a domain is already loaded, keep it
31     # and keep it for further processing too
32     foreach d in domains_to_schedule:
33       proc_d = current_schedule.find( d )
34       if ( proc_d != NONE ):
35         new_schedule[proc_d] = d
37       else:
38         victims.push( proc_d )
39
40     # sort remaining domains
41     # by number of waiting rays
42     domains_to_schedule.sort()
43
44     # for all victim procs,
45     # assign the next unassigned domain
46     # (with the next most rays requesting it)
47     foreach v in victims:
48       if ( domains_to_schedule.empty() ):
49         break
50       Domain d = domains_to_schedule.pop()
51       new_schedule[v] = d
52
53     SendScheduleToAllProcesses( new_schedule )
```

Fig. 8. Pseudocode for LoadAnotherPolicy(), called by the scheduling framework.

all slave processes in the previous stage (lines 17–22). Popular domains with large ray counts represent the best opportunity to get the most additional work done with the fewest new domain loads. Thus, the master sorts demanded domains by their ray counts (line 42). Assuming that loads are expensive, the master avoids loading new domains by allowing any slave process that has a domain with a non-zero ray count to process that same domain in the next stage (lines 32–38). Any slave with unassigned domains is put on a *victims* list (line 38). For each process on the victims list, new domains are assigned in order of decreasing ray count (lines 47–51). Once either the victims list or the demanded domain list is empty, the scheduling phase ends and the computation proceeds.

The first new policy, which we call *LoadAnyOnce*, attempts to reduce the *LoadOnce* policy's scheduling overhead while retaining its advantages over static scheduling. Rather than preferentially keeping already-loaded domains resident, this policy (see Figure 7) simply sorts the available domains by ray count (line 35) and then assigns domains to slave processes in order of decreasing ray count, i.e. the victims list is simply

the list of all slave processes (line 31). This change offers two potential advantages over *LoadOnce*. First, the scheduler can run faster because it doesn't need to search the domain demand list for already-loaded domains (see Figure 6, lines 30–38). Second, it avoids the pathology that may afflict *LoadOnce* in which a resident domain with very few entering rays takes precedence over domains with significantly higher ray counts.

The second new policy, which we call *LoadAnother*, simplifies *LoadOnce* in a different way that has the potential to improve load balance for highly skewed ray distributions. *LoadAnother* (Figure 8) computes the victims list just as for *LoadOnce*, i.e. it prioritizes resident domains that have non-zero ray count and puts all remaining slaves on the victims list. However, these already resident domains are not removed from the demanded domains list (i.e. Figure 6, line 36), which allows them to be assigned by the master to a second slave if their ray count is sufficiently high. Since some slaves can now send their rays to either of two processes that have the same domain, we must change the communication phase to allow each slave to choose the target copy. For simplicity, this choice is made randomly; the slave sends all of its rays that are destined for the given domain to the chosen target process.

V. Methodology

The optimal scheduling algorithm depends on the characteristics of the available hardware and the configuration of the ray tracing problem; this configuration includes multiple factors, each of which must be considered when evaluating the scheduling algorithms. For this study, we consider the following factors:

- **Dataset**. Certain datasets may exhibit unique behaviors.
- **Data size**. We consider the data size separately from the dataset, because we can obtain different resolutions of the same dataset.
- **Ray effects**. We explore the impact of secondary ray effects, such as specular and diffuse lighting.
- **Camera position**. We explore the effects of camera zoom on performance.
- **Concurrency**. We explore performance as a function of the number of processors used.

When studying performance, each of these factors can conflate with the others. Further, studying the entire cross-product of all factors leads to a prohibitive number of experiments. So we perform our analysis in two phases; for each phase, we hold some factors constant and vary others. This separation produces a tractable number of tests and isolates the impact of individual test factors. Ultimately, this approach allows us to identify configurations where a particular scheduling algorithm can be most useful. The first phase (Section VI) focuses on evaluating all five scheduling algorithms, while the second phase (Section VII) focuses on the scalability of our three dynamic algorithms.

A. System Configuration

Our experiments are run on *Longhorn* and *Stampede*, both hosted at the Texas Advanced Computing Center. *Longhorn* is a 2,048 core, 256 node cluster, where each node contains two four-core Intel Xeon E5540 "Gainestown" processors and 48 GB of local RAM. All nodes are connected via a Mellanox QDR InfiniBand fabric, and we use MVAPICH2 v1.4 for MPI-based communication. *Stampede* is a 462,462 core, 6400 node cluster, where each node contains two eight-core Intel Xeon E5-2680 "Sandy Bridge" processors, a 61-core Intel Xeon Phi SE10P coprocessor (not used in our experiments) and 32 GB of local RAM. All nodes are connected via a Mellanox FDR InfiniBand fabric, and we use MVAPICH2 v1.9a2 for MPI-based communication.

Our ray tracer is implemented within VisIt [42], a parallel visualization tool designed to operate on large-scale data. We use the VisIt infrastructure to load data and to generate isosurfaces, while all code related to ray tracing and ray scheduling is our own. To focus on the effects of the schedulers, we turn off all caching within the VisIt infrastructure, so that only one dataset is maintained per process. Each load of non-resident data accesses the I/O system.

All MPI communication in our implementation is two-way asynchronous. This implementation decision impacts dynamic schedulers the most, since they have the highest degree of communication among processes.

TABLE I
Perlin Noise Dataset Configurations

	256^3	512^3	1024^3	2048^3	4096^3
domains	8	64	64	512	512
disk size	0.065 GB	0.51 GB	4 GB	32.3 GB	257 GB
volume %			polygons		
10%	0.01M	0.02M	0.08M	0.83M	3.33M
20%	0.11M	0.41M	1.68M	8.03M	32.2M
30%	0.56M	3.56M	13.5M	45.3M	181M
40%	1.37M	11.7M	41.1M	117M	468M
50%	2.00M	18.7M	64.1M	177M	711M

B. Datasets

Our study uses a set of synthesized Perlin noise [43] volumes scaled by powers-of-two from 256^3 to 4096^3. From these volumes we extract five isosurfaces that yield increasing geometric complexity, from a 10% isovalue to 50%. We show the full details of the volume size, decomposition and isosurface complexity in Table I. Sample images can be found in Figure 1.

We extract isosurfaces using VisIt's VTK-based isosurfacing and internal BVH acceleration structure, and we then ray trace the returned geometry using two directional lights. The specular reflection cases use two-bounce reflections and the diffuse reflection cases use Monte Carlo integration with sixteen hemispheric samples and a 10% termination chance per bounce. While the isosurface extraction and BVH generation is performed each time the dataset is loaded from disk, the cost is small relative to the I/O cost. These costs are all included in the rendering times in Sections VI and VII. We do not save the BVH since that would incur additional disk and I/O costs. In addition, we use the coarse acceleration structure from the spatial decomposition implied by the disk image of each dataset, since large simulation-derived datasets are typically stored across multiple files.

VI. Evaluating Scheduling Algorithms

For the first phase of our evaluation, we vary five factors:

- Scheduling algorithm: *Image*, *Domain*, *LoadOnce*, *LoadAnyOnce*, and *LoadAnother* (5 options)
- Data size: 1024^3 and 4096^3 versions of the 40% isosurface from the Perlin noise dataset (2 options)
- Camera position: five positions ranging from "zoomed out" to "zoomed in" (5 options)
- Ray effects: primary rays and shadows, as well as primary rays, shadows, and diffuse reflection rays (2 options)
- Concurrency: 16 processors and 64 processors (2 options)

Thus, we run 200 tests ($= 5 \times 2 \times 5 \times 2 \times 2$), terminating any test that exceeds four hours.

The results (see Figure 9) lead to the following conclusions:

- *LoadAnyOnce* is the only scheduling algorithm that completes all renderings within the four hour window.
- *LoadAnother* exceeds four hours most often, followed by *Image*. That said, these algorithms are the fastest scheduling algorithms in some of the other configurations.
- For the regimes where the static schedulers are expected to perform best, the dynamic algorithms are competitive with the traditional schedulers. For example, in the tests calculating just primary rays and shadows with the 4096^3 dataset and using 64 processors, the dynamic algorithms were faster than the others for all camera positions.
- Both *Domain* and *Image* can exhibit poor behaviors based on the zoom factor. The dynamic algorithms are more successful at avoiding pitfalls.

We also consider the efficiency of each algorithm, which informs our ultimate goal of understanding which scheduling algorithm has the fastest execution time for a given configuration. Parallel efficiency is typically measured as a ratio of serial and parallel execution times. In our case, it is generally intractable to run our algorithms serially on these large datasets, and in any case our interest is in comparing different parallel algorithms. We therefore define efficiency by considering the processor's activity. If a processor is currently "rendering,", i.e., calculating intersections or other work which tangibly advances the ray tracing, then we consider that processor to be *productive.* If all processors are productive for the entire execution time, then the algorithm should exhibit perfect scalability, modulo issues such as hot caches. In a parallel setting processors are frequently not productive, and typically their efficiency is eroded either by data loads from disk or by idle time. In Figure 10, we plot the efficiency of each test—that is, the proportion of the total test time (over all processors) that is productive. The results suggest that the dynamic scheduling algorithms have a slight advantage in efficiency, but the larger point is that certain tests make it very difficult to achieve high efficiency, for example those that use 4096^3 data. In the remainder of this section, we examine the relationships between efficiencies, test configurations, and scheduling algorithm.

Table II compares the scheduling algorithms across different configurations. Each row holds one factor constant and averages the efficiencies over the remaining factors. The averages are unweighted, meaning that tests that finish quickly contribute equally to the average as those that run for a long time. Tests that did not finish in four hours were given an efficiency of zero. The zoomed in and zoomed out tests only reflect a subset of all tests. Note that the variance for each average is high and conclusions must be drawn with caution. This table is complemented by Table III, which shows the percentage of tests that failed to complete.

These tables reveal several points:

- Efficiency for the 4096^3 data is very low, partially because the number of ray tracing operations (i.e., intersections, ray calculation) is proportional to the image size, which was constant in our tests. As larger datasets take more overhead to manage, the resulting efficiencies drop.
- *Image* does well with zoomed in camera positions, since less data needs to be loaded. The *Domain* algorithm does poorly, since this configuration creates a bottleneck.
- When the camera is zoomed out, the *Domain* algorithm does better, and the *Image* algorithm does worse.
- Of the dynamic algorithms, *LoadAnyOnce* has the highest efficiency (or near highest) for every factor.
- *LoadAnother* has the worst efficiency of the dynamic schemes, partially because it exceeds the four hour time limit so often.
- Comparing between *Image* and *LoadAnyOnce*, *LoadAnyOnce* has a better average efficiency in the cases where *Image* does poorly (4096^3 data or zoomed out camera position) and provides similar efficiencies on most other factors. The only case where *LoadAnyOnce* is clearly inferior is with zoomed in camera positions.

TABLE II
Efficiency of scheduling algorithms for different test configurations.

Factor	*Domain*	*Image*	*Load-Once*	*Load-Any-Once*	*Load-Another*
16 processors	6.4%	9.2%	7.7%	9.0%	7.2%
64 processors	2.9%	3.5%	3.2%	3.1%	3.0%
1024^3 data	8.6%	12.7%	10.6%	10.9%	10.0%
4096^3 data	0.7%	0.0%	0.3%	1.1%	0.1%
primary rays	2.7%	3.9%	4.1%	4.1%	3.9%
secondary rays	6.7%	8.8%	6.8%	8.0%	6.3%
zoomed out	5.2%	2.9%	5.5%	6.2%	4.9%
zoomed in	3.1%	13.2%	4.9%	5.4%	4.4%

TABLE III
Percentage of tests that failed to complete within four hours for different test configurations and algorithms.

Factor	*Domain*	*Image*	*Load-Once*	*Load-Any-Once*	*Load-Another*
16 processors	10%	25%	25%	0%	45%
64 processors	15%	25%	20%	0%	20%
1024^3 data	5%	0%	0%	0%	0%
4096^3 data	20%	50%	45%	0%	65%
primary rays	5%	0%	0%	0%	20%
secondary rays	20%	50%	45%	0%	45%
zoomed out	0%	25%	12.5%	0%	12.5%
zoomed in	37.5%	25%	25%	0%	37.5%

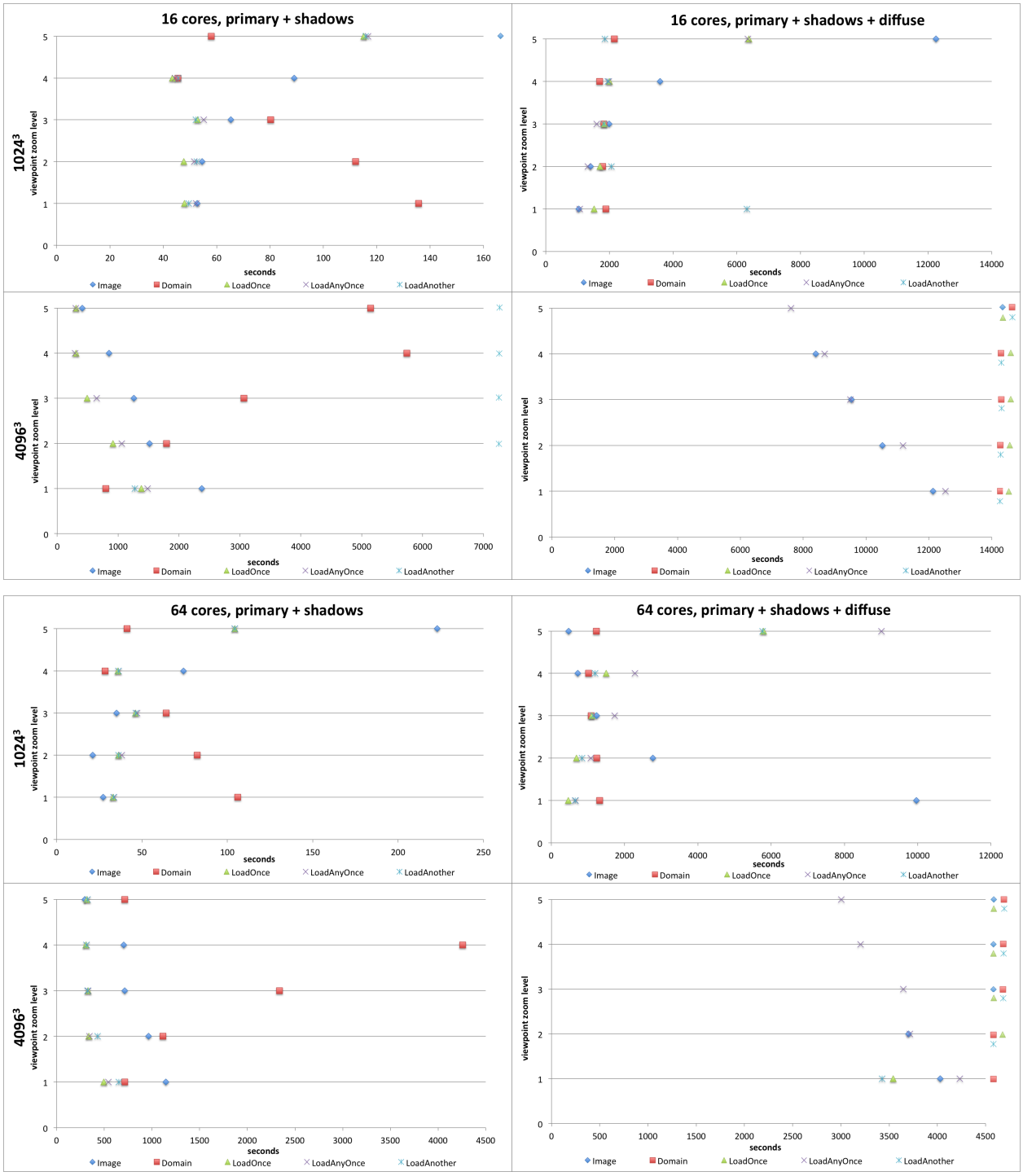


Fig. 9. Comparison of scheduling algorithms. Five factors are varied in the study; each of the eight scatter plots holds three of these factors constant and varies the remaining two. The factors held constant are concurrency, data size, and ray effects. Each factor has two options. The X-axes correspond to time, measured in seconds. The Y-Axis represents camera position. There are five camera positions and each horizontal line within a scatter plot is for one camera position. The bottommost line is the most zoomed out position, and the topmost line the most zoomed in. Each test is colored by the scheduling algorithm employed. Finally, tests that do not complete within four hours are rendered to the right of its scatter plot.

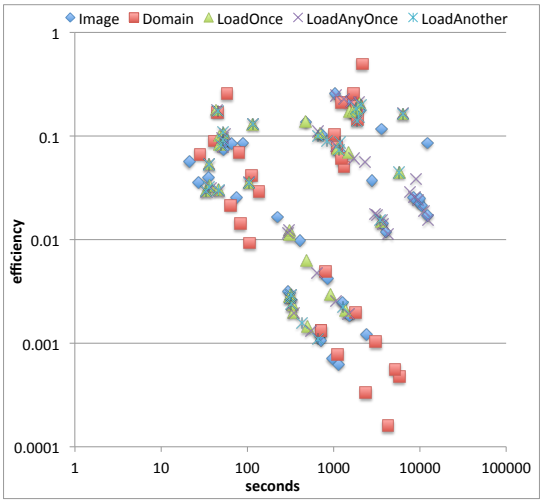


Fig. 10. Efficiency results. total run time (X-Axis) vs. efficiency (Y-Axis).

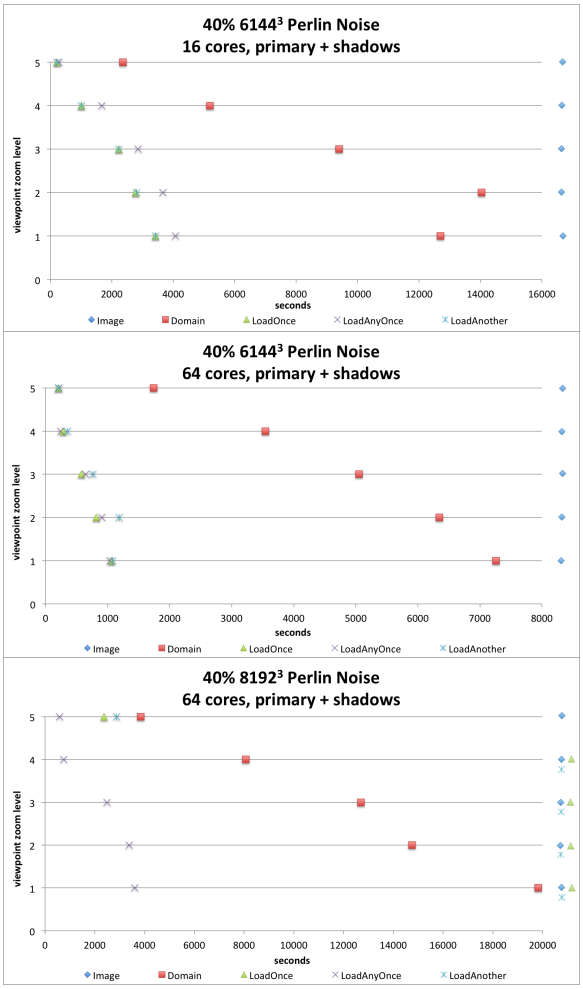


Fig. 11. Results from our larger data evaluation. The plots are arranged as in Figure 9.

A. Evaluating Larger Data

In this section, we extend the above study to larger data sets that stress the limits of the test machine. We test two larger versions of the Perlin noise set: at 6144^3 (4096 domains, 729 GB on disk, 557M tri at the 40% isosurface); and at 8192^3 (4096 domains, 1722 GB on disk, 992M tri at the 40% isosurface). Since these datasets stress both the capabilities and the runtime limits of TACC *Longhorn*, we limit our experiments to trace only primary rays plus shadows. We present the results of these runs in Figure 11. Note that for the 8192^3 case, we present only results on 64 cores, since only three of the twenty-five runs on sixteen cores completed. We note that the *LoadAnyOnce* schedule is again the only schedule to successfully complete every run and no *Image* run completes. While *LoadAnyOnce* suffers a bit of performance penalty at 16 cores due to swapping already-loaded domains among processors, the fewer overall loads at 64 cores eliminates this disadvantage.

B. Evaluating Platform Differences

We have also extended the study described in Section VI to run on TACC's *Stampede* machine. The goal here is to measure the effects of technological improvements, in particular, the degree to which a new supercomputer—with its increased disk speeds, faster communication times, and increased computational power—can accelerate the tests. Table IV shows the results as speedups of *Stampede* performance over *Longhorn* performance averaged over all tests for each algorithm. Our key finding is that the newer architecture does lead to speedups, but they do not qualitatively change execution times. We believe that each scheduling algorithm benefits for different reasons. Algorithms that perform many loads, such as *Image*, benefit from increased disk speed, while those that require significant parallel coordination, such as *Domain*, benefit from the improved network. Finally, the enhanced speedup for the *LoadAnyOnce* scheduler is misleading, because so many of the computationally-challenging tests—ones that could receive bigger speedups—only finished with this scheduling algorithm, and thus cannot be compared with the others. We would expect numbers from such comparisons to improve their average speedups. We leave further breakdown of this topic for future work.

TABLE IV
SPEEDUP OF STAMPEDE PERFORMANCE OVER LONGHORN PERFORMANCE.

Scheduling Algorithm	Average Speedup
Domain	1.26
Image	1.19
LoadOnce	1.27
LoadAnyOnce	1.43
LoadAnother	1.26
All Schedulers	1.29

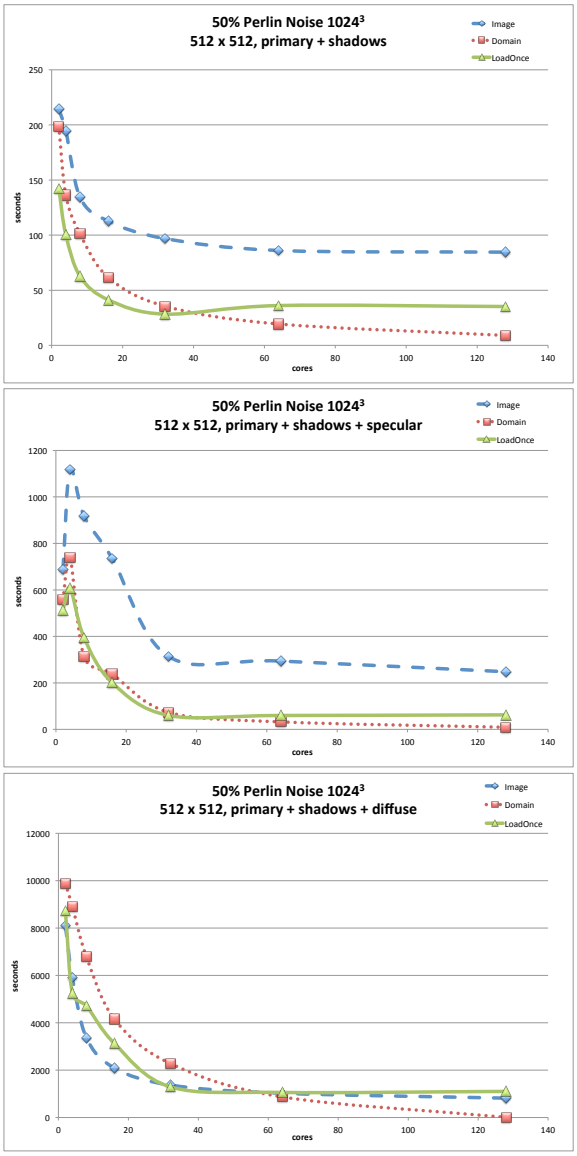


Fig. 12. Results from tests using three lighting models: shadows from two lights (top); shadows and specular reflections (middle); and shadows with 4×4 sampled diffuse reflections.

VII. Scaling Study

In this section, we present a detailed performance evaluation of our three original scheduling algorithms [3]—dynamic (*LoadOnce*), image decomposition (*Image*), and domain decomposition (*Domain*)—for rendering the Perlin noise dataset. We test performance along three axes: lighting model, geometry within each dataset, and overall dataset size. Through each axis we perform a strong scaling study by increasing the number of processes used to render the images.

We first test the performance of each schedule under increasingly complex lighting models. As described in Section V, our simplest lighting model uses only shadows from two light sources, which provides at most two secondary rays per intersection. We then add two-bounce specular reflections, which spawns three secondary rays (two shadow + one specular) per non-shadow intersection. Finally, we replace the specular reflections with 4×4 diffuse reflections, which provides eighteen rays (two shadow + sixteen diffuse) per non-shadow intersection. Diffuse rays are culled with a 10% chance of termination each bounce.

We next test the effect of geometry load. We have produced five sizes of Perlin noise volume from 256^3 to 4096^3 in increasing powers of two along each axis. These five volumes test the impact of data load time on each schedule. The number of domains in the volume also increases as the volumes grow larger. This increase will impact the number of processes needed to hold the volume resident; it is also likely to increase the number of domains that must be loaded during the render.

Last, by taking various isosurfaces of a volume of Perlin noise, we can create pseudo-random datasets with varying amounts of geometry that all form features similar to what might be found in dense scientific datasets. We can thus measure the effect of total non-zero data on schedule performance. We take five representative isosurfaces, shown in Figure 1.

In each test set, we also evaluate the strong scaling performance of each schedule by increasing the number of processes used in the render. We are thus able to observe the parallel efficiency enabled by each schedule.

In the rest of this section, we present a representative sample of results across these three dimensions. By holding two of the dimensions constant in each series, we can examine the effect of each dimension on each schedule.

A. Lighting Tests

For our lighting tests, we hold the other two dimensions constant: we use the 1024^3 dataset at the 50% isosurface level. We present the lighting test results in Figure 12. In each of these cases, we see that the *LoadOnce* schedule is best or competitive at small process counts. When the number of processes matches or exceeds the number of domains (64 for the 1024^3 case), the *Domain* schedule outperforms the others since it only loads each domain once. The *Image* schedule performs best when ray transfer costs are high compared to domain loads, since only the *Domain* and *LoadOnce* schedules incur ray transfer costs. Since the diffuse reflection model generates eighteen rays per intersection, and since there is sufficient geometry in the 50% case to cause many intersections, the ray transfer costs outweigh the cost of a domain in the 1024^3 dataset. Once there are sufficient processes to hold all domains, however, the *Domain* schedule again outperforms the others since it incurs no domain loading costs.

B. Geometry Tests

For our geometry tests, we use diffuse lighting, which is the most interesting result from Section VII-A, and we hold the data size constant at 1024^3. We present the results in a log-log chart in Figure 13. We see that the performance trend for the *Image* and *LoadOnce* schedules is similar across increasing geometric loads, while the *Domain* schedule converges past 64 processes. Similar to the results above, the *LoadOnce* schedule generally outperforms the other schedules at low process counts across all geometry loads, only losing to the *Image* schedule at the largest geometry loads (40% and 50%) and at middle process counts (8 and 16), where the image schedule

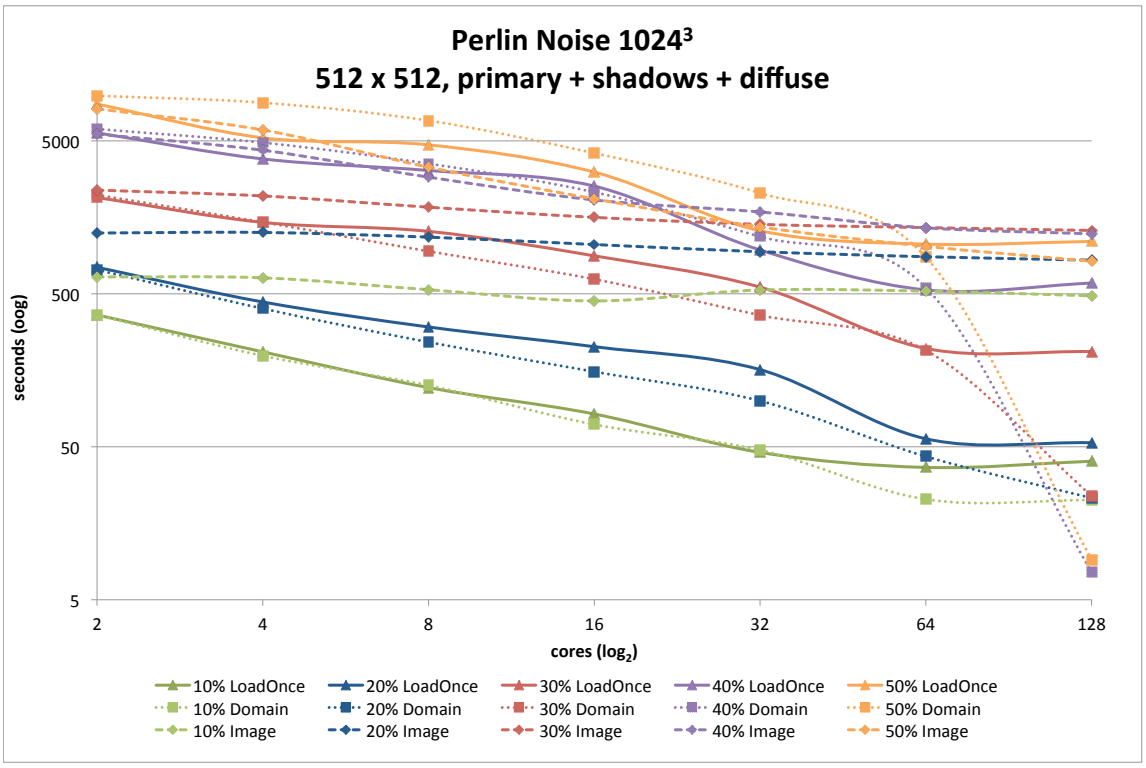


Fig. 13. Results from the isosurface tests on the 1024^3 dataset using 4×4 sampled diffuse reflections and shadows. The chart axes are $\log_2$-$\log_{10}$.

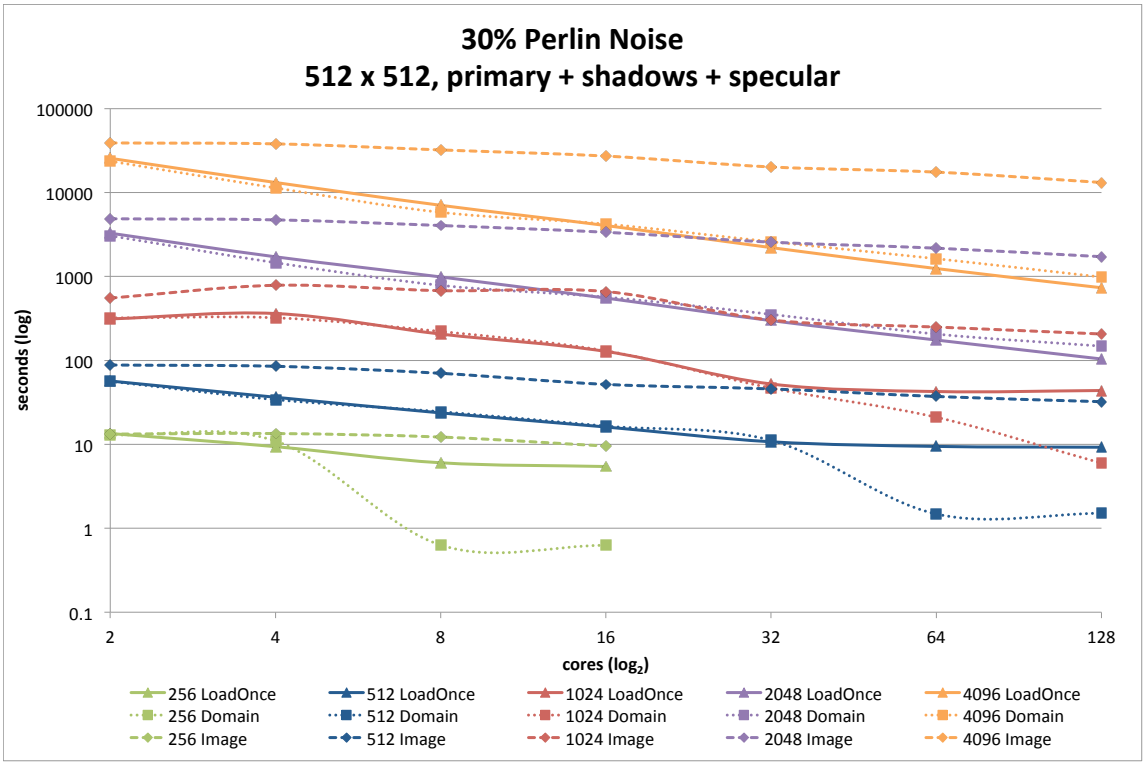


Fig. 14. Results from the data size tests with the 30% isosurface with specular reflections and shadows. The 256^3 results stop at sixteen cores since the 256^3 dataset only has eight domains.

benefits from parallel rendering while the other schedules must transfer rays as well as load domains. The *Domain* schedule again outperforms the other two schedules when there are sufficient processes to hold all domains, but when there are large geometry loads, it is the worst-performing schedule.

C. Data Size Tests

In the final test series we hold both the lighting model and the geometric complexity constant and vary the size of the dataset rendered. Due to the runtime limitations of *Longhorn*, we use the specular lighting model rather than the diffuse model from the previous tests. We present the results of these tests in Figure 14. We see that consistent trends for each schedule across data sizes. As the data size increases, the *Image* schedule is penalized more due to the increased load cost for each domain. The *LoadOnce* schedule slightly outperforms the *Domain* schedule until sixty-four processes. This matches or exceeds the total number of domains in the datasets up to 1024^3; however, for the datasets that have more domains (2048^3, 4096^3) the *LoadOnce* schedule still outperforms the *Domain* schedule.

D. Discussion

The results of the above experiments express the behavior of these schedulers across a wide range of conditions. We see that the *LoadOnce* schedule outperforms the *Image* and *Domain* schedules for cases where each process is likely to load many data domains, particularly when data load costs exceed ray communication costs. The *Image* scheduler performs best when ray communication costs exceed data load costs, since the *Image* scheduler never redistributes rays. The *Domain* scheduler performs best when the number of processors available matches or exceeds the number of data domains, since in that case each domain has a dedicated process and no data is reloaded.

VIII. Future Work and Conclusion

In this paper, we have presented and analyzed a family of dynamic scheduling algorithms for large-scale distributed memory ray tracing. This approach was designed for very large datasets that do not fit in the aggregate memory of a distributed memory supercomputer. Traditional ray tracing approaches often fail to render such datasets, and, as we have shown here and elsewhere [3], our schedulers can render datasets that would otherwise be too large to complete. Our more detailed analysis here shows that our dynamic scheduling policies are robust across many data sizes and rendering modes. Indeed, even on data sets that favor static scheduling, our schemes are competitive with the best known traditional static scheduling schemes. We also show that hardware advancements are not likely to bring about sufficient performance improvements to allow rendering of such datasets by traditional algorithms and that further development of efficient algorithms is needed.

There are many additional directions to explore in the space of dynamic schedulers. For example, a dynamic scheduler could speculatively load data based on anticipated ray travel, particularly for an animation sequence where rendering information from the previous frame is available, though care should be taken to keep scheduling costs low relative to data load and ray intersection work. We anticipate that enabling asynchronous ray communication by moving to a one-way MPI communication model will further increase the performance benefit of dynamic schedules over static schedules, though the communication patterns make this change decidedly non-trivial.

Acknowledgment

Thanks to Kelly Gaither, Karl Schulz, Keshav Pingali, Bill Mark and the anonymous reviewers for their helpful comments. We also thank Greg D. Abram for his Perlin noise data. This work was funded in part by National Science Foundation grants ACI-9984660, EIA-0303609, ACI-0313263, CCF-0546236, OCI-0622780, OCI-0726063, OCI-0906379, OCI-1134872 and ACI-1339863; an Intel Research Council grant; an IC^2 Institute fellowship; and the US Department of Energy Early Career Program.

References

[1] D. E. DeMarle, C. P. Gribble, S. Boulos, and S. G. Parker, "Memory Sharing for Interactive Ray Tracing on Clusters," *Parallel Computing*, vol. 31, no. 2, pp. 221–242, February 2005.

[2] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan, "Rendering Complex Scenes with Memory-Coherent Ray Tracing," *Computer Graphics (Proceedings of SIGGRAPH)*, vol. 31, no. Annual Conference Series, pp. 101–108, August 1997. [Online]. Available: citeseer.ist.psu.edu/pharr97rendering.html

[3] P. A. Navrátil, D. S. Fussell, C. Lin, and H. Childs, "Dynamic Scheduling for Large-Scale Distributed-Memory Ray Tracing," in *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, 2012, pp. 61–70.

[4] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P. P. Sloan, "Interactive Ray Tracing for Isosurface Rendering," in *Proceedings of IEEE Visualization*, 1998, pp. 233–238.

[5] J. Bigler, A. Stephens, and S. Parker, "Design for Parallel Interactive Ray Tracing Systems," in *Proceedings of Interactive Ray Tracing*, 2006, pp. 187–196.

[6] H. Childs, M. A. Duchaineau, and K.-L. Ma, "A Scalable, Hybrid Scheme for Volume Rendering Massive Data Sets," in *Eurographics Symposium on Parallel Graphics and Visualization*, 2006, pp. 153–162.

[7] T. Peterka, H. Yu, R. Ross, K.-L. Ma, and R. Latham, "End-to-End Study of Parallel Volume Rendering on the IBM Blue Gene/P," in *Proceedings of International Conference on Parallel Processing*, 2009, pp. 566–573.

[8] M. Howison, E.Wes Bethel, and H. Childs, "MPI-Hybrid Parallelism for Volume Rendering on Large, Multi-Core Systems," in *Eurographics Symposium on Parallel Graphics and Visualization*, 2010, pp. 1–10.

[9] M. Howison, E. W. Bethel, and H. Childs, "Hybrid Parallelism for Volume Rendering on Large-, Multi-, and Many-Core Systems," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 1, pp. 17–29, Jan. 2012.

[10] E. Reinhard and F. W. Jansen, "Rendering Large Scenes using Parallel Ray Tracing," in *Proceedings of Eurographics Workshop of Parallel Graphics and Visualization*, 1996, pp. 873–885.

[11] E. Reinhard, A. G. Chalmers, and F. W. Jansen, "Hybrid Scheduling for Parallel Rendering using Coherent Ray Tasks," in *Proceedings of IEEE Parallel Visualization and Graphics Symposium*, 1999, pp. 21–28.

[12] I. Wald, P. Slusallek, and C. Benthin, "Interactive Distributed Ray Tracing of Highly Complex Models," in *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, 2001, pp. 277–288.

[13] D. E. DeMarle, S. Parker, M. Hartner, C. Gribble, and C. Hansen, "Distributed Interactive Ray Tracing for Large Volume Visualization," in *Proceedings of the Symposium on Parallel and Large-Data Visualization and Graphics*, 2003, pp. 87–94.

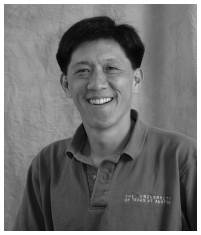
[14] T. Ize, C. Brownlee, and C. D. Hansen, "Real-Time Ray Tracer for Visualizing Massive Models on a Cluster," in *Eurographics Symposium on Parallel Graphics and Visualization*, 2011, pp. 61–69.

[15] C. Brownlee, T. Fogal, and C. Hansen, "Image-parallel Ray Tracing using OpenGL Interception," in *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2013, pp. 65–72.
[16] S. Parker, M. Parker, Y. Livnat, P. P. Sloan, C. Hansen, and P. Shirley, "Interactive Ray Tracing for Volume Visualization," *IEEE Transactions on Computer Graphics and Visualization*, vol. 5, no. 3, pp. 238–250, July-September 1999.
[17] A. Reshetov, A. Soupikov, and J. Hurley, "Multi-Level Ray Tracing Algorithm," *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, vol. 24, no. 3, pp. 1176–1185, 2005.
[18] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley, "State of the Art in Ray Tracing Animated Scenes," *Computer Graphics Forum*, vol. 28, no. 6, pp. 1691–1722, 2009.
[19] I. Wald, T. J. Purcell, J. Schmittler, C. Benthin, and P. Slusallek, "Realtime Ray Tracing and Its Use for Interactive Global Illumination," in *Proceedings of EUROGRAPHICS STAR — State of The Art Report*, 2003.
[20] I. Wald, A. Dietrich, and P. Slusallek, "An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models," in *Proceedings of the Eurographics Symposium on Rendering*, 2004, pp. 81–92.
[21] E. Gobbetti, F. Marton, and J. A. I. Guitián, "A Single-Pass GPU Ray Casting Framework for Interactive Out-of-Core Rendering of Massive Volumetric Datasets," *The Visual Computer*, vol. 24, no. 7–9, pp. 797–806, July 2008.
[22] J. Salmon and J. Goldsmith, "A Hypercube Ray-Tracer," in *Proceedings of the Third Conference on Hypercube Computers and Applications*, G. Fox, Ed., 1988, pp. 1194–1206.
[23] S. A. Green and D. J. Paddon, "A Highly Flexible Multiprocessor Solution for Ray Tracing," *The Visual Computer*, vol. 6, pp. 62–73, 1990.
[24] I. Wald, C. Benthin, and P. Slusallek, "Distributed Interactive Ray Tracing of Dynamic Scenes," in *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, 2003, pp. 77–85.
[25] D. E. DeMarle, C. P. Gribble, and S. G. Parker, "Memory-Savvy Distributed Interactive Ray Tracing," in *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, D. Bartz, B. Raffin, and H.-W. Shen, Eds., 2004, pp. 93–100.
[26] T. Kato, "'Kilauea' – Parallel Global Illumination Renderer," *Parallel Computing*, vol. 29, pp. 289–310, 2003.
[27] T. Kato and J. Saito, "'Kilauea' – Parallel Global Illumination Renderer," in *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization (EGPGV)*, D. Bartz, X. Pueyo, and E. Reinhard, Eds., 2002, pp. 7–16.
[28] F. Dachille IX and A. Kaufman, "GI-Cube: An Architecture for Volumetric Global Illumination and Rendering," in *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, August 2000, pp. 119–128. [Online]. Available: citeseer.ist.psu.edu/455729.html
[29] T. Aila and S. Laine, "Understanding the Efficiency of Ray Traversal on GPUs," in *Proceedings of High Performance Graphics*, 2009, pp. 145–149.
[30] 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*, vol. 28, no. 2, pp. 385–396, 2009.
[31] T. Aila and T. Karras, "Architecture Considerations for Tracing Incoherent Rays," in *Proceedings of High Performance Graphics*, M. Doggett, S. Laine, and W. Hunt, Eds., 2010, pp. 113–122.
[32] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms," *ACM Computing Surveys*, vol. 6, no. 1, pp. 1–55, March 1974. [Online]. Available: doi.acm.org/10.1145/356625.356626
[33] S. A. Green and D. J. Paddon, "Exploiting Coherence for Multiprocessor Ray Tracing," *IEEE Computer Graphics and Applications*, vol. 9, no. 11, pp. 12–26, 1989.
[34] I. Wald, P. Slusallek, C. Benthin, and M. Wagner, "Interactive Rendering with Coherent Ray Tracing," *Computer Graphics Forum (Proceedings of Eurographics 2001)*, vol. 20, no. 3, pp. 153–164, 2001. [Online]. Available: citeseer.ist.psu.edu/wald01interactive.html
[35] D. Voorhies, *Graphics Gems 2*. Academic Press, 1991, ch. Space-Filling Curves and a Measure of Coherence, pp. 26–30, 485–486.
[36] S. Boulos, I. Wald, and C. Benthin, "Adaptive Ray Packet Reordering," in *Proceedings of Interactive Ray Tracing*, 2008, pp. 131–138.
[37] P. A. Navrátil, D. S. Fussell, C. Lin, and W. R. Mark, "Dynamic Ray Scheduling to Improve Ray Coherence and Bandwidth Utilization," in *Proceedings of Interactive Ray Tracing*, 2007, pp. 95–104.
[38] J. Steinhurst, G. Coombe, and A. Lastra, "Reordering for Cache Conscious Photon Mapping," in *Proceedings of Graphics Interface*, 2005, pp. 97–104.
[39] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P. P. Sloan, "Interactive Ray Tracing," in *Proceedings of Interactive 3D Graphics*, 1999, pp. 233–238.
[40] M. Dippé and J. Swensen, "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis," *Computer Graphics (Proceedings of SIGGRAPH)*, vol. 18, no. 3, pp. 149–158, July 1984.
[41] H. Kobayashi, S. Nishimura, H. Kubota, T. Nakamura, and Y. Shigei, "Load Balancing Strategies for a Parallel Ray-Tracing System Based on Constant Subdivision," *The Visual Computer*, vol. 4, pp. 197–209, 1988.
[42] H. Childs, E. Brugger, B. Whitlock, and 19 others, "VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data," in *Proceedings of SciDAC 2011*, July 2011, http://press.mcs.anl.gov/scidac2011.
[43] K. Perlin, "An Image Synthesizer," *Computer Graphics*, vol. 19, no. 3, pp. 287–296, July 1985.

Paul A. Navrátil received the BA (1999), BS (1999), MS (2006) and PhD (2010) degrees from the University of Texas at Austin. He holds dual roles as Manager of the Scalable Visualization Technologies group at the Texas Advanced Computing Center and Adjunct Professor in the Division of Statistics and Scientific Computation, both at the University of Texas at Austin. His research interests include efficient algorithms for large-scale parallel visualization and data analysis and innovative design for large-scale visualization systems.

Hank Childs holds dual roles as an assistant professor of Computer Science at the University of Oregon and a Computer Systems Engineer at Lawrence Berkeley National Laboratory. He received his Ph.D. from the University of California at Davis in 2006. Childs' research interests are in scientific visualization, high performance computing, and their intersection, and he is the co-editor of the book "High Performance Visualization."

Donald S. Fussell is Trammell Crow Regents' Professor in the Computer Science Department, Director of the Laboratory for Graphics and Parallel Systems, a member Computer Engineering Research Center of the Electrical and Computer Engineering Department, and a member of the Institute for Computational Engineering and Sciences at the University of Texas at Austin. His research interests include Computer Graphics, Computer Games, Computer Architecture, and Computer Systems Design.



Calvin Lin received the BSE from Princeton University in 1985 and the PhD in Computer Science from the University of Washington in 1992. He joined the faculty at The University of Texas in 1996, where he is now a University Distinguished Teaching Professor and the Director of the Turing Scholars Honors Program. His research interests include compilers, parallel computing, and computer architecture, and he is a co-author of the textbook, "Principles of Parallel Programming."