

# Dynamic Scheduling for Large-Scale Distributed-Memory Ray Tracing

Paul A. Navratil<sup>1</sup>, Hank Childs<sup>2</sup>, Donald S. Fussell<sup>1</sup> and Calvin Lin<sup>1</sup>

1 = University of Texas at Austin

2 = Lawrence Berkeley National Laboratory

**DISCLAIMER:** This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.

**Acknowledgments:** Thanks to Kelly Gaither, Karl Schulz, Keshav Pengali, Bill Mark and the anonymous reviewers for their helpful comments. We also thank Ilian T. Iliev and Paul Shapiro for the n-body particle data. This work was funded in part by National Science Foundation grants ACI- 9984660, EIA-0303609, ACI-0313263, CCF-0546236, OCI-0622780, OCI-0726063, and OCI-0906379; an Intel Research Council grant; and an IC<sub>2</sub> Institute fellowship. This work was supported by the Director, Office of Science, Office and Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

# Dynamic Scheduling for Large-Scale Distributed-Memory Ray Tracing

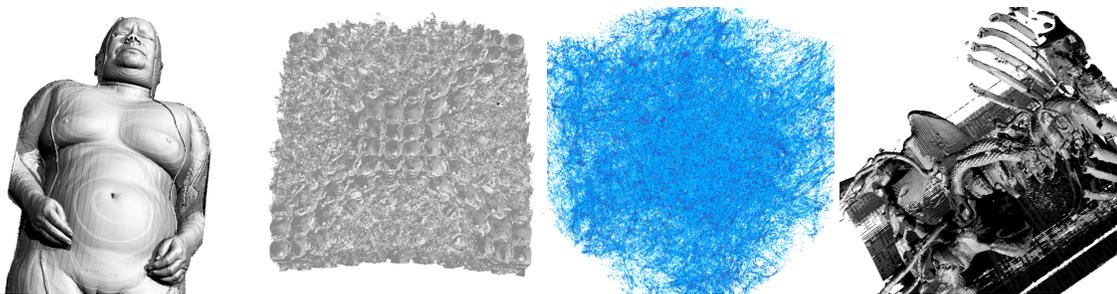


Fig. 1. The datasets used in this paper. From left to right: the Visible Female, Richtmyer-Meshkov instability, particle density of an n-body cosmological simulation, Viatronix abdomen CT scan.

**Abstract**— Ray tracing is an attractive technique for visualizing scientific data because it can produce high quality images that faithfully represent physically-based phenomena. Its embarrassingly parallel reputation makes it a natural candidate for visualizing large data sets on distributed memory clusters, especially for machines without specialized graphics hardware. Unfortunately, the traditional recursive ray tracing algorithm is exceptionally memory inefficient on large data, especially when using a shading model that generates incoherent secondary rays. As visualization moves through the petascale to the exascale, disk and memory efficiency will become increasingly important for performance, and traditional methods are inadequate.

This paper presents a dynamic ray scheduling algorithm that effectively manages both ray state and data accesses. Our algorithm can render datasets that are larger than aggregate system memory, which existing statically scheduled ray tracers cannot render. For example, using 1024 cores of a supercomputing cluster, our unoptimized algorithm ray traces a 650GB dataset from an N-Body simulation with shadows and reflections, at about 1100 seconds per frame. For smaller problems that fit in aggregate memory, but are larger than typical shared memory, our algorithm is competitive with the best static scheduling algorithm.

---

## 1 INTRODUCTION

### 2 INTRODUCTION

Ray tracing is an important method for creating high quality images, because it offers a broad range of physically-based shading options, including shadows, reflections and interactions with participating media. Such effects, which are often found in movies and photo-realistic images, are increasingly used for scientific visualization, because they help viewers better understand spatial relationships in data [15]. Of course, any method, including ray tracing, that must realistically simulate global illumination will be much more computationally expensive than the simpler methods that are traditionally used for real time graphics on GPUs.

To speed up ray tracing, it is tempting to process rays in parallel. This parallelism, however, is only efficient when it makes effective use of the memory system. Ideally, the parallel tasks would operate on the same memory-resident data, because if the working set grows larger than available memory, contention and thrashing significantly reduce overall performance. Unfortunately, traditional recursive ray tracers cannot bound the size of their working sets because after the first generation of rays (“primary” or “camera” rays), divergent rays in subsequent ray generations typically travel through different regions of space accessing a large amount of object data, leading to dramatically increased working sets.

For small data sets, the performance impact of incoherent secondary rays can be masked by a shared memory system, where the working sets of all parallel tasks can be kept resident. However, most large scientific simulations are now run on distributed memory supercomputers [36] and produce ever-increasing amounts of data per timestep [18]. Such data are typically too large to be relocated for analysis and are typically too large for a single shared-memory resource. As a result, the machine used to produce the data must be the same machine to ray trace the data. Further, 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 [9]. As visualization moves to the petascale and beyond, disk- and memory-efficient algorithms will be essential for good performance.

To date, most parallel ray tracing research has been limited to either ray tracing on shared-memory machines [26, 3] or to ray casting (tracing only first-generation rays) on distributed memory architectures [7, 28, 16]. Recent work on distributed-memory ray tracers [31, 30, 41, 11, 9, 17] has demonstrated only modest scaling and has been hampered by various system limitations, including limited interconnect bandwidth and limited disk I/O bandwidth.

In the serial realm, Pharr et al. [29] reformulate the ray tracing solution to allow more flexible scheduling of ray-object intersection calculations. This formulation groups rays and data into coherent work units, known as ray queues, which present a trade-off: They increase locality (ray coherence) at the cost of increased memory state. In the parallel realm, the tradeoffs are much more complex because of the additional need to consider load balance. In addition, Pharr et al.’s use of disk to cache excess ray state 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.

This paper extends this idea of flexible scheduling to support parallel ray tracing. The result is a novel approach to distributed memory ray tracing that uses dynamic ray scheduling to improve memory locality while maintaining load balance. The basic idea is to reorder ray traversal and intersection calculations based on the data that are resident on each processor. By building locally coherent work units

of rays and data, our ray tracer has the flexibility to schedule these work units across the parallel environment to achieve better overall system performance. We show that dynamic scheduling of rays and data can improve performance for large datasets where disk I/O limits performance. Our algorithm is able to ray trace, with shadows and reflections, a 650GB n-body dataset on a 1024 node cluster, which a statically scheduled ray tracer could not render at all. For a smaller dataset that a static scheduler can complete, our dynamic scheduler reduces data loads by  $10\times$  to  $48\times$ . Our scheduler also exhibits better performance than static strategies for volumetric ray casting

The remainder of this paper proceeds as follows. We discuss related work in Section 3. We present our approach in Section 4. We describe our testing methodology in Section 5, and we present our results in Section 6. We then conclude with a discussion of future work.

### 3 RELATED WORK

In this section, we place our approach in the context of prior work and other approaches to large-scale ray tracing.

#### 3.1 Improving Memory Access Coherence

The importance of using data coherence to improve rendering performance has been known for over thirty years [35]. However, it was only recently discovered that by reordering ray computations and by queuing rays with data in a spatially localized manner, the number of accesses to data on disk can be significantly reduced [29]. The same principle improves memory performance [34] and cache performance [24]. Each of these works demonstrates that the additional ray coherence is achieved at the cost of additional state to be maintained. Ray reordering alone can improve SIMD-instruction utilization [4], but it does not achieve the same level of spatial coherence as reordering and queuing together, particularly for incoherent secondary rays.

#### 3.2 Shared-Memory Ray Tracing

Most parallel ray tracers assume a shared address space architecture [26, 25, 32, 3, 39, 40]. While these systems achieve impressive rendering performance, the shared address space does not map to supercomputer clusters, and it tends to hide rather than expose load balance concerns from the programmer. Explicitly out-of-core ray tracers [38, 13] 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.

#### 3.3 Distributed-Memory Ray Tracing

In distributed memory, non-queueing ray tracers face a tradeoff between coherence—achieved by tracing ray groups that pass through contiguous pixels—and load balance—achieved by tracing disparate pixels in hopes of balancing the rendering work [33]. These systems typically optimize performance by relying on an expensive preprocessing step that improves data coherence, such as a low-resolution rendering pass to pre-load data on the processes [14], or an expensive pre-built acceleration structure to guide on-demand data loads [37, 41].

DeMarle et al. [9, 10, 11] 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. [17] update this approach using the Manta ray tracer [3] and modern hardware, but they experience similar memory and scaling limitations while retaining the expensive preprocessing step.

Reinhard et al. [30, 31] 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.

To balance load, the Kilauea system [19, 20] 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 [8] and on a single workstation with GPU acceleration [2, 5, 1], solutions which are not feasible for the large datasets produced by supercomputing clusters.

### 3.4 Large-Scale Direct Volume Ray Casting

Recent work in large-scale ray casting uses a fixed data decomposition to render images across hundreds-of-thousands of processes [7, 16, 28]. These approaches process the entire dataset in core using a domain decomposition, rendering each sub-domain simultaneously and then compositing the results into the final image. This approach, which is a form of speculative execution, is effective because there is a fixed and regular amount of work to perform in the absence of reflections. When reflections are included, however, this fixed data approach is prone to load imbalance. Further, the speculative rendering wastes work when generated sub-domain images are rejected by the final image composition pass.

## 4 ALGORITHM OVERVIEW

This section describes the specific challenges for distributed-memory ray scheduling, the baseline approaches we modeled, and our dynamic scheduling algorithm.

### 4.1 Scheduling Considerations

While serial ray scheduling implementations tradeoff increased locality with additional state, moving to a distributed parallel environment adds load balancing to the list of concerns. With this added variable, it is not clear how one can apply a serial scheduling algorithm in a parallel environment. A successful parallel schedule will strike a balance among locality, state size and load balance to provide both memory efficiency and high utilization. Because the characteristics of ray computations can change over the course of generating a single image, such as from tracing coherent camera rays to incoherent diffuse reflections, we expect that a dynamic schedule will provide improved performance compared to a static baseline. Pseudocode for the tested schedules can be found in Figures 3 – 6.

### 4.2 Static Ray Scheduling Baselines

To establish a performance baseline, we implement two static schedules that represent direct extensions of a Pharr-like approach to the parallel domain: an image-plane decomposition, where a subset of camera rays and their child rays are traced to completion on each processor; and a static domain decomposition, where domains are assigned to a particular process and rays are sent among processes as the rays move across domains.

```

ProcessQueue(queue)
{
    while (! queue.empty() ) {
        r = q.top();
        q.pop();

        PerformRayOperations(d, r, q);

        if (! RayFinished(r) ) Enqueue(queues, r);
        else ColorFramebuffer(r);
    }
}

```

Fig. 2. Pseudocode for ProcessQueue(), used in each schedule pseudocode (see Figures 3 – 6). PerformRayOperations() includes traversal, intersection, shading and spawning new rays.

```

ImagePlaneTrace()
{
  rays = GenerateRays();
  queues = EnqueueRays(rays);

  while (! queues.empty() ) {
    q = FindQueueWithMostRays(queues);
    d = LoadDomain(q.domain_id);
    ProcessQueue(q);
    queues.delete(q);
  }
  MergeFramebuffers();
}

```

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

**Static Image-Plane Decomposition** — rays are evenly divided among processes by contiguous image-plane decomposition, and data is loaded on each process as ray computation requires. At each scheduling step, each process selects the domain with the most local rays queued. This schedule optimizes for load balance, but it may exhibit poor locality. This strategy is similar to previous image coherence strategies [3, 9, 10, 11, 14, 26, 27, 37, 41]. This schedule also corresponds to the demand-driven component of the schedule used by Reinhard et al. [30, 31]; and it directly parallelizes a Pharr-like approach by using multiple serial instances run in parallel, where each seeded with a subset of camera rays. See the pseudocode in Figure 3.

```

DomainTrace() {
  rays = GenerateRays();
  queues = EnqueueRays(rays);

  last_d = NONE;
  done = FALSE;
  while (! done ) {
    # only has rays for its domains
    q = FindQueueWithMostRays(queues);
    if (q.domain_id != last_d) {
      d = LoadDomain(q.domain_id);
      last_d = q.domain_id;
    }
    ProcessQueue(q);
    queues.delete(q);

    # send rays to procs with
    # next domain
    SendRaysToNeighbors(queues);
    done = NoProcessHasRays();
  }
  MergeFramebuffers();
}

```

Fig. 4. Pseudocode for Static Domain Decomposition Schedule.

**Static Domain Decomposition** — the dataset is spatially subdivided, and these smaller domains are distributed among the available processes, typically in round-robin order. A process can be assigned multiple domains if there are more domains than processes, or it can be assigned no domain if there are more processes than domains. Domain data is loaded at first use, rather than prefetched. Rays are sent to the process that contains data needed for computation. At each scheduling step, each process selects the assigned domain with the most local rays queued. This schedule optimizes for locality, but it may exhibit poor load balance. This strategy is similar to previous domain decomposition strategies [8, 12, 21] and to the data parallel component of the scheduling strategy in Reinhard et al. [30, 31]. This decomposition also corresponds to the data distribution used in large-scale volume renderers [7, 16, 28]; and it directly parallelizes a Pharr-like approach

by using multiple serial instances run in parallel, where each is assigned a set of domains and rays are sent among the processes. See the pseudocode in Figure 4.

```

ScheduleNextRound(loaded_domain, queues) {
  foreach q in queues
    queue_info.insert(q.domain_id, q.size());
  SendQueueInfoToMaster(loaded_domain,
                        queue_info);

  if (isMaster()) {
    ReceiveQueueInfo(loaded_domains,
                    queue_infos);
    foreach p in ProcessCount()
      foreach q in queue_infos[p] {
        to_schedule.insert(q.domain_id);
        is_loaded.insert(loaded_domains[p], p);
      }

    foreach p in ProcessCount()
      if (!to_schedule
          .contains(loaded_domains[p])
          to_evict.insert(p);

    foreach domain_id in to_schedule
      if (is_loaded.contains(domain_id)) {
        proc_id = is_loaded[domain_id];
        schedule.insert(proc_id, domain_id);
      }
      else { to_assign.insert(domain_id); }

    while (!(to_assign.empty()
            || to_evict.empty())) {
      domain_id = to_assign.top();
      proc_id = to_evict.top();
      schedule.insert(proc_id, domain_id);
      to_assign.pop();
      to_evict.pop();
    }
    SendScheduleToAll(schedule);
  }
  ReceiveSchedule(schedule);
  return schedule[ MyProcessId() ];
}

```

Fig. 5. Pseudocode for ScheduleNextRound(), called in the dynamic schedule.

### 4.3 Dynamic Scheduling Algorithm

Our dynamic scheduling algorithm combines the benefits of the image-plane and domain decompositions to adapt to the changing characteristics of a ray tracing rendering. Our algorithm begins with an image-plane distribution of rays, with potentially duplicated domains across processes if many rays are concentrated in a particular domain. As the rendering progresses, the schedule shifts to a domain-decomposition style where rays are sent between processes and data remains resident. In contrast to the approaches above, our algorithm takes real-time feedback from the rendering to improve the scheduling for coherence and maintain high utilization.

**Dynamic Schedule** — rays are evenly divided across processes. After the initial ray distribution, each scheduling step sends rays to processes that already contain the domain data required for intersection. After swapping rays, each process operates on the domain with the most rays waiting for it. Processes that have domain data not needed by any rays may be assigned a new domain that is immediately needed by current rays. See the pseudocode in Figures 5 and 6.

The advantage of our dynamic schedule is its ability to reorder rays and defer computation until the required data has been loaded into memory. We expect that the dynamic schedule will see a reduced number of domain loads when compared against the static image-plane

```

DynamicRayWeightedTrace() {
  rays = GenerateRays();
  queues = EnqueueRays(rays);

  q = FindQueueWithMostRays(queues);
  d = LoadDomain(q.domain_id);
  last_d = q.domain_id;
  done = FALSE;
  while (! done ) {
    if (q.domain_id != last_d) {
      d = LoadDomain(q.domain_id);
      last_d = q.domain_id;
    }
    ProcessQueue(q);
    queues.delete(q);

    q = ScheduleNextRound(last_d, queues);
    done = NoProcessHasRays();
  }
  MergeFramebuffers();
}

```

Fig. 6. Pseudocode for Dynamic Ray-Weighted Schedule. `ProcessQueue()` is defined in Figure 2 and `ScheduleNextRound()` is defined in Figure 5.

decomposition and will achieve better load balance than the static domain decomposition.

## 5 METHODOLOGY

This section describes our experimental methodology, including the hardware platform, the datasets, and the rendering methods that we used to evaluate our scheduling strategy.

### 5.1 System Configuration

All experiments are run on *Longhorn*, a 2048 core, 256 node cluster hosted at the Texas Advanced Computing Center. 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 Infini-Band switch, and we use MVAPICH2 v1.4 for our MPI implementation. Our ray tracer is implemented within VisIt [6], a visualization tool designed to operate in parallel on large-scale data. We use the VisIt infrastructure to load data and to generate isosurfaces; we implemented all code related to ray tracing and ray scheduling. To focus on the effects of the schedules, 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 schedules most, since they have the highest degree of communication among processes.

### 5.2 Datasets

We perform a scaling study of our approach using datasets of various size and granularity from four domains: the visible female dataset from the National Library of Medicine, the Richtmyer-Meshkov instability dataset from Lawrence Livermore National Laboratory, a particle density field from an n-body cosmological simulation and a high-resolution CT scan of an abdominal cavity from Viatronix. The particular data sizes and decompositions on disk are presented in Table 1. Sample images of the data are given in Figure 1.

For each dataset, we extract an isosurface using VisIt’s VTK-based isosurfacing and internal BVH acceleration structure, and we then ray trace the returned geometry using two directional lights and, for the n-body particle data, two-bounce reflections. 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 Section 6. We do not

Table 1. Dataset Sizes and Decomposition

	Resolutions	
Visible Female	512 × 512 × 1734	
	24 domains 1.69 GB total size	
Richtmyer Meshkov	2048 × 2048 × 1920	
	960 domains 7.5 GB total size	
CT Scan	512 <sup>3</sup> 125 domains 4 GB total size	4096 <sup>3</sup> 4096 domains 256 GB total size
Cosmology	512 <sup>3</sup> 512 domains 8.7 GB total size	6144 <sup>3</sup> 4096 domains 650 GB total size

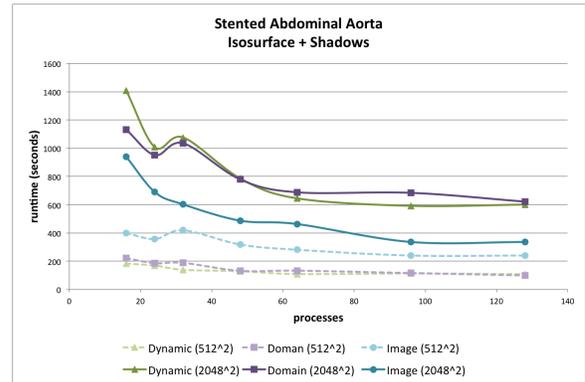


Fig. 7. **Ray Tracing of Abdominal CT scan** — Schedule performance for ray tracing on an isosurface of a 512<sup>3</sup> abdominal CT scan at image resolutions of 512<sup>2</sup> and 2048<sup>2</sup> (lower is better). The render includes shadow rays for two directional lights. Runtime is given in seconds per frame.

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. For the n-body particle density field, we also perform direct volume ray casting, as described by Levoy [22, 23].

## 6 RESULTS

This section evaluates our three scheduling strategies on the datasets described in Section 5. Our primary results are based on rendering 2048 × 2048 images of each dataset. In addition, we provide results that estimate the impact of a more optimized ray tracer by rendering 512 × 512 images of each dataset, which keeps data access costs about the same while reducing the ray computation cost. The difference between the 2048 × 2048 results and the 512 × 512 results approximates improved ray operation performance, as ray operations are a smaller fraction of total execution time.

We find that the image-plane schedule performs best for cases where ray operation costs (traversal and intersection) outweigh data I/O costs. Each process runs continuously, since there is no synchronization, and because I/O costs are small compared to ray operation costs, there is little penalty for redundant data loads. However, when I/O costs outweigh ray operation costs, typically the case for large datasets, the redundant data loads overcome the benefit of continuous parallel execution. As a result, the image-plane schedule performs significantly worse than the domain and ray-weighted schedules. Figure 7 demonstrates this effect on the small abdominal CT scan dataset.

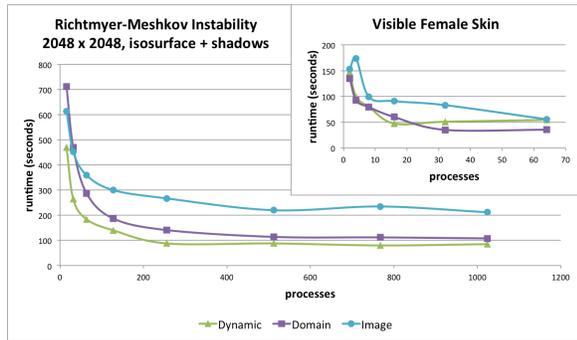


Fig. 8. **Richtmyer-Meshkov and Visible Female** — Schedule performance for the Richtmyer-Meshkov instability and the Visible Female datasets (lower is better). The render includes shadows from two directional lights; runtime is given in seconds per frame.

When few primary rays are cast, the I/O costs are a significant portion of overall runtime, and the image-plane schedule performs poorly. However, when many primary rays are cast, the I/O costs shrink relative to the ray operation costs, and the relative performance of the image-plane schedule improves. The static image-plane schedule can be as much as 78% faster than dynamic scheduling, since the I/O costs are amortized over many ray calculations. The results of Figure 7 are an outlier in two respects: First, our tracer is unoptimized, which artificially inflates the cost of ray calculations, and second, the abdominal CT scan isosurface is our smallest dataset, and our technique is targeted at much larger data. Nevertheless, this result suggests that if the data can reside completely in memory, an image-plane decomposition is a competitive technique. We see this effect exclusively on our smaller datasets: for the larger datasets discussed below, the I/O costs always dominate the total runtime, and the image-plane schedule performs poorly.

When rendering scenes larger than the memory available to a single process but that can still fit within aggregate memory, the domain and dynamic ray-weighted schedules perform significantly better than the image-plane schedule. In particular, the domain schedule performs best when the available aggregate memory can hold the dataset, so that each process receives one data domain. Since each domain is resident on a process, there is no I/O beyond the initial load. This effect can be seen in Figure 8. For the Richtmyer-Meshkov instability dataset, the ray-weighted schedule runs 124% faster than the domain schedule at 64 processes, but the domain schedule runs 64% faster at 1024 processes, where each process receives a single domain.

When rendering scenes that are larger than available aggregate memory, dynamic scheduling provides significant performance gains, as shown in Figure 9. When data load costs dominate execution time, which will be increasingly true as simulations and datasets increase, the dynamic ray-weighted schedule improves performance 8× to 14× over static schedules. When ray computation costs are large, the dynamic ray-weighted schedule still improves performance 3× to 5× over static schedules.

The performance gains of dynamic scheduling are primarily due to its ability to reduce data domain loads from disk. Figure 10 shows that ray-weighted dynamic scheduling can reduce data loads by 10× to 48×, regardless of ray computation load. We note that the image-plane schedule always touches more domains as the number of processes increases, since each process must load a domain if even one ray requires it. Under the domain schedule, a process will repeatedly swap among its assigned sub-domains. This swapping only stops when there are sufficient processes available to assign a single domain to most processes, as is the case with the Abdominal CT scan dataset. We also note some oscillation in the result trends, particularly for the domain schedule and for small processor counts. This occurs because the order in which domains are processed depends in part on the total

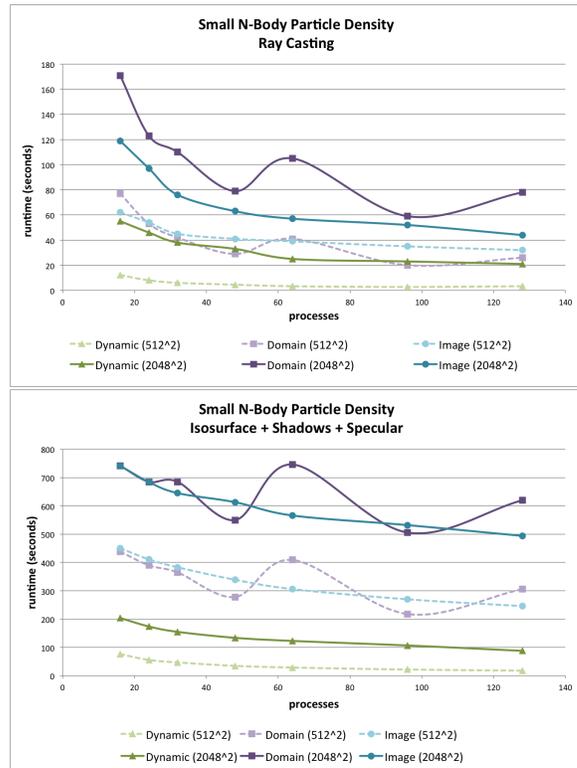


Fig. 9. **Direct Volume Ray Casting and Ray Tracing of N-Body Particle Density** — Schedule performance for direct volume ray casting and ray tracing on a  $512^3$  particle density field at image resolutions of  $512^2$  and  $2048^2$  (lower is better). The ray tracing includes two-bounce reflections and shadow rays for two directional lights. Runtime is given in seconds per frame.

number of processors available, which affects the order in which child rays are both generated and processed and results in different data access patterns; also, the round-robin domain assignment for the domain schedule causes each processor to receive a different set of domains, which can cause a particular process to load more data.

## 6.1 Scaling

To test the scalability of our approach, we ran the dynamic schedule on large versions of our datasets (exact details are in Table 1). The static schedules failed to complete on these larger datasets. The domain schedule fails when the ray queue becomes too large for a particular process to contain all queued rays along with the currently-loaded sub-domain. The image-plane schedule fails to finish within runtime limits on *Longhorn*. Figure 11 shows how the ray-weighted dynamic schedule performs on large datasets. We believe that the increased runtime for the largest number of processes tested is due to increased communication overhead while available parallelism from the data was exhausted.

Figure 12 compares strong scaling speedup for the three schedules. Dynamic ray-weighted scheduling exhibits monotonically increasing speedup until the scaling limit of the problem size is reached. The slope of the speedup line might be improved with ray calculation optimizations and interprocessor communication optimizations for exchanging rays among processes.

## 6.2 Effects of Decomposition on Disk

Because the I/O system has a significant impact on performance, we evaluate the performance of different approaches to decomposing the dataset on the disk. We organize the n-body datasets with two levels of subdivision, and we find that dividing the data into many spatially

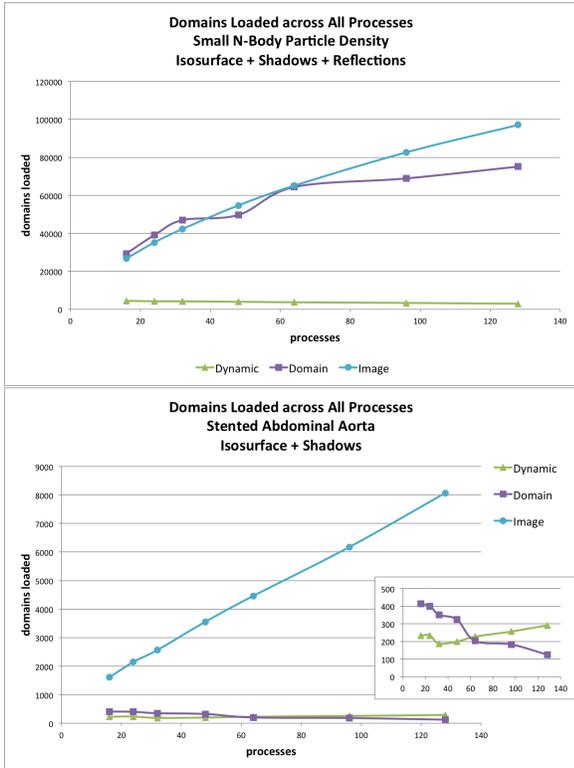


Fig. 10. **Spatial Domains Loaded from Disk** — Number of domains (spatial subdivisions) loaded from disk for each schedule for the cosmology and CT scan datasets (lower is better). Dynamic scheduling significantly reduces the number of domains loaded from disk, which is the primary factor for the performance gain of our approach.

distinct sub-domains can improve scheduling performance. If there are too few sub-domains, the flexibility of the scheduler is limited, but if there are too many sub-domains, the data become fragmented, which limits the number of rays queued at each domain. Empirically, a good balance occurs when the number of sub-domains is several times the number of processes used to render them. The execution times presented in Table 2 are for direct volume ray casting.

## 7 FUTURE WORK AND CONCLUSION

In this paper, we have presented a dynamic scheduling approach to large-scale distributed memory ray tracing. Our ray-weighted dynamic schedule is robust across many data sizes and rendering modes, and it provides an order of magnitude speedup over static scheduling methods when data access costs dominate the execution time. In addition, our dynamic schedule can render datasets that cannot be rendered by a static schedule. Our ray tracer has not been thoroughly optimized, and we have argued that as ray calculation costs are reduced through optimization, the gap between dynamic and static schedules will further increase.

We have just begun to explore the space of possible dynamic schedules. Further work is warranted to identify schedules that achieve particular system goals. In particular, it may be possible to schedule a sub-domain across several available processes, though this may increase both I/O and communication costs. A dynamic schedule could also speculatively load data based on anticipated ray travel, particularly for an animation sequence where rendering information from the previous frame is available. We anticipate that moving to a one-way communication model will further increase the performance benefit of dynamic schedules over static schedules.

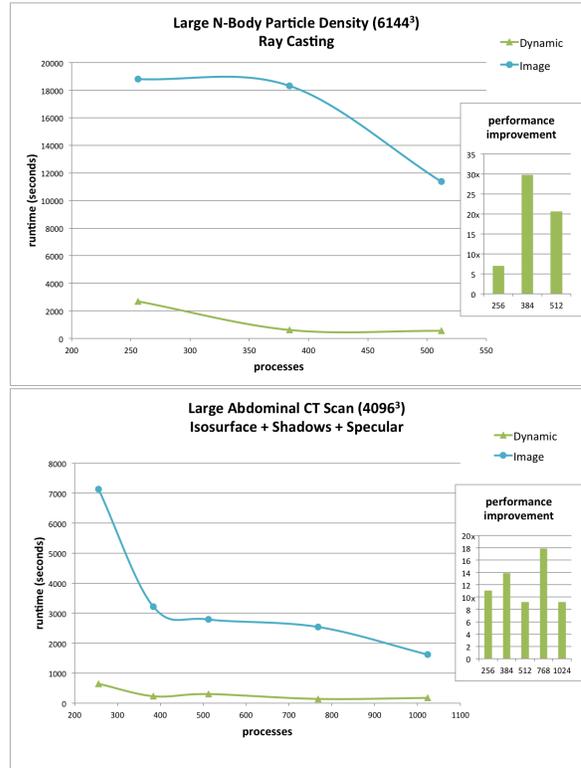


Fig. 11. **Dynamic Schedules for Large Data** — Dynamic schedule performance for direct volume ray casting of a 6144<sup>3</sup> n-body particle density field and for ray tracing of an isosurface extracted from a 4096<sup>3</sup> abdominal CT scan (lower is better). The isosurface render includes shadow rays for two directional lights and two-bounce specular reflections. Runtime is given in seconds per frame.

## ACKNOWLEDGMENTS

Thanks to Kelly Gaither, Karl Schulz, Keshav Pengali, Bill Mark and the anonymous reviewers for their helpful comments. We also thank Ilian T. Iliev and Paul Shapiro for the n-body particle data. This work was funded in part by National Science Foundation grants ACI-9984660, EIA-0303609, ACI-0313263, CCF-0546236, OCI-0622780, OCI-0726063, and OCI-0906379; an Intel Research Council grant; and an IC<sup>2</sup> Institute fellowship.

## REFERENCES

- [1] T. Aila and T. Karras. Architecture Considerations for Tracing Incoherent Rays. In M. Doggett, S. Laine, and W. Hunt, editors, *Proceedings of High Performance Graphics*, 2010.
- [2] T. Aila and S. Laine. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of High Performance Graphics*, 2009.
- [3] J. Bigler, A. Stephens, and S. Parker. Design for Parallel Interactive Ray Tracing Systems. In *Proceedings of Interactive Ray Tracing*, 2006.
- [4] S. Boulos, I. Wald, and C. Benthin. Adaptive Ray Packet Reordering. In *Proceedings of Interactive Ray Tracing*, 2008.
- [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. In P. Dutré and M. Stamminger, editors, *Proceedings of Eurographics*, 2009.
- [6] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, K. Bonnell, M. Miller, G. H. Weber, C. Harrison, D. Pugmire, T. Fogal, C. Garth, A. Sanderson, E. W. Bethel, M. Durant, D. Camp, J. M. Favre, O. Rübel, P. Navrátil, M. Wheeler, P. Selby, and F. Vivodtzev. 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>.

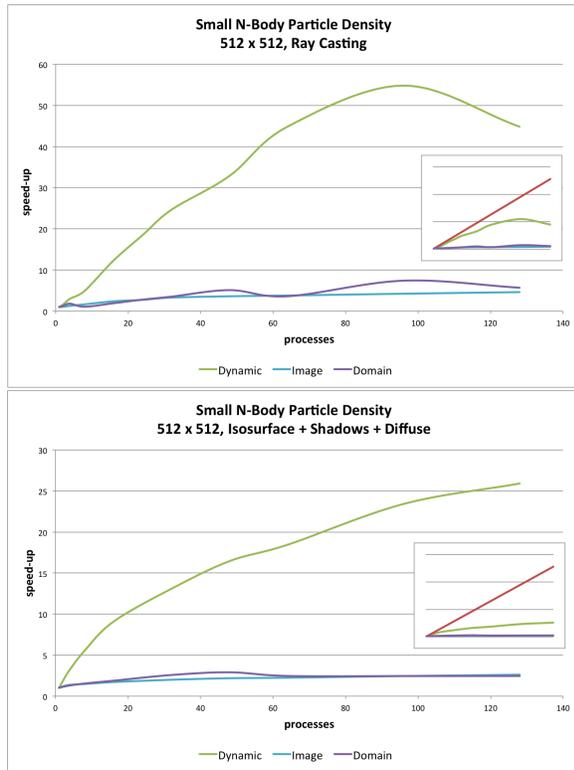


Fig. 12. **Schedule Speedup** — Performance of image-plane schedule, domain schedule and dynamic schedule for direct volume ray casting and ray tracing an isosurface of a  $512^3$  n-body particle density field (higher is better). The isosurface render includes shadow rays for two directional lights and  $16\times$  sampled, two-bounce diffuse reflections.

[7] 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*, pages 153–162, 2006.

[8] F. Dacheux and A. Kaufman. GI-Cube: An Architecture for Volumetric Global Illumination and Rendering. In *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 119–128, August 2000.

[9] D. E. DeMarle, C. P. Gribble, S. Boulos, and S. G. Parker. Memory Sharing for Interactive Ray Tracing on Clusters. *Parallel Computing*, 31(2):221–242, February 2005.

[10] D. E. DeMarle, C. P. Gribble, and S. G. Parker. Memory-Savvy Distributed Interactive Ray Tracing. In D. Bartz, B. Raffin, and H.-W. Shen, editors, *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, 2004.

[11] 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.

[12] M. Dippé and J. Swensen. An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis. *Computer Graphics (Proceedings of SIGGRAPH 1984)*, 18(3):149–158, July 1984.

[13] 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*, 24(7–9):797–806, July 2008.

[14] S. A. Green and D. J. Paddon. A Highly Flexible Multiprocessor Solution for Ray Tracing. *The Visual Computer*, 6:62–73, 1990.

[15] C. P. Gribble and S. G. Parker. Enhancing Interactive Particle Visualization with Advanced Shading Models. In *Proceedings of the 3rd Symposium on Applied Perception in Graphics and Visualization*, pages 111–118, 2006.

[16] 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.

Table 2. **Effects of Disk Decomposition on Execution Time** — Effect of the number of data files on scheduling performance. Cases marked 'OOM' exceeded memory limits; the cases marked 'DNF' did not finish within queue time limits.

512 <sup>3</sup> N-Body Particle Density 128 processes			
sub-domains	n=8	64	512
MB per sub-domain	1100	130	17
Dynamic	151	54	21
Domain	186	86	78
Image-Plane	92	56	44

6144 <sup>3</sup> N-Body Particle Density 256 processes			
sub-domains	n=512	4096	32768
MB per sub-domain	3400	436	55
Dynamic	2881	2164	6340
Domain	OOM	OOM	OOM
Image-Plane	DNF	DNF	DNF

[17] 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.

[18] C. Johnson and R. Ross, editors. *Visualization and Knowledge Discovery: Report from the DOE/ASCR Workshop on Visual Analysis and Data Exploration at Extreme Scale*. United States Department of Energy, October 2007.

[19] T. Kato. "Kilauea" – Parallel Global Illumination Renderer. *Parallel Computing*, 29:289–310, 2003.

[20] T. Kato and J. Saito. "Kilauea" – Parallel Global Illumination Renderer. In D. Bartz, X. Pueyo, and E. Reinhard, editors, *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization (EGPGV)*, pages 7–16, 2002.

[21] 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*, 4:197–209, 1988.

[22] M. Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[23] M. Levoy. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[24] 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.

[25] 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*, 5(3):238–250, July–September 1999.

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

[27] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P. P. Sloan. Interactive Ray Tracing. In *Proceedings of Interactive 3D Graphics*, 1999.

[28] 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*, pages 566–573, 2009.

[29] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering Complex Scenes with Memory-Coherent Ray Tracing. *Computer Graphics (Proceedings of SIGGRAPH)*, 31(Annual Conference Series):101–108, August 1997.

[30] 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.

[31] E. Reinhard and F. W. Jansen. Rendering Large Scenes using Parallel Ray Tracing. In *Proceedings of Eurographics Workshop of Parallel Graphics and Visualization*, 1996.

- [32] A. Reshetov, A. Soupikov, and J. Hurley. Multi-Level Ray Tracing Algorithm. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 24(3):1176–1185, 2005.
- [33] J. Salmon and J. Goldsmith. A Hypercube Ray-Tracer. In G. Fox, editor, *Proceedings of the Third Conference on Hypercube Computers and Applications*, pages 1194–1206, 1988.
- [34] J. Steinhurst, G. Coombe, and A. Lastra. Reordering for Cache Conscious Photon Mapping. In *Proceedings of Graphics Interface*, pages 97–104, 2005.
- [35] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A Characterization of Ten Hidden-Surface Algorithms. *ACM Computing Surveys*, 6(1):1–55, March 1974.
- [36] TOP500.org. Architecture Share for 11/2011, November 2011.
- [37] 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.
- [38] 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.
- [39] 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. In *Proceedings of EUROGRAPHICS STAR — State of The Art Report*, 2007.
- [40] 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.
- [41] I. Wald, P. Slusallek, and C. Benthin. Interactive Distributed Ray Tracing of Highly Complex Models. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, pages 277–288, 2001.