Hybrid Parallelism for Volume Rendering on Large, Multi- and Many-core Systems

Mark Howison, E. Wes Bethel, *Member, IEEE,* and Hank Childs

(Invited Paper)

Abstract—With the computing industry trending towards multi- and many-core processors, we study how a standard visualization algorithm, ray-casting volume rendering, can benefit from a hybrid parallelism approach. Hybrid parallelism provides the best of both worlds: using distributed-memory parallelism across a large numbers of nodes increases available FLOPs and memory, while exploiting shared-memory parallelism among the cores within each node ensures that each node performs its portion of the larger calculation as efficiently as possible. We demonstrate results from weak and strong scaling studies, at levels of concurrency ranging up to 216,000, and with datasets as large as 12.2 trillion cells. The greatest benefit from hybrid parallelism lies in the communication portion of the algorithm, the dominant cost at higher levels of concurrency. We show that reducing the number of participants with a hybrid approach significantly improves performance.

Index Terms—Volume visualization, parallel processing

1 Introduction

Many in the HPC community have expressed concern that parallel programming languages, models, and execution frameworks that have worked well to-date on single-core massively parallel systems may "face diminishing returns" as the number of computing cores on a chip increase [1]. In this context, we explore the performance and scalability of a common visualization algorithm – ray-casting volume rendering – implemented with different parallel programming models and run on both a large supercomputer comprised of six-core CPUs and a cluster with many-core GPUs. We compare a traditional distributed-memory implementation based solely on message-passing against a "hybrid" implementation, which uses a blend of message-passing (inter-chip) and shared-memory (intra-CPU/intra-GPU) parallelism. The thesis we wish to test is that there are opportunities in the hybrid-memory implementation for performance and scalability gains that result from using shared-memory parallelism among cores within a chip.

- *M. Howison is with the Center for Computation and Visualization, Brown University, 94 Waterman Street, Providence, RI 02912, USA, and the Computational Research Division, Lawrence Berkeley National Laboratory, One Cyclotron Road, Berkeley, CA 94720, USA. Email: mhowison@brown.edu*
- *E.W. Bethel and H. Childs are with the Computational Research Division, Lawrence Berkeley National Laboratory, One Cyclotron Road, Berkeley, CA 94720, USA Email: {mhowison,ewbethel,hchilds}@lbl.gov*

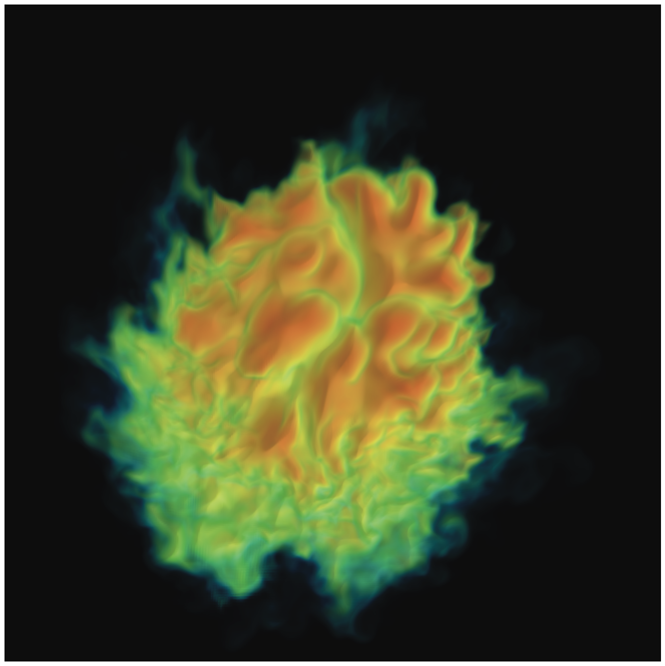

Fig. 1. This 4608^2 image of a combustion simulation result was rendered by our MPI+pthreads implementation running on 216,000 cores of the JaguarPF supercomputer.

Over the years, there has been a consistent and well-documented concern that the overall runtime of large-data visualization algorithms is dominated by I/O costs (e.g., [2]–[4]). During our experiments, we observed results consistent with previous work: there is a significant cost associated with scientific data I/O. In this study, however, we focus exclusively on the performance and scalability of the ray-casting volume rendering algorithm, not on parallel I/O performance. This approach is valid for many visualization use cases, such as creating multiple images from a single dataset that fits entirely within the memory footprint of a large system, or creating one or more images of data that is already resident in memory, as in the case of *in-situ* visualization.

Our findings (Section 5) show that there is indeed opportunity for performance gains when using hybrid-parallelism for raycasting volume rendering across a wide range of concurrency levels. The hybrid-memory implementation runs faster, requires less memory and, for this particular algorithm and set of implementation choices (Section 3), requires less communication bandwidth than the distributed-memory implementation.

This paper is an extension of our previous work [5], and we have added the following new components: (1) a weak scaling study to better understand performance as we increase problem size along with processor count; (2) a new platform and implementation – a hybrid/CUDA implementation run on a distributed memory GPU cluster – along with both strong and weak scaling studies on that platform to better understand how the change in ratio of cores to nodes impacts performance and scalability; (3) a study to determine the optimal image tile size and shape, based upon the principles of autotuning, for both the CPU and GPU platforms. For the sake of completeness, we reuse many of the figures and concepts from our earlier work.

2 Background and Previous Work

2.1 Parallel Volume Rendering

Volume rendering is a common technique for displaying 2D projections of 3D sampled data [6], [7] and is computationally, memory, and data I/O intensive. In the quest towards interactivity, as well as to address the challenges posed by growing data size and complexity, there has been a great deal of work over the years in the space of parallel volume visualization (see Kaufman and Mueller [8] for an overview). The focus of our work here is on a hybrid-memory implementation (Section 2.2) at extreme concurrencies to take advantage of multi- and many-core processor architectures.

Our hybrid-memory implementation makes use of a design pattern common to many parallel volume rendering applications that use a mixture of both object- and pixel-level parallelism [9]–[12]. The design employs an object-order partitioning to distribute source data blocks to processors where they are rendered using ray casting [6], [7], [13], [14]. Within a processor, we then use an image-space decomposition, similar to Nieh and Levoy [15], to allow multiple rendering threads to cooperatively generate partial images that are later combined via compositing into a final image [6], [7], [14].

This design approach, which uses a blend of object- and pixel-level parallelism, has proven successful in achieving scalability and tackling large data sizes. The TREX system [3] is a parallel volume rendering algorithm on a shared-memory platform that uses object-parallel data domain decomposition and texture-based, hardware-accelerated rendering followed by a parallel, software-based composition phase with image-space partitioning. The design choices for which part of the SGI Origin to use for different portions of the algorithm reflect a desire to achieve optimal performance at each algorithmic stage and to minimize inter-stage communication costs. Müller et al. [16] implemented a raycasting volume renderer with object-order partitioning on a small 8-node GPU cluster using programmable shaders. This system sustained frame rates in the single digits for datasets as large as 1260^3 with an image size of 1024×768. More recently, Moloney et al. [17] implement a GLSL texture-based volume renderer that runs on a 32-node GPU system. This work takes advantage of the sort-first architecture to accelerate certain types of rendering, like occlusion culling. Childs et al. [18] present a parallel volume rendering scheme for massive datasets (with one hundred million unstructured elements and a 3000^3 rectilinear data set). Their approach parallelizes over both input data elements and output pixels, and is demonstrated to scale well on up to 400 processors.

Peterka et al. [4] run a parallel volume rendering algorithm at massive concurrency, rendering 4480^3 data sizes with 32,000 cores on an IBM BG/P system. They demonstrated generally good scalability and found that the compositing phase slowed down when more than ten thousand cores were involved, likely due to hardware or MPI limitations. To address this problem, they reduced the number of processors involved in the compositing phase. Later, Peterka et al. introduce the radix-k parallel compositing algorithm [19] to address compositing performance at large concurrency. Kendall et al. [20] build upon this work by demonstrating an approach for tuning radix-k's algorithmic parameters to various architectures, and include new compression and load balancing optimizations for better scalability and performance on diverse architectures.

The most substantial difference between our work and previous work in parallel volume rendering is that we are exploiting hybrid parallelism at extreme concurrency, and performing in-depth studies to better understand scalability characteristics as well as potential performance gains of the hybrid-parallel approach.

2.2 Hybrid Parallelism

Hybrid parallelism has evolved in response to the widespread deployment of multi-core chips in distributed-memory systems. The hybrid model allows data movement among nodes using traditional MPI motifs like scatter and gather, but within nodes using shared-memory parallelism via threaded frameworks like POSIX threads or OpenMP. Previous work comparing distributed-memory versus hybrid-memory implementations (e.g., [21], [22]) has focused on benchmarking well-known computational kernels. In contrast, our study examines this space from the perspective of visualization algorithms.

The previous studies point to several areas where hybrid memory may outperform distributed memory. First, hybrid memory tends to require a smaller data footprint for applications with domain decomposition (e.g. parallel volume rendering), since fewer domains means less "surface area" between domains and hence less exchange of "ghost" data. Second, the MPI runtime allocates various tables, buffers, and constants on a per-task basis. Today, the gain from using fewer tasks to reduce this memory overhead may seem small with only four or six cores per chip, but the trend towards hundreds of cores per chip with less memory per core will amplify these gains. Third, hybrid-memory implementations can use a single MPI task per node for collective operations like scatter-gather and all-to-all, thereby reducing the absolute number of messages traversing the inter-connect. While the size of the messages in this scenario may be larger or smaller depending upon the specific problem, a significant factor influencing overall communication performance is latency, the cost of which is reduced by using fewer messages.

Peterka et al. [23] investigated a hybrid-parallel implementation of a volume renderer that uses four POSIX threads per MPI task on the IBM BlueGene/P architecture. They identify similar improvements during the compositing phase as we report in our study. Their strong scaling study extends to 4096-way concurrency with a 1120^3 dataset and an 1204^2 image, whereas we conduct both strong and weak scaling studies with up to a 23040^3 dataset and 216,000-way concurrency, and compare hybrid-memory implementations that use POSIX threads, OpenMP, and CUDA.

Fogal et al. [24] have recently studied the performance of volume rendering on a larger multi-GPU cluster. While their use of an MPI-based compositing phase is similar to our implementation, they use a different slice-based volume rendering technique implemented with GLSL shaders in OpenGL. Their implementation is an add-on to VisIt, a production parallel visualization tool, whereas our implementation is a stand-alone research prototype. Still, their results show similar advantages for increasing the density of cores per node, as made possible by a GPU cluster. Because they use out-of-core access to CPU memory, they are able to process datasets as large as 8192^3 on 256 GPUs. In contrast, we operate within the available device memory of 448 GPUs to process a 4608^3 dataset. However, we explore image sizes as large as 4608^2, while Fogal et al. use a smaller 1024×768 image.

3 Implementation

From a high level view, our parallel volume rendering implementation using a design pattern that forms the basis of previous work (e.g., [9]–[12]). Given a source data volume S and n parallel tasks, each task reads in $1/n$ of S, performs raycasting volume rendering on this data subdomain to produce a set of image fragments, then participates in a compositing stage in which fragments are exchanged and combined into a final image. The completed image is gathered to the root task for display or I/O to storage. Figure 2 provides a block-level view of this organization.

Our distributed-memory implementation is written in C/C++ using the MPI [25] library. The portions of the implementation that are shared-memory parallel are written using a combination of C/C++ and and either POSIX threads [26], OpenMP [27], or CUDA (version 3.0) [28] so that we are actually comparing three hybrid-memory implementations that we refer to as hybrid/pthreads, hybrid/OpenMP, and hybrid/CUDA.

The distributed-memory and hybrid-memory implementations differ in several key respects. First, the raycasting volume rendering algorithm runs in serial on the distributed-memory tasks, but is multi-threaded on the hybrid-memory tasks. We discuss this issue in more detail in Section 3.1. Second, the communication topology in the compositing stage differs slightly, and we discuss this issue in more detail in Section 3.2. A third difference is in how data is partitioned across the tasks. In the distributed-memory implementation, each task loads and operates on a disjoint block of data. In the hybrid-memory implementation, each task loads a disjoint block of data and each of its worker threads operate in parallel on that data using an image-parallel decomposition [15].

3.1 Parallel Raycasting

Our raycasting volume rendering code implements Levoy's method [6]: we compute the intersection of a ray with a data block, and then compute color at a fixed step size along the ray through the volume. All colors along the ray are composited front-to-back using the "over" operator. Output consists of a set of image fragments that contain an x, y pixel location, R, G, B, α color, and a z-coordinate. The z-coordinate is the location in eye coordinates where the ray penetrates the block of data. Later, these fragments are composited in the correct order to produce a final image (see Section 3.2).

Each distributed-memory task invokes a serial raycaster that operates on its own disjoint block of data. Since we are processing structured rectilinear grids, all data subdomains are spatially disjoint, so we can safely use the ray's entry point into the data block as the z-coordinate for sorting during the subsequent composition step.

In contrast, the hybrid-memory tasks invoke a raycaster with shared-memory parallelism that consists of T threads executing concurrently to perform the raycasting on a shared block of data. As in [15], we use an image-space partitioning: each thread is responsible for raycasting a portion of the image. In the pthreads and OpenMP raycasting implementation, our image-space partitioning is interleaved, with the image divided into many tiles that are distributed amongst the threads. The CUDA raycasting implementation is slightly different because of the data-parallel nature of the language: the image is treated as a 2D CUDA grid, which is divided into CUDA thread blocks. Each thread block corresponds to an image tile, and the individual CUDA threads within each block are mapped to individual pixels in the image.

We directly ported our pthreads raycasting implementation to CUDA and made no additional optimizations to

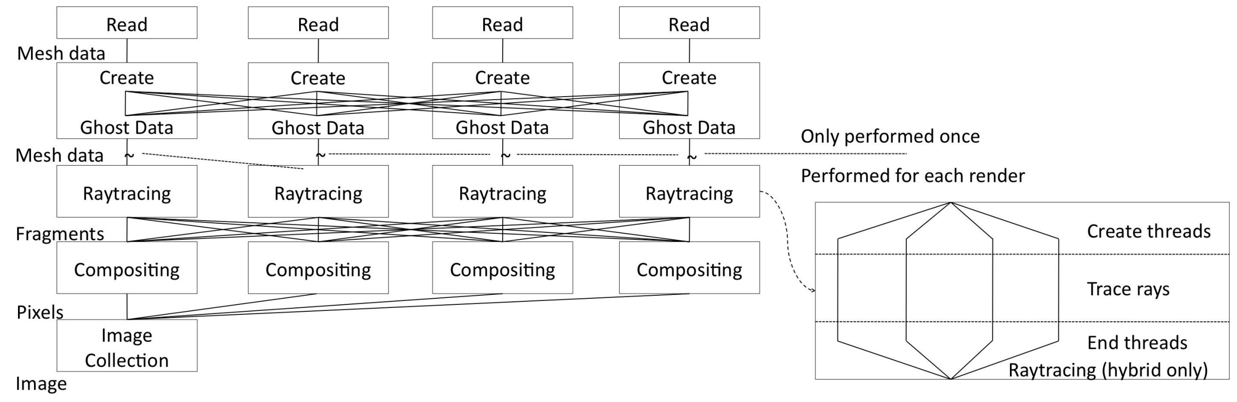

Fig. 2. Diagram of our system architecture.

improve performance. Even though CUDA thread blocks require a minimum of 32 threads to saturate computational throughput, our optimal configuration uses only 8 threads, which points to the branching nature of our algorithm that causes "divergence" among CUDA threads. A thread block is executed in a single-instruction-multiple-thread (SIMT) fashion in which "warps" of 32 threads are executed across 4 clock cycles in sets of 8 threads that share a common instruction. If those 8 threads do not share a common instruction, such as when conditionals cause branching code paths, the threads "diverge" and must be executed individually. This situation is prevalent in our algorithm. For example, imagine a thread block owning a region of the image that only partially covers the data volume. Some of the threads immediately exit because of the "empty-space-skipping" [16] optimization in our algorithm, while the other threads proceed to cast rays through the volume. Even the threads that proceed together with raycasting have rays of different lengths, which can cause divergence and load imbalance.

It is possible to improve performance of the CUDA raycasting implementation by dividing it into separate stages for calculating the ray intersections and integrating the rays, but the trade-off is the additional memory required to temporarily buffer the rays between the two stages. Although we tested this optimization and it led to better performance at small scales, we chose to use the direct CUDA raycasting port in our study so as to maintain a algorithmically consistent comparison with the pthreads and OpenMP implementations.

3.2 Parallel Compositing

Compositing begins by partitioning the pixels of the final image across the tasks. Next, an all-to-all communication step exchanges each fragment from the task where it was generated in the raycasting phase to the task that owns the region of the image in which the fragment lies. This exchange is done using an `MPI_Alltoallv` call. After the exchange, each task then performs the final compositing for each pixel in its region of the image using the "over" operator, and the final image is gathered on the root task.

`MPI_Alltoallv` provides the same functionality as performing direct sends and receives, but bundles the messages into fewer point-to-point exchanges for greater efficiency. This call uses a variety of strategies based on concurrency level and total message size. One of the strategies employed is pairwise exchange, so, provided the heuristics of which strategy to employ are good, the `MPI_Alltoallv` should always match or exceed that of pairwise exchange. Pairwise exchange shares certain characteristics with binary swap, namely messages are collated and exchanged in a way that minimizes the number of point to point communications. The key difference between binary swap and pairwise exchange is that binary swap calculates intermediate compositings as messages are exchanged between the tasks, reducing the message sizes as as the algorithm progresses. Although our implementation may appear to be related to direct-send at first glance, it in fact has performance characteristics closer to binary swap, with the caveat that the total message size is greater. Of course, an optimized binary swap implementation would outperform our implementation and the results from Peterka et al in [4] showed the Radix-K provided further improvement, improving on binary swap by as much as 40%. Regardless, we believe the number of messages and message sizes for `MPI_Alltoallv` is similar to those of the more sophisticated algorithms and therefore findings regarding the benefits of hybrid parallelism also provide evidence of potential benefit for those algorithms.

Peterka et al. [4] reported scaling difficulties for compositing when using more than 8,000 tasks. They solved this problem by reducing the number of tasks receiving fragments to be no more than 2,000. We emulated this approach, again limiting the number of tasks receiving fragments, although we experimented with values higher than 2,000.

In the hybrid-memory implementations, only one thread per socket participates in the compositing phase. That thread gathers fragments from all other threads in the same socket, packs them into a single buffer, and transmits them to other compositing tasks. This approach results in fewer messages than if all threads in the hybrid-memory implementation were to send messages to all other threads. Our aim here is to better understand the opportunities for improving performance in the hybrid-memory implementation. The overall effect of this design choice is an improvement in communication characteristics, as indicated in Section 5.4.

4 METHODOLOGY

Our methodology is designed to test the hypothesis that a hybrid-memory implementation exhibits better performance and resource utilization than the distributed-memory implementation.

Our multi-core CPU test system, JaguarPF, is a Cray XT5 located at Oak Ridge National Lab and in 2009 was ranked number one on the list of Top 500 fastest supercomputers with a peak theoretical performance of 2.3 Petaflop [29]. Each of its 18,688 nodes has two sockets, and each socket has a six-core 2.6GHz AMD Opteron processor, for a total of 224,256 compute cores. With 16GB per node (8GB per socket), the system has 292TB of aggregate memory and roughly 1.3GB per core.

Our many-core GPU cluster, Longhorn, is located at the Texas Advanced Computing Center and has 256 host nodes with dual-socket quad-core Intel Nehalam CPUs and 24GB of memory. They share 128 NVIDIA OptiPlex 2200 external quad-GPU enclosures for a total of 512 FX5800 GPUs. Each GPU has a clock speed of 1.3Ghz, 4GB of device memory, and can execute 30 CUDA thread blocks concurrently. We choose to treat the FX5800 as a generic "many-core" processor with a data-parallel programming model (CUDA) that serves as a surrogate for anticipating what future many-core clusters may look like. In terms of performance, we position the FX5800 relative to the Opteron in terms of its actual observed runtime for our particular application rather than relying on an *a priori* architectural comparison.

We conducted the following three studies:

- **strong scaling**, in which we fixed the image size at 4608^2 and the dataset size at 4608^3 (97.8 billion cells) for all concurrency levels;
- **weak–dataset scaling**, with the same fixed 4608^2 image, but a dataset size increasing with concurrency up to 23040^3 (12.2 trillion cells) at 216,000-way parallel; and
- **weak scaling**, in which we increased both the image and dataset up to 23040^2 and 23040^3, respectfully, at 216,000-way parallel.

Note that at the lowest concurrency level, all three cases coincide with a 4608^2 image and 4608^3 dataset size.

On JaguarPF, we share a data block among six threads and use one sixth as many tasks. Although we could have shared a data block among as many as twelve threads on each dual-socket six-core node, sharing data across sockets results in non-uniform memory access. Based on preliminary tests, we estimated this penalty to be around 5 or 10% of the raycasting time. Therefore, we used six threads running on the cores of a single six-core processor. On Longhorn, we use one task per GPU device, and load the data block into GPU device memory where it is shared among all CUDA threads.

Because the time to render is view-dependent, we executed each raycasting phase ten times over a selection of ten different camera locations. The raycasting times we report are an average over all locations.

In the compositing phase, we tested either four or five (depending on memory constraints) ratios of total tasks to compositing tasks. We restricted the compositing experiment to only two views (the first and last) because it was too costly to run a complete battery of all view and ratio permutations. Since the runtime of each trial can vary due to contention for JaguarPF's interconnection fabric with other users, we ran the compositing phase ten times for both views. We report mean and minimum times over this set of twenty trials. Minimum times most accurately represent what the system is capable of under optimal, contention-free conditions, while mean times help characterize the variability of the trials.

4.1 Determining Optimal Algorithmic Parameters

The size and shape of the image tiles used in an image-space partitioning for shared-memory raycasting can be treated as a tunable parameter. To determine which configurations performed best on the AMD Opteron CPU, we ran a parameter sweep over 100 different sizes and shapes using our shared-memory raycaster on a 384^3 dataset on a single CPU with six threads. We varied the image tile width and height over the range $\{1, 2, 4, \ldots, 256, 512\}$ and measured frame rendering time (FRT) over thirty different views using dynamic work assignments. For this problem, there are three "invalid" block size configurations, 256×512, 512×512, and 512×256, that produce decompositions resulting in only one or two total work blocks.

From the test results, shown in Figure 3, we see clear "sweet" and "sour" spots in performance. To produce that figure, we "normalized" FRT by the minimum so that values range from $[1.0 \ldots \infty]$. In this particular set of results, the maximum data value is about 3.0, which means the FRT of the worst-performing tile size was about three times as slow as the best-performing configuration. We see that very small and very large image tiles produce poorer performance: at small tile sizes, the threads' access to the work list is serialized through a mutex and performance is adversely affected. At large tile sizes, there are not enough tiles to result in good load balance. Therefore, the sweet spot tends to fall in regions of medium-sized tiles.

An interesting feature in Figure 3 is the relatively poor performance for block widths of $\{1, 2, 4\}$. At these configurations, we observe in hardware performance counter

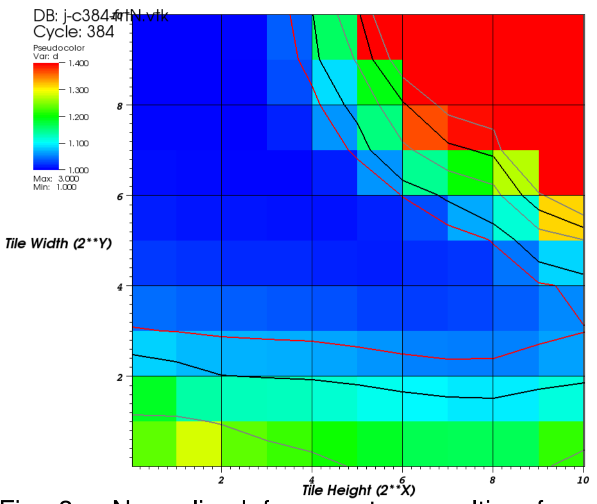

Fig. 3. Normalized frame rates resulting from using different image tile sizes in the pthreads implementation. Reds correspond to higher and blues to lower frame rates.

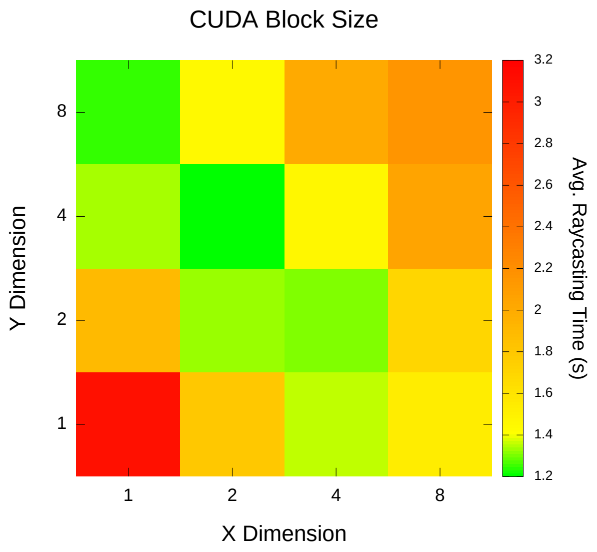

Fig. 4. A sweep over CUDA thread block sizes found that 4×2 was the optimal configuration.

data a relatively high level of L1, L2, and TLB cache misses (not shown due to space limitations); there seems to be a direct correlation between higher cache miss rates and higher FRT. The reason for higher cache miss rates in general is due to degraded spatial and/or temporal locality of memory accesses. We are using a row-major ordering of volume data in memory for each data block, which could be a contributing factor to relatively higher cache miss rates in certain configurations. More study, including comparison with more cache-friendly layout schemes like Z-ordering [30], would help reveal more insights into the cache utilization characteristics of different parameter choices. Nonetheless, the main point here is that our approach, which is consistent with autotuning in general, is to empirically measure performance across a range of algorithmic parameters in order to determine the configurations that perform better and worse for a given problem configuration on a given platform. The alternative is to attempt to derive a quantitative performance model of a complex system in order to predict which parameter settings will result in the best performance given a specific algorithm, problem configuration, and machine architecture.

Based upon the results of this study, we decided to perform our scalability studies using an image tile size of 32×32 pixels. That tile size lies within the sweet spot that is visible in Figure 3. Other tile size choices could be equally valid: we see that tile sizes having relatively large width and relatively small height also lie within the sweet spot of FRT.

We also performed a sweep over CUDA thread block sizes on the FX5800 using the MPI+CUDA implementation on a subset of 48 GPUs (see Figure 4). For the FX5800, the maximum number of CUDA threads per thread block is 512, and all registers for all threads in the block must fit in only 16KB of memory. With these constraints, we swept over thread block sizes up to 8×8. The experiment found that 4×2 was the optimal size.

On the GPU, image tiles (CUDA thread blocks) are automatically load balanced by the CUDA runtime's scheduler. On the CPU, however, we had the choice of implementing either a dynamic or static work assignment of image tiles to threads. We ran a similar study where we varied tile size across both algorithms. The results of that study, not shown here due to space limitations, indicate the static assignment produces slightly better frame rates at the smallest block sizes, but that the dynamic assignment produces better results at medium and larger block sizes and has better load balance characteristics across a larger range of block sizes. Therefore, we chose to use the dynamic work assignment since it results in better performance than static assignments for 32×32 tiles.

4.2 Source Data and Decomposition

Starting with a 512^3 dataset of combustion simulation results [1], we used trilinear interpolation to upscale it to arbitrary sizes in memory. We scaled equally in all three dimensions to maintain a cubic volume. Our goal was to choose a problem size that came close to filling all available memory (see Table 1). Although upscaling may distort the results for a data-dependent algorithm, the only data dependency during raycasting is early ray termination. However, we found that for our particular dataset and transfer function, there was always at least one data block for which no early terminations occurred. Moreover, the cost of the extra conditional statement inside the ray integration loop to test for early termination added a 5% overhead. Therefore, we ran our study with early ray termination turned off, and we believe that upscaling the dataset does not effect our results.

1. Sample data courtesy J. Bell and M. Day, Center for Computational Sciences and Engineering, Lawrence Berkeley National Laboratory.

TABLE 1
Problem Configurations

			Strong Scaling			*Weak Scaling*
	Distributed Tasks	**Hybrid Tasks**	**Distributed Block**	**Hybrid Block**	**Memory Per GPU/Node (MB)**	**Data Size**
GPU	-	56	-	$1152 \times 576 \times 329$	3331	$2304 \times 2304 \times 2303$
	-	112	-	$576 \times 576 \times 329$	1666	$2900 \times 2900 \times 2898$
	-	224	-	$576 \times 288 \times 329$	833	$3656 \times 3656 \times 3654$
	-	448	-	$288 \times 288 \times 329$	416	$4608 \times 4608 \times 4606$
CPU	1728 (12^3)	288	$384 \times 384 \times 384$	$384 \times 768 \times 1152$	10368	4608^3
	13824 (24^3)	2304	$192 \times 192 \times 192$	$192 \times 384 \times 576$	1296	9216^3
	46656 (36^3)	7776	$128 \times 128 \times 128$	$128 \times 256 \times 384$	384	13824^3
	110592 (48^3)	18432	$96 \times 96 \times 96$	$96 \times 192 \times 288$	162	18432^3
	216000 (60^3)	36000	$76 \times 76 \times 76$	$76 \times 153 \times 230$	80.4/81.6	23040^3

Because the compute nodes on an XT5 system have no physical disk for swap space, allocating beyond the amount of physical memory causes program termination. Our total memory footprint was four times the number of bytes in the entire dataset: one for the dataset itself, and the other three for the gradient data, which we computed by central difference and used in shading calculations. Although each node has 16GB of memory, we could reliably allocate only 10.4GB for the data block and gradient field at 1,728-way concurrency because of overhead from the operating system and MPI runtime library.

On JaguarPF, we chose concurrencies that are cubic numbers to allow for a clean decomposition of the entire volume into cubic blocks per distributed-memory task. In hybrid memory, these blocks are rectangular because we aggregated six blocks ($1\times2\times3$) into one shared block.

4.2.1 Strong Scaling

On JaguarPF, we used a fixed 4608^3 dataset at all concurrencies, except for 216,000-way where the 4608^3 dataset could not be evenly divided. With increasing concurrency, each task was assigned a correspondingly smaller data block, as seen in Table 1.

At 216,000-way concurrency we rounded down to a 4560^3 distributed-memory dataset and a $4560 \times 4590 \times 4600$ hybrid-memory dataset. As a result, the distributed-memory dataset is approximately 1.4% smaller. While this difference might seem to give an advantage to the distributed-memory implementation, results in later sections show that hybrid-memory performance and resource utilization are uniformly better.

On Longhorn, we were unable to use all 512 GPUs, so we instead configured our largest test as $8 \times 8 \times 7$ blocks over 448 GPUs. Starting at 56-way concurrency, the largest dataset we could reliably fit in GPU device memory was 2304^3. We processed this data size with a matching image size of 2304^2 at concurrencies in the range $\{56, 112, 224, 448\}$.

4.2.2 Weak–dataset and Weak Scaling

On JaguarPF, we assigned a 384^3 block to each distributed-memory task t all concurrency levels, and aggregated these blocks into a $384 \times 786 \times 1152$ block for the hybrid-memory tasks. As a result, we processed datasets with sizes ranging from 4608^3 to 23040^3 (see Table 1). For the weak–dataset scaling, we maintained the same 4608^2 image size at all concurrency levels, whereas for weak scaling we scaled the image size up to 23040^2.

On Longhorn, we conducted a weak scaling study from 56- to 448-way concurrency with data sizes ranging from 2304^3 to 4608^3 and image sizes from 2304^2 to 4608^2. We had to round down the z-dimensions of the dataset slightly to maintain integer multiples in the decomposition (see 'Data Size' in Table 1).

5 Results

We compare the cost of MPI runtime overhead and corresponding memory footprint in Section 5.1; the absolute amount of memory required for data blocks and ghost (halo) exchange in Section 5.2; the scalability of the ray-casting and compositing algorithms in Sections 5.3 and 5.4; and the communication resources required during the compositing phase in Section 5.4. We conclude in Section 5.5 with a comparison of results from the six-core CPU system and the many-core GPU system to understand how the balance of shared-memory versus distributed-memory parallelism affects overall performance.

5.1 Initialization

Because there are fewer hybrid-memory tasks, they incur a smaller aggregate memory footprint for the MPI runtime environment and program-specific data structures that are allocated per task. Table 2 shows the memory footprint of the program as measured directly after calling `MPI_Init` and reading in command-line parameters [2]. Memory usage was sampled only from tasks 0 through 6, but those values agreed within 2% of each other. Therefore, the per-task values we report in Table 2 are from task 0 and the per-node and aggregate values are calculated from the per-task value.

2. We collected the `VmRSS`, or "resident set size," value from the `/proc/self/status` interface.

TABLE 2
Memory Usage at MPI Initialization

CPU Cores	Memory Type	Tasks	Per Proc. (MB)	Per Node (MB)	Agg. (GB)
1728	Hybrid	288	67	133	19
1728	Distrib.	1728	67	807	113
13824	Hybrid	2304	67	134	151
13824	Distrib.	13824	71	857	965
46656	Hybrid	7776	68	136	518
46656	Distrib.	46656	88	1055	4007
110592	Hybrid	18432	73	146	1318
110592	Distrib.	110592	121	1453	13078
216000	Hybrid	36000	82	165	2892
216000	Distrib.	216000	176	2106	37023

TABLE 3
CUDA Initialization Time (s)

	Concurrency			
Scaling	56	112	224	448
Strong	2.10	2.06	1.83	1.73
Weak	2.10	2.15	2.53	2.47

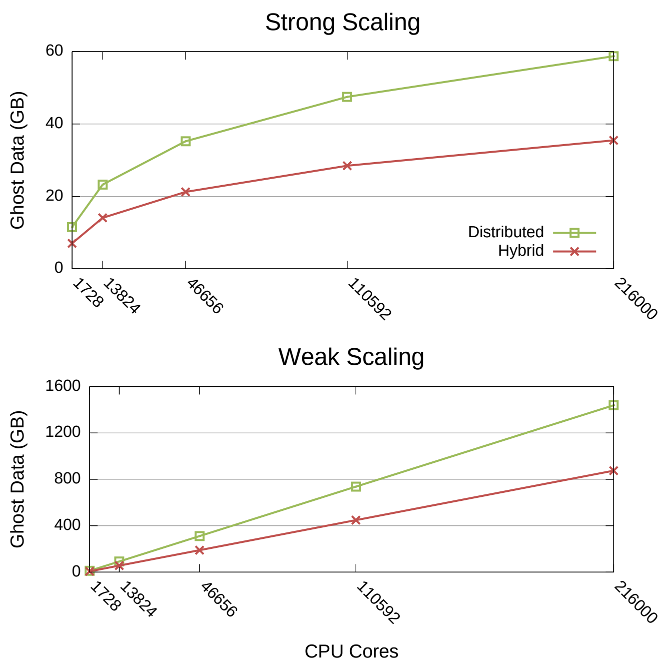

Fig. 5. Ghost data requirements.

The distributed-memory implementation uses twelve tasks per node while the hybrid-memory one uses only two. At 216,000-way concurrency, the runtime overhead per distributed-memory task is more than 2× the overhead per hybrid-memory task. The per-node and aggregate memory usage is another factor of six larger for the distributed-memory implementation because it uses 6× as many tasks. Thus, the distributed-memory implementation uses nearly 12× as much memory per-node and in-aggregate than the hybrid-memory implementation for initializing the MPI runtime at 216,000-way concurrency.

A major disadvantage of using GPU co-accelerators is the cost of initializing the GPU device in CUDA and of transferring data between host and device memories. For our hybrid/CUDA implementation, it took up to 2.53s to initialize the GPU on each node and transfer the data block from CPU memory to GPU memory across the PCI-Express bus (transferring the array of image fragments back to CPU memory had negligible cost). Although this upfront cost can be amortized in the use case where several volume renderings are computed for the same dataset, if only one volume rendering is computed, then this cost makes the hybrid/CUDA implementation less competitive than the hybrid/pthreads or hybrid/OpenMP ones. However, the initialization cost is a limitation of the hybrid/CUDA implementation in particular, and not the hybrid-parallelism approach in general.

5.2 Ghost Data

Two layers of ghost data are required in our raycasting phase: the first layer for trilinear interpolation of sampled values, and the second layer for computing the gradient field using central differences (gradients are not pre-computed for our dataset). Because hybrid memory is decomposed into fewer, larger blocks, it requires less exchange and storage of ghost data by roughly 40% across all concurrency levels and for both strong and weak scaling (see Figure 5).

5.3 Raycasting

All of the scaling studies demonstrated good scaling for the raycasting phase, as no message passing is involved (see Figure 6). For these runs and timings, we used trilinear interpolation for data sampling along the ray as well as a Phong-style shader. The final raycasting time is essentially the runtime of the thread that takes the most integration steps. This behavior is entirely dependent on the view. Our approach, which is aimed at understanding "average" behavior, uses ten different views and reports an average runtime.

In the strong scaling study, we achieved linear scaling up to 216,000-way concurrency for the raycasting phase with distributed memory (see Figure 6). The hybrid-memory implementation exhibited different scaling behavior because of its different decomposition geometry: the distributed blocks had a perfectly cubic decomposition, while in hybrid memory we aggregated $1 \times 2 \times 3$ cubic blocks into a larger rectangular block (see Table 1). The smaller size of the GPU cluster limited the concurrencies we could choose for the hybrid/CUDA implementation, leading to similarly irregular blocks in that case.

The interaction of the decomposition geometry and the camera direction determines the maximum number of ray integration steps, which is the limiting factor for the raycasting time. At lower concurrencies, this interaction benefited the hybrid-memory implementation by

TABLE 4
Raycasting Time (s)

	Strong Scaling			*Weak–dataset Scaling*			*Weak Scaling*			**Hybrid/CUDA**		
Cores	**Distributed**	**Hybrid/ pthreads**	**Hybrid/ OpenMP**	″	″	″	″	″	″	**GPUs**	*Strong*	*Weak*
1728	24.88	22.31	22.23	24.88	22.31	22.23	24.88	22.31	22.23	56	1.04	1.04
13824	3.10	2.84	2.83	7.33	7.06	7.06	26.65	24.92	24.93	112	0.54	1.09
46656	0.92	0.85	0.85	3.56	3.51	3.51	27.18	25.88	25.88	224	0.29	1.16
110592	0.38	0.37	0.37	2.13	2.15	2.15	27.45	26.59	26.50	448	0.15	1.18
216000	0.19	0.21	0.20	1.46	1.48	1.58	*22.17	26.79	26.88			

* *Data size is smaller due to memory constraints.*

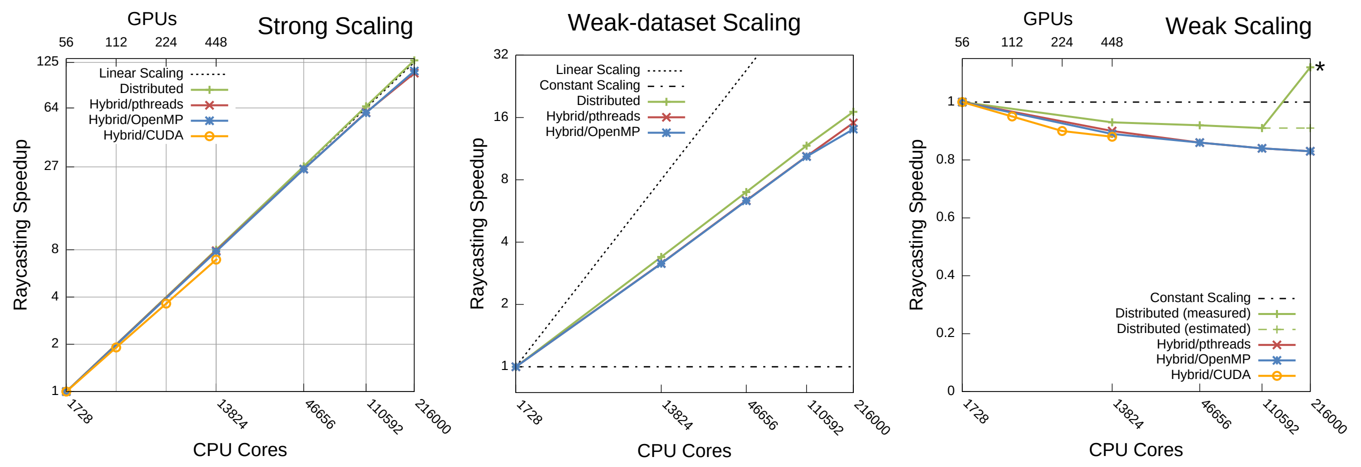

Fig. 6. The speedups (referenced to 1,728 cores) for both the raycasting phase and the total render time (raycasting and compositing). The raycasting speedup is linear with distributed memory, but sublinear with hybrid memory: this effect is caused by the difference in decomposition geometries (cubic versus rectangular).

as much as 11% (see Table 4). At higher concurrencies the trend flips and the distributed-memory implementation outperforms the hybrid-memory one by 10%. We expect that if we were able to run the hybrid-memory implementation with regular blocks (such as $2 \times 2 \times 2$ on an eight-core system), both implementations would scale identically. We also note that at 216,000 cores, raycasting is less than 20% of the total runtime (see Figure 9), and the hybrid-memory implementation is over 50% faster because of gains in the compositing phase that we describe in the next subsection.

For weak scaling, the hybrid-memory implementation maintained 80% scalability out to 216,000 cores. Overall raycasting performance is only as fast as the slowest thread and because of perspective projection, the number of samples each thread must calculate varies. This variation becomes larger at scale as each core is operating on a smaller portion of the overall view frustum, which accounts for the 20% degradation.

The distributed-memory result at 216,000-way concurrency appears (misleadingly) to be superlinear, but that is because we could not maintain the data size per core. Although we could maintain it for the weak–dataset scaling, increasing the image size to 23040^3 in the weak scaling study caused the temporary buffers for the image fragments (the output of the raycasting phase) to overflow. To accommodate the fragment buffer, we had to scale down to a data size of 19200^3 (7.1 trillion cells), instead of the 23040^3 data size (12.2 trillion cells) used in hybrid memory. We report the measured values for this smaller data size, and also estimated values for the full data size assuming linear scaling by the factor $23040^3/19200^3$.

For the weak–dataset scaling study, the expected scaling behavior is neither linear nor constant, since the amount of work for the raycasting phase is dependent on both data size and image size. With a varying data size but fixed image size, the scaling curve for weak–dataset scaling should lie between those of the pure weak and pure strong scaling, which is what we observe. Overall, 216,000-way concurrency was $10\times$ faster than 1,728-way concurrency.

5.4 Compositing

Above 1,728-way concurrency, we observed that compositing times are systematically better for the hybrid-memory implementation (see Figure 7). The compositing phase has two communication costs: the `MPI_Alltoallv` call that exchanges fragments from the task where they originated during raycasting to the compositing task that owns their region of image space; and the `MPI_Reduce` call that reduces the final image components to the root task for assembly and output to a file or display (see Figure 9 for a breakdown of

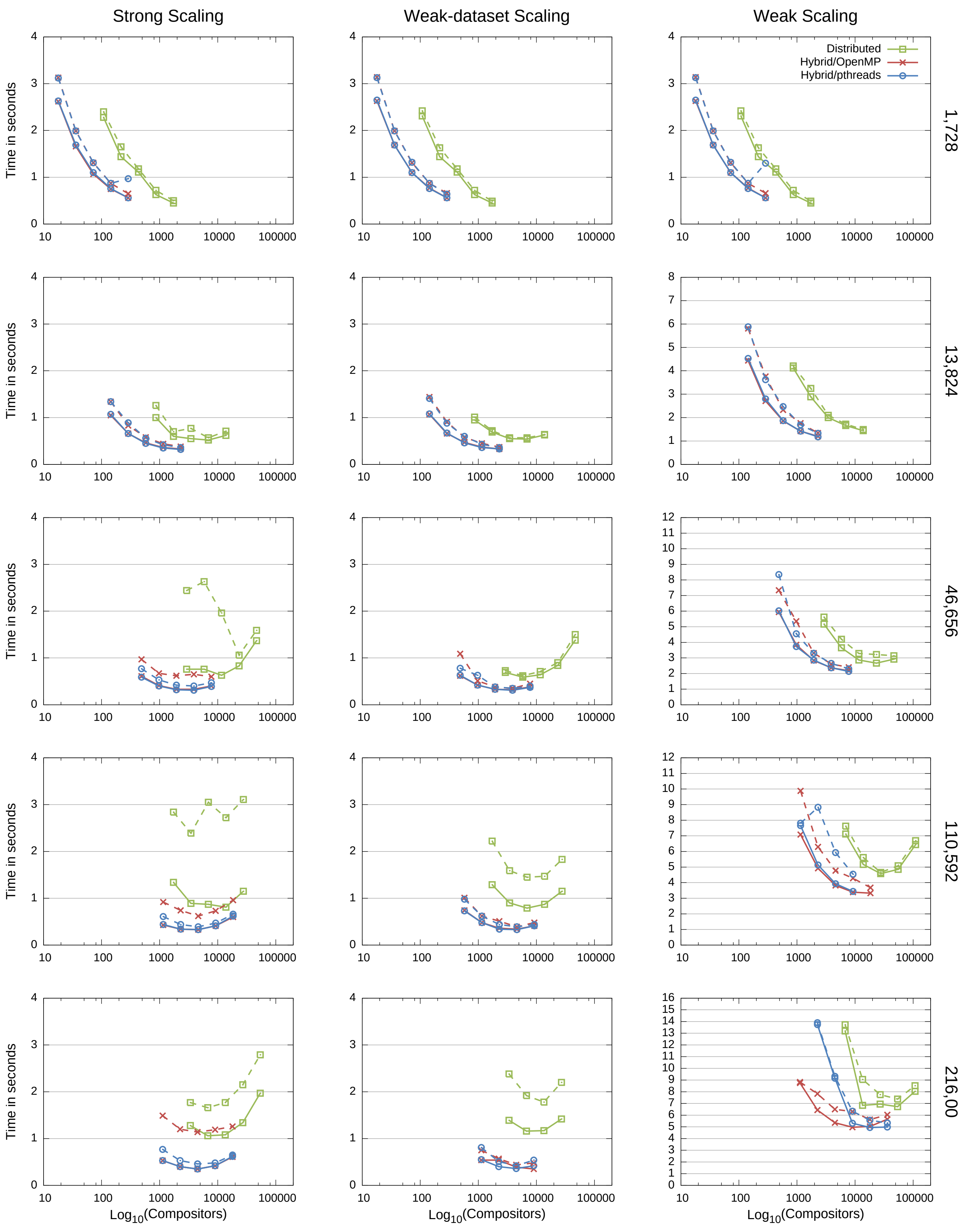

Fig. 7. Compositing times for different ratios of compositing tasks to total tasks. Solid lines show minimum times taken over ten trials each for two different views; dashed lines show the corresponding mean times.

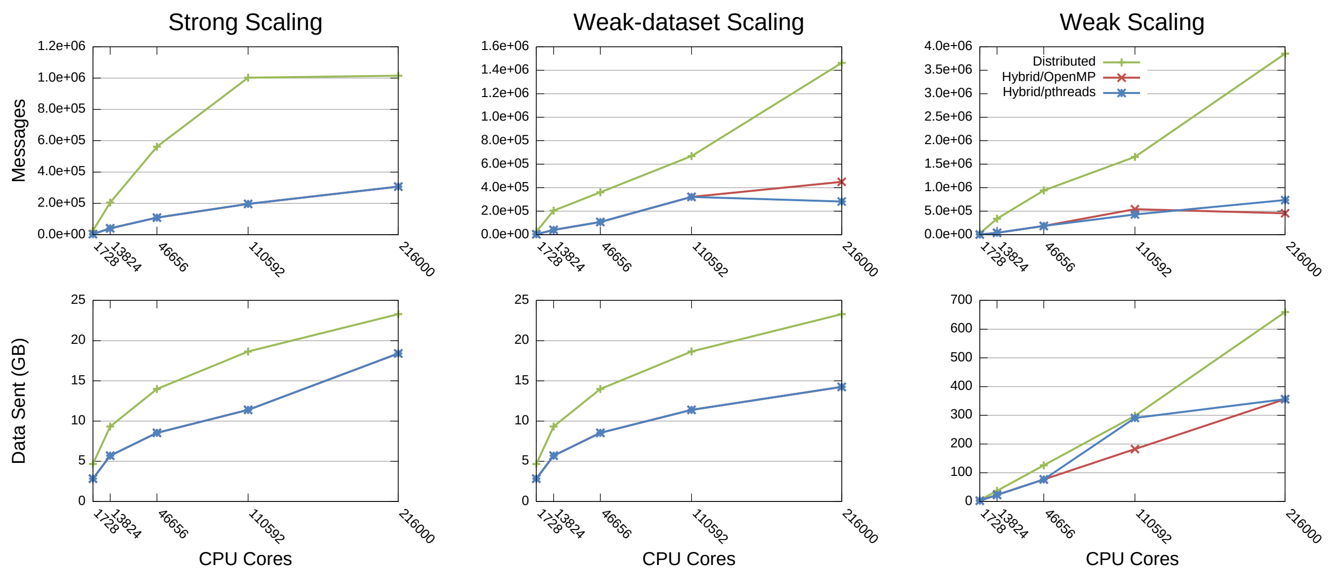

Fig. 8. The number of messages and total data sent during the fragment exchange in the compositing phase.

these costs). During the fragment exchange, the hybrid-memory implementation can aggregate the fragments in the memory shared by six threads, and therefore it uses on average about $6\times$ fewer messages than the distributed-memory implementation (see Figure 8). In addition, the hybrid-memory implementation exchanges less fragment data because its larger data blocks allow for more compositing to take place during ray integration. Similarly, one sixth as many hybrid-memory tasks participate in the image reduction, which leads to better performance.

Our compositing phase used a subset of tasks for compositing and we explored different ratios of compositors to renderers. For the strong and weak–dataset scaling studies, we observe an inflection point starting at 13,824-way concurrency for hybrid memory and 46,656-way for distributed memory, where the optimal compositing configuration is to use a subset in the range of 2,000 to 8,000 tasks. We believe this inflection point exists due to the characteristics of the underlying interconnect fabric. Because the hybrid-memory implementation generates fewer and smaller messages, the critical point occurs at a higher level of concurrency than for distributed memory. Interestingly, in the weak scaling study, the inflection point does not arise until higher concurrency, and using a larger subset of compositing tasks in the range of 2,000 to 30,000 is optimal. Moreover, it was not possible to use subsets as small as in the strong scaling study because of the larger image size with weak scaling; setting the subset too small led to overallocation from temporary data structures and MPI buffers as the additional data was concentrated in fewer compositors.

Peterka et al. [4] first observed that the reduced number of compositors at higher concurrencies increased overall performance. In their strong scaling study, they concluded that 1,000 to 2,000 compositors were optimal for up to 32,768 total tasks. We note, however, that there are many differences between our study and theirs in the levels of concurrency, architectures, operating systems, communication networks, and MPI libraries, each potentially introducing variation in the ideal number of compositors.

5.5 Overall Performance

In the strong scaling study at 216,000-way concurrency, the best compositing time with hybrid memory (0.35s, 4500 compositors) was $3\times$ faster than with distributed (1.06s, 6750 compositors). Furthermore, at this scale compositing time dominated rendering time, which was roughly 0.2s for both implementations. Thus, the total render time was $2.2\times$ faster with hybrid memory (0.56s versus 1.25s). Overall, the strong scaling study shows that the advantage of hybrid memory over distributed becomes greater as the number of cores increases (see Figure 9).

For weak–dataset and weak scaling, the hybrid-memory implementation still shows gains over the distributed-memory one, but they are less pronounced because the raycasting phase dominates. Although the 216,000-way breakdown for weak scaling looks like it favors distributed memory, this is actually an artifact of the reduced data size (19200^3) we were forced to use in the distributed-memory implementation to avoid out-of-memory errors. Comparing an estimated value for a 23040^3 distributed-memory data size suggests that the hybrid-memory implementation would be slightly faster.

At 448-way concurrency with the 4608^3 dataset, the hybrid/CUDA implementation averaged 1.18s for raycasting and exhibited a minimum compositing time of 0.15s (median of 0.19s). The raycasting performance positions this run close to the 46,656-way run on JaguarPF (see Figure 10). However, the compositing time for the

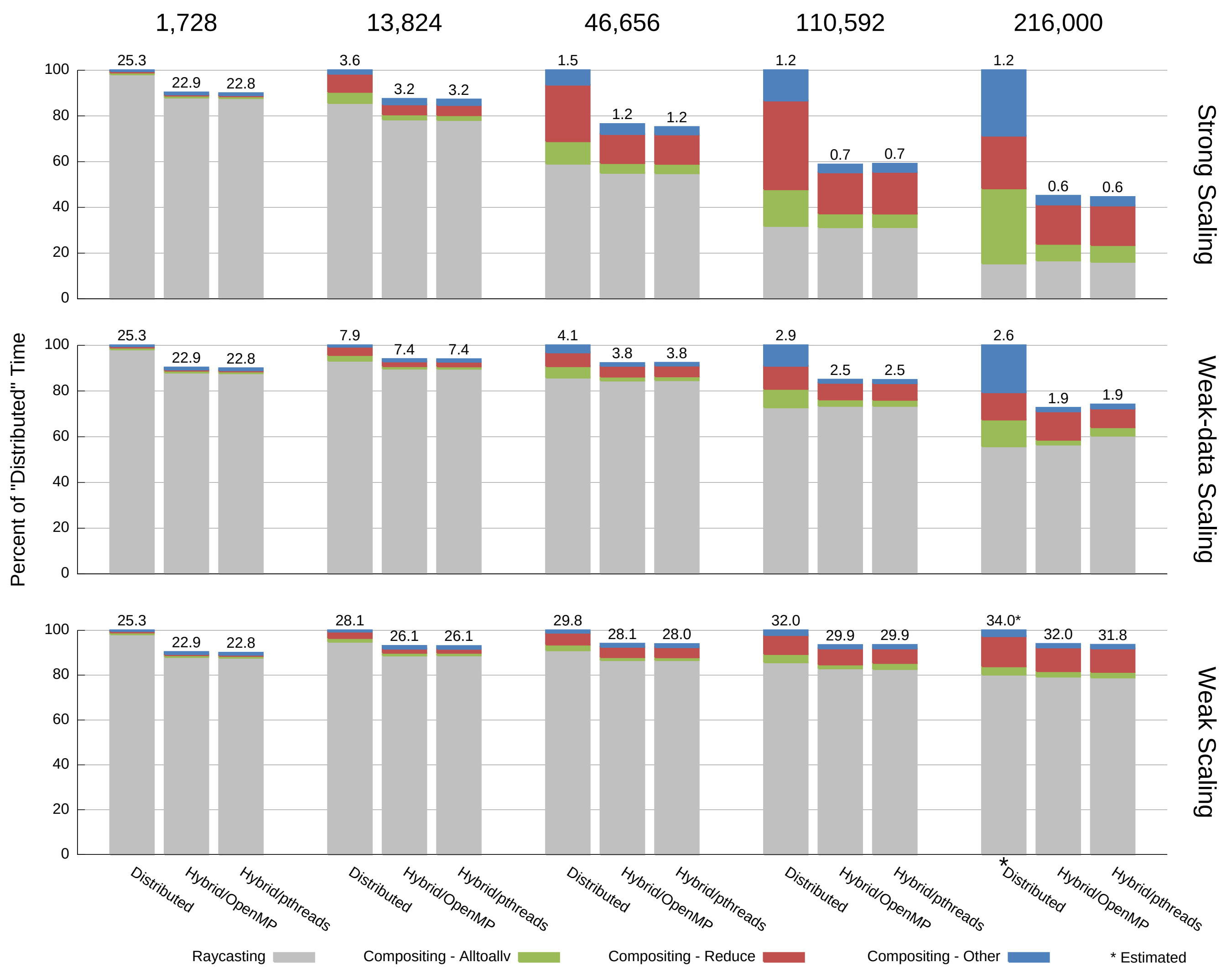

Fig. 9. Total render time split into raycasting and compositing components and normalized at each concurrency level as a percentage of the distributed-memory time. "Compositing – Other" includes the time to coordinate the destinations for the fragments and to perform the over operator.

hybrid/CUDA run on Longhorn (0.15s) is half that of hybrid-memory runs at 46,656-way concurrency on JaguarPF (0.31 to 0.33s).

This result from Longhorn points more generally to the increasing potential of hybrid parallelism for raycasting volume rendering at higher core-to-node ratios. The shared-memory parallelism used by the dynamically-scheduled, image-based decomposition in the raycasting phase scales well to high concurrencies per node (thousands of threads in the case of CUDA). In turn, decreasing the absolute number of nodes improves the communication performance during the compositing phase.

We anticipate that future multi-core CPU systems with more shared-memory parallelism will find a similar "sweet spot" in hybrid-parallel volume rendering performance as we observed on Longhorn. For instance, we expect that our hybrid/pthreads and hybrid/OpenMP implementations will benefit from additional shared-memory parallelism on the recently procured Cray XT5 system Hopper at NERSC, which will have dual-socket 12-core Opteron CPUs for 24 cores per node.

6 Conclusion and Future Work

The multi-core era offers new opportunities and challenges for parallel applications. Our study shows that for raycasting volume rendering, hybrid parallelism enables performance gains and uses less resources than a distributed-memory implementation on large supercomputers comprised of both multi-core CPUs and many-core GPUs. The advantages are reduced memory footprint, reduced MPI overhead, and reduced communication traffic. These advantages are likely to become more pronounced in the future as the number of cores per CPU increases while per-core memory size and bandwidth decrease.

Our study used `MPI_Alltoallv` to perform compositing. While we are encouraged by the results showing favorable hybrid-parallel performance in terms of

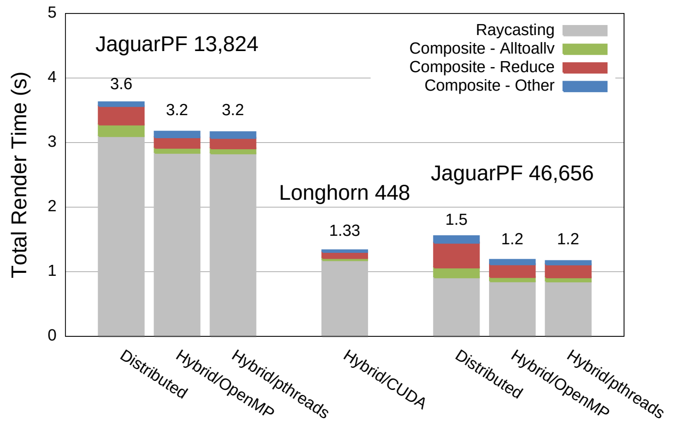

Fig. 10. Total render time for the 4608^3 dataset, with the hybrid/CUDA implementation on Longhorn bracketed by the strong scaling results from JaguarPF.

reduced number of messages, reduced message size, and faster overall runtime, it would interesting to see improvement in the context of compositing algorithms like binary swap and Radix-K. We leave this for future work.

Acknowledgment

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 through the Scientific Discovery through Advanced Computing (SciDAC) program's Visualization and Analytics Center for Enabling Technologies (VACET).

This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. Additionally, preliminary results for this work were collected using resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing the GPU cluster Longhorn that contributed to the research results reported within this paper.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[2] C. T. Silva, A. E. Kaufman, and C. Pavlakos, "PVR: High-Performance Volume Rendering," *Computing in Science and Engineering*, vol. 3, no. 4, pp. 18–28, 1996.

[3] J. Kniss, P. McCormick, A. McPherson, J. Ahrens, J. Painter, A. Keahey, and C. Hansen, "Interactive Texture-Based Volume Rendering for Large Data Sets," *IEEE Computer Graphics and Applications*, July/August 2001.

[4] 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 ICPP'09 Conference*, September 2009.

[5] M. Howison, E. W. Bethel, and H. Childs, "MPI-Hybrid Parallelism for Volume Rendering on Large, Multicore Clusters," in *Proceedings of Eurographics Parallel Visualization and Graphics*, Nörkopping, Sweden, May 2010.

[6] M. Levoy, "Display of Surfaces from Volume Data," *IEEE Computer Graphics and Applications*, vol. 8, no. 3, pp. 29–37, May 1988.

[7] R. A. Drebin, L. Carpenter, and P. Hanrahan, "Volume rendering," *SIGGRAPH Computer Graphics*, vol. 22, no. 4, pp. 65–74, 1988.

[8] A. Kaufman and K. Mueller, "Overview of Volume Rendering," in *The Visualization Handbook*, C. D. Hansen and C. R. Johnson, Eds. Elsevier, 2005, pp. 127–174.

[9] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh, "A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering," in *Proceedings of the 1993 Parallel Rendering Symposium*. ACM Press, October 1993, pp. 15–22.

[10] R. Tiwari and T. L. Huntsberger, "A Distributed Memory Algorithm for Volume Rendering," in *Scalable High Performance Computing Conference*, Knoxville, TN, USA, May 1994.

[11] K.-L. Ma, "Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures," in *PRS '95: Proceedings of the IEEE symposium on Parallel rendering*. New York, NY, USA: ACM, 1995, pp. 23–30.

[12] C. Bajaj, I. Ihm, G. Joo, and S. Park, "Parallel ray casting of visibly human on distributed memory architectures," in *VisSym'99 Joint EUROGRAPHICS-IEEE TVCG Symposium on Visualization*, 1999, pp. 269–276.

[13] P. Sabella, "A Rendering Algorithm for Visualizing 3D Scalar Fields," *SIGGRAPH Computer Graphics*, vol. 22, no. 4, pp. 51–58, 1988.

[14] C. Upson and M. Keeler, "V-buffer: visible volume rendering," in *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1988, pp. 59–64.

[15] J. Nieh and M. Levoy, "Volume Rendering on Scalable Shared-Memory MIMD Architectures," in *Proceedings of the 1992 Workshop on Volume Visualization*. ACM SIGGRAPH, October 1992, pp. 17–24.

[16] C. Müller, M. Strengert, and T. Ertl, "Optimized volume raycasting for graphics-hardware-based cluster systems," in *Proceedings of Eurographics Parallel Graphics and Visualization*, 2006, pp. 59–66.

[17] B. Moloney, M. Ament, D. Weiskopf, and T. Moller, "Sort First Parallel Volume Rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 99, no. PrePrints, 2010.

[18] 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.

[19] T. Peterka, D. Goodell, R. Ross, H.-W. Shen, and R. Thakur, "A Configurable Algorithm for Parallel Image-compositing Applications," in *Supercomputing '09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2009, pp. 1–10.

[20] W. Kendall, T. Peterka, J. Huang, H.-W. Shen, and R. Ross, "Accelerating and Benchmarking Radix-k Image Compositing at Large Scale," in *Proceedings of Eurographics Parallel Visualization and Graphics*, Nörkopping, Sweden, May 2010.

[21] G. Hager, G. Jost, and R. Rabenseifner, "Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes," in *Proceedings of Cray User Group Conference*, 2009.

[22] D. Mallón, G. Taboada, C. Teijeiro, J. Tourino, B. Fraguela, A. Gómez, R. Doallo, and J. Mourino, "Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures," in *16th European PVM/MPI Users' Group Meeting, (EuroPVM/MPI'09)*, September 2009.

[23] T. Peterka, R. Ross, H. Yu, K.-L. Ma, W. Kendall, and J. Huang, "Assessing improvements in the parallel volume rendering

pipeline at large scale," in *Proc. SC'08 Ultrascale Visualization Workshop*, Austin, TX, 2008.

[24] T. Fogal, H. Childs, S. Shankar, J. Krüger, D. Bergeron, and P. Hatcher, "Large data visualization on distributed memory multi-GPU clusters," in *Proceedings of High Performance Graphics*, 2010.

[25] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI – The Complete Reference: The MPI Core, 2nd edition*. Cambridge, MA, USA: MIT Press, 1998.

[26] D. R. Butenhof, *Programming with POSIX threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[27] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[28] NVIDIA Corporation, *NVIDIA CUDA^{TM} Programming Guide Version 3.0*, 2010, http://developer.nvidia.com/object/cuda_3_0_downloads.html.

[29] "The top 500 supercomputers," 2009, http://www.top500.org.

[30] J. K. Lawder and P. J. H. King, "Using Space-filling Curves for Multi-dimensional Indexing," in *Lecture Notes in Computer Science*, 2000, pp. 20–35.

PLACE PHOTO HERE

Mark Howison is an application scientist at Brown University's Center for Computation and Visualization and a computer systems engineer in Lawrence Berkeley National Laboratory's Visualization Group. His research interests include scientific computing, visualization, graphics, and parallel I/O. Howison has an MS in computer science from the University of California, Berkeley. Contact him at mhowison@brown.edu.

PLACE PHOTO HERE

E. Wes Bethel is a staff scientist at Lawrence Berkeley National Laboratory, where he conducts and leads research, development, and deployment activities in high-performance, parallel visual data exploration algorithms and architectures. Bethel has a PhD in computer science from the University of California, Davis. He's a member of ACM Siggraph and IEEE. Contact him at ewbethel@lbl.gov.

PLACE PHOTO HERE

Hank Childs is a computer systems engineer at Lawrence Berkeley National Laboratory and a researcher at the University of California, Davis. His research interests include parallel visualization and production visualization applications. Childs has a PhD in computer science from the University of California at Davis. Contact him at hchilds@lbl.gov.