Hybrid Parallelism

E. Wes Bethel

Computational Research Division
Lawrence Berkeley National Laboratory,
Berkeley, California, USA, 94720.

David Camp

Computational Research Division
Lawrence Berkeley National Laboratory,
Berkeley, California, USA, 94720.

Hank Childs

Computational Research Division
Lawrence Berkeley National Laboratory,
Berkeley, California, USA, 94720.

Christoph Garth

University of Kaiserslautern,
Kaiserslautern, Germany.

Mark Howison

Brown University,
Providence, RI, USA, 02912.

Kenneth I. Joy

University of California, Davis,
Davis, CA, USA, 95616.

David Pugmire

Oak Ridge National Laboratory,
Oak Ridge, TN, USA, 37831.

December 2012

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. Some of the research in this work used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Legal Disclaimer

Abstract

Hybrid parallelism refers to a blend of distributed- and shared-memory parallel programming techniques within a single application. This report presents results from two studies. They aimed to explore the thesis that hybrid parallelism offers performance advantages for visualization codes on multi-core platforms. The findings show that, compared to a traditional distributed-memory implementation, the hybrid parallel approach uses a smaller memory footprint, performs less interprocess communication, has faster execution speed, and, for some configurations, performs significantly less data I/O.

Preface

The material in this technical report is a chapter from the book entitled *High Performance Visualization—Enabling Extreme Scale Scientific Insight* [3], published by Taylor & Francis, and part of the CRC Computational Science series.

Contents

1 Introduction **5**

2 Hybrid Parallelism and Volume Rendering **6**
2.1 Background and Previous Work 6
2.2 Implementation 6
2.2.1 Shared-Memory Parallel Ray Casting 7
2.2.2 Parallel Compositing 8
2.3 Experiment Methodology 8
2.4 Results 9
2.4.1 Initialization 9
2.4.2 Ghost Data/Halo Exchange 9
2.4.3 Ray Casting 9
2.4.4 Compositing 11
2.4.5 Overall Performance 12

3 Hybrid Parallelism and Integral Curve Calculation **15**
3.1 Background and Context 15
3.2 Design and Implementation 16
3.2.1 Parallelize Over Seeds 16
3.2.2 Parallelize Over Blocks 17
3.3 Experiment Methodology 18
3.3.1 Factors Influencing Parallelization Strategy 18
3.3.2 Test Cases 18
3.3.3 Runtime Environment 18
3.3.4 Measurements 18
3.4 Results 19
3.4.1 Parallelization Over Seeds 19
3.4.2 Parallelization Over Blocks 19

4 Conclusion and Future Work **20**

1 Introduction

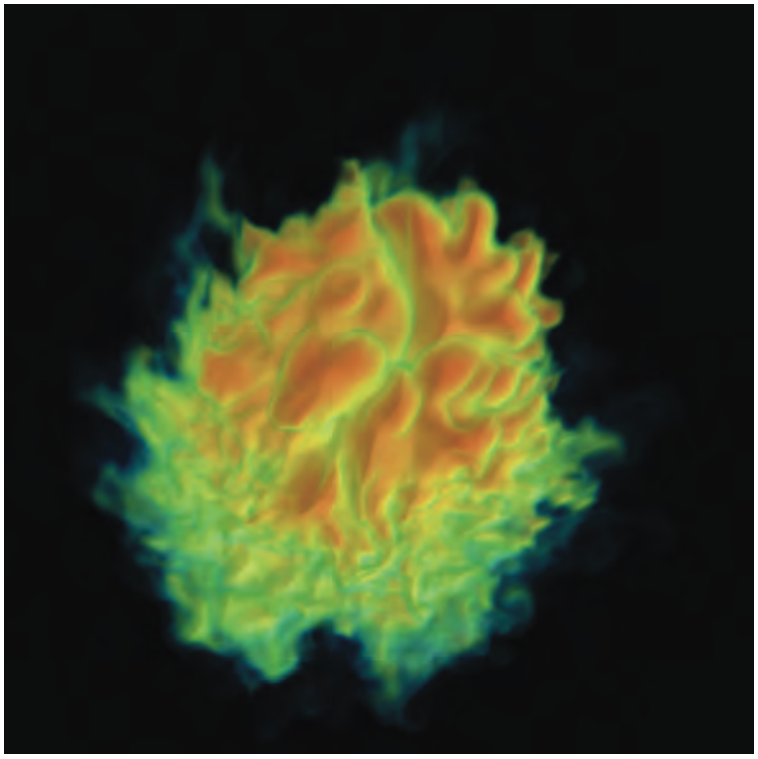

Figure 1: 4608^2 image of a combustion simulation result, rendered by hybrid parallel MPI+pthreads implementation running on 216,000 cores of the JaguarPF supercomputer. Image source: Howison et al., 2011 [15]. Combustion simulation data courtesy of J. Bell and M. Day (LBNL).

A *distributed-memory parallel* computer is made up of multiple nodes, with each node containing one or more cores. Each instance of a parallel program is called a task (or sometimes a *Processing Element* or PE). A pure distributed-memory program has one task for each core on each node of the computer. This is not necessary, however. Hybrid parallel programs have fewer tasks per node than cores. They make use of the remaining cores by using *threads*, which are lightweight programs controlled by the task. Threads can share memory amongst themselves and between the main thread associated with the task; allowing for optimizations that are not possible with distributed-memory programming. For example, consider a distributed-memory parallel computer with eight quad-core nodes. A pure distributed-memory program would have thirty-two tasks running and none of these tasks would make use of shared-memory techniques (although some cores would reside on the same node). A hybrid configuration could have eight tasks, each running with four threads, sixteen tasks, each running with two threads, or even configurations where the number of tasks and threads per node varies.

This report defines and uses the following terminology and notation. *Traditional parallelism*, or P^T, refers to a design and implementation that uses only MPI for parallelism, regardless of whether the parallel application is run on a distributed- or shared-memory system. *Hybrid parallelism*, or P^H, refers to a design and implementation that uses both MPI and some other form of shared-memory parallelism like POSIX threads [5], OpenMP [4], OpenCL [13], CUDA [10], and so forth.

The main focus of this report is to present results from two different experiments within the field of high performance visualization that aim to study the extent to which visualization algorithms can benefit from hybrid parallelism when applied to today's largest data sets and on today's largest computational platforms. The studies presented in this report use a P^H design,

whereby each MPI task will in turn invoke some form of shared-memory parallelism on multi-core CPUs and many-core GPUs.

One experiment studies a hybrid parallel implementation of ray casting volume rendering at extreme-scale concurrency. The other studies a hybrid parallel implementation of integral curve computation using two different approaches to parallelization. The material in this report consolidates information from earlier publications on hybrid parallelism for volume rendering [14, 15] and streamline/integral curve computations [7, 22]. Both of these studies show that a P^H implementation runs faster, uses less memory, and performs less communication and data movement than its P^T counterpart. In some cases, the difference is quite profound, and reveals many insurmountable obstacles if P^T paths continue to be used on exascale-class computational platforms.

There are several factors that influence scalability at extreme levels of concurrency. The cost of initialization itself may become a limiting factor. One of the studies in this report reveals that there is a cost, in terms of memory footprint, associated with each MPI task that grows at a nonlinear rate with the concurrency level. Another factor is the overhead associated with synchronizing processes. Traditional message-based approaches that have worked well in distributed-memory parallel environments may prove to be too costly when there are hundreds to thousands of cores per chip. More generally, communication patterns and associated overhead may likely suffer from similar scalability limits. Load balancing, which refers to the process of having each of the individual processes perform about the same amount of work, has been the subject of much research over the years in distributed-memory parallel environments. Adding in the complexity of scores, hundreds to thousands of cores per chip, adds to that complexity, which, in turn, can also limit scalability.

2 Hybrid Parallelism and Volume Rendering

2.1 Background and Previous Work

Volume rendering is a common technique used for displaying 2D projections of 3D sampled data [11, 16] and it is computationally, memory, and data I/O intensive. The study focuses on an P^Himplementation at extreme concurrencies in order to take advantage of multi- and many-core processor architectures.

This study's P^Himplementation makes use of a design pattern common in many parallel volume rendering applications that use a mixture of both object- and pixel-level parallelism [2, 17, 18, 25]. The design employs an object-order partitioning to distribute source data blocks to processors where they are rendered using ray casting [11,16,23,27]. Then, within a processor, an image-space decomposition, similar to Nieh and Levoy [19], is used to allow multiple rendering threads to cooperatively generate partial images that are later combined, via compositing, into a final image [11, 16, 27]. 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 most substantial difference between the work in this study and previous work in parallel volume rendering is the exploitation of P^H parallelism at an order of magnitude of greater concurrency than any previous studies. This work also performs an in-depth study to better understand scalability characteristics as well as potential performance gains of the P^Happroach.

2.2 Implementation

The study's parallel volume rendering implementation uses a design pattern similar to that in previous works (e.g., see [2,17,18,25]). Given a source data volume S and n parallel tasks, each task reads in $1/n$ of S, performs ray casting volume rendering on this data subdomain to produce a set of image fragments, and, 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 for storage. Figure 2 provides a block-level view of this organization.

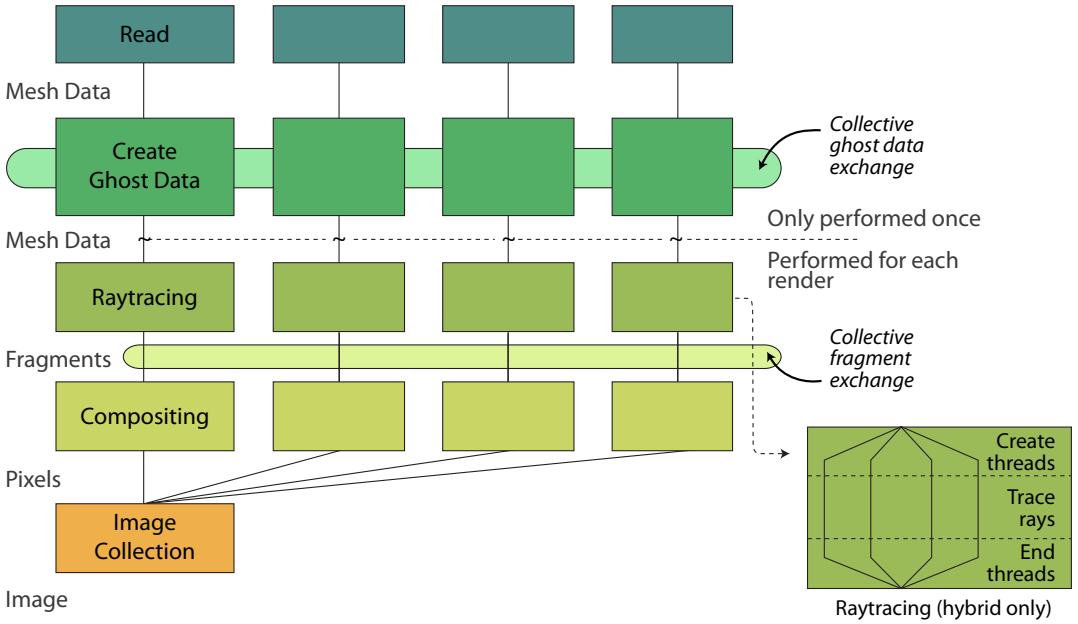

Figure 2: P^H volume rendering system architecture. Image courtesy of Mark Howison, E. Wes Bethel, and Hank Childs (LBNL).

The P^T implementation is written in C/C++, using the MPI [24] library. The portions of the implementation that are shared-memory parallel are written using a combination of C/C++ and either POSIX threads [5], OpenMP [8], or CUDA (version 3.0) [20] so that the study is comparing three P^H implementations referred to as hybrid/pthreads, hybrid/OpenMP, and hybrid/CUDA.

The P^T and P^H implementations differ in several key respects. First, the ray casting volume rendering algorithm is a serial process in the P^T implementation; this serial process is replicated across N processors. In contrast, in the P^H implementation, the ray casting algorithm runs in a shared-memory parallel fashion rather than as a serial process. Second, the communication topology in the compositing stage differs slightly between the P^T and P^H implementations—the details of which are the subject of Section 2.2.2. The third difference is how data is partitioned across the tasks. In the P^T implementation, each task loads and operates on a disjoint block of data. In the P^H 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 [19].

2.2.1 Shared-Memory Parallel Ray Casting

The implementation of ray casting volume rendering code follows Levoy's method [16]: first, compute the intersection of a ray with a data block, and then, compute the 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 [21]. 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.

Each P^T task invokes a serial ray caster that operates on its own disjoint block of data. Since this code is processing structured rectilinear grids, all data subdomains are spatially disjoint, and the z-coordinate for sorting during the subsequent composition step is the ray's entry point into the data block.

In contrast, the P^H tasks invoke a ray caster with shared-memory parallelism that consists of T threads, executing concurrently to perform the ray casting on a shared block of data. As in

Nieh and Levoy [19], the authors use an image-space partitioning: each thread is responsible for ray casting a portion of the image. In the pthreads and OpenMP ray casting implementations, the image-space partitioning is interleaved, with the image divided into many tiles that are distributed amongst the threads. The CUDA ray casting 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. There are a number of CUDA-centric issues and considerations with this implementation. For more information, see Howison et al. 2011 [15].

2.2.2 Parallel Compositing

In comparison to parallel image compositing methods, this particular implementation uses an approach that is somewhat different. 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 ray casting 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 [21]. The final image is then gathered on the root task. (See the original studies for additional details [14, 15].)

The P^H and P^T compositing differ in one key way that results in fewer, but larger messages in the P^H implementation, which in turn, results in P^H having a better performance. In both the P^H and P^T implementations, each of the MPI tasks will participate in the fragment exchange communication process. Because there are far fewer MPI tasks in the P^H implementation, those MPI tasks will exchange fewer and larger messages. In the P^H implementation, one thread gathers fragments from all other threads in the shared-memory parallel CPU or GPU and performs the communication rather than having all threads participate in the communication. The overall effect of this design choice is an improvement in communication characteristics, as presented later in Section 2.4.4.

2.3 Experiment Methodology

The study's methodology is designed to test the hypothesis that a P^H implementation exhibits better performance and resource utilization than the P^T implementation.

The study uses two systems, a large Cray XF5 system, JaguarPF, located at Oak Ridge National Laboratory, and a large GPU cluster, Longhorn, located at the Texas Advanced Computing Center. In 2009, JaguarPF was ranked number one on the list of TOP500 fastest supercomputers with a peak theoretical performance of 2.3 petaflops [26]. 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.

Longhorn has 256 host nodes with dual-socket, quad-core Intel Nehalem 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. The study treats 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, the study positions the FX5800 relative to the Opteron in terms of its actual observed runtime for this particular application rather than relying on an *a priori* architectural comparison.

The study includes three types of scalability experiments:

- **strong scaling**, in which the image size is fixed at 4608^2 and the data set size at 4608^3 (97.8 billion cells) for all concurrency levels;
- **weak-data set scaling**, with the same fixed 4608^2 image, but also, a data set size increasing with concurrency up to 23040^3 (12.2 trillion cells) at 216,000-way parallel; and

- **weak scaling**, in which both the image and the data set increase in size up to 23040^2 and 23040^3, respectively, at 216,000-way parallel.

Note that at the lowest concurrency level, all three cases coincide with a 4608^2 image and 4608^3 data set size. The source data for this experiment consisted of an output from a combustion simulation code that aims to perform a laboratory-scale modeling of a flame. Figure 1 shows a sample image produced by these runs. (See the original studies for additional details of the experimental methodology configuration, including details of the source data used in the experiment, how data is partitioned across processors on each platform, and data/image sizes for each of the three scaling experiments [14, 15].)

2.4 Results

The study compares the cost of MPI runtime overhead and corresponding memory footprint in Section 2.4.1; the absolute amount of memory required for data blocks and ghost (halo) exchange in Section 2.4.2; the scalability of the ray casting and compositing algorithms in Section 2.4.3 and Section 2.4.4; and the communication resources required during the compositing phase in Section 2.4.4. Section 2.4.5 concludes with a comparison of results from the six-core CPU system and the many-core GPU system to understand how the balance of P^H vs. P^T parallelism affects the overall performance.

2.4.1 Initialization

Because there are fewer P^H tasks, they incur a smaller aggregate memory footprint for the MPI runtime environment and program-specific data structures that are allocated per task. Table 1 shows the memory footprint of the program as measured directly after calling `MPI_Init` and reading in command-line parameters.[1] Memory usage was sampled only from tasks 0 through 6, but those values agreed within 2% of each other. Therefore, the per-task values reported in Table 1 are from task 0 and the per-node and aggregate values are calculated from the per-task value.[2]

The P^T implementation uses twelve MPI tasks per node while the P^H one uses only two. The P^H implementation's two MPI tasks per node each go six-way parallel, so that all cores on both node's multi-core CPUs are fully occupied in both the P^H and P^T implementations. At a 216,000-way concurrency, the runtime overhead per P^T task is more than 2× the overhead per P^H task. The per-node and aggregate memory usage is 6× larger for the P^T implementation, because it uses 6× as many tasks. Thus, the P^T implementation uses nearly 12× as much memory per-node and in-aggregate than the P^H one for initializing the MPI runtime at a 216,000-way concurrency.

2.4.2 Ghost Data/Halo Exchange

Two layers of ghost data are required in the ray casting phase: the first layer is for trilinear interpolation of sampled values, and the second layer is for computing the gradient field using central differences (gradients are not precomputed for this data set). In the P^H configuration, since the source data is partitioned into fewer, larger blocks, the P^H version requires less exchange and storage of ghost data by roughly 40% across all concurrency levels and for both strong and weak scaling compared to the P^T version—Figure 3 illustrates these results.

2.4.3 Ray Casting

All of the scaling studies demonstrate good scaling for the ray casting phase, since no message passing is involved (Fig. 4). The authors used trilinear interpolation for data sampling along

[1] The authors collected the `VmRSS`, or "resident set size," value from the `/proc/self/status` interface.

[2] In Table 1, the value in the Per Task column multiplied by twelve, the number of cores per node, which consists of two, six-core CPUs, does not always equal the value in the Per Node column due to the round-off error that results from presenting values as integral numbers of MB or GB.

Table 1: Comparison of memory usage at MPI Initialization for P^H and P^T volume rendering implementations. At 216,000-way concurrency, the P^T implementation uses twelve times the memory of the P^H version.

CPU cores	Implementation	Tasks	Per Task (MB)	Per Node (MB)	Agg. (GB)
1728	P^H	288	67	133	19
1728	P^T	1728	67	807	113
13824	P^H	2304	67	134	151
13824	P^T	13824	71	857	965
46656	P^H	7776	68	136	518
46656	P^T	46656	88	1055	4007
110592	P^H	18432	73	146	1318
110592	P^T	110592	121	1453	13078
216000	P^H	36000	82	165	2892
216000	P^T	216000	176	2106	37023

the ray as well as a Phong-style shader for these runs and timings. The final ray casting time is essentially the runtime of the thread that takes the most integration steps. This behavior is entirely dependent on the view. The study's approach, which is aimed at understanding "average" behavior, uses ten different views and reports an average runtime.

In the strong scaling study, performance for the ray casting phase proves to be linear up to a 216,000-way concurrency with P^T (see Fig. 4). The P^H implementation exhibited different scaling behavior because of its different decomposition geometry: the P^T data blocks had a perfectly cubic decomposition, but the P^H version uses $1 \times 2 \times 3$ cubic blocks, resulting in a larger rectangular block than in the P^T version (see Howison et al. 2011 [15], Table 4, for additional details). The smaller size of the GPU cluster limited the feasible concurrencies 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 ray casting time. At lower concurrencies, this interaction benefited the P^H implementation by as much as 11% (see Howison et al. 2011 [15], Table 4). At higher concurrencies, the trend flips and the P^T implementation outperforms the P^H one by 10% in the ray casting phase. The authors expected that if they were able to run the P^H implementation with cubic-shaped data blocks (such as $2 \times 2 \times 2$ on an eight-core system), the ray casting phase of both implementations would scale identically. They also note that at 216,000 cores, ray casting is less than 20% of the total runtime (see Fig. 6), and the P^H implementation is over 50% faster because of gains in the compositing phase, which is the subject of the next subsection.

For weak scaling, the P^H implementation maintained 80% scalability out to 216,000 cores. Overall ray casting 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 a higher concurrency, since each core is operating on a smaller portion of the overall view frustum, which accounts for the 20% degradation.

The P^T result at a 216,000-way concurrency appears (misleadingly) to be superlinear, but that is because the authors could not maintain the data size per core. Although they could maintain it for the weak-data set 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 ray casting phase) to overflow. To accommodate the fragment buffer, the authors scaled down to a data size of 19200^3 (7.1 trillion cells), instead of the 23040^3 data size (12.2 trillion cells) used for the

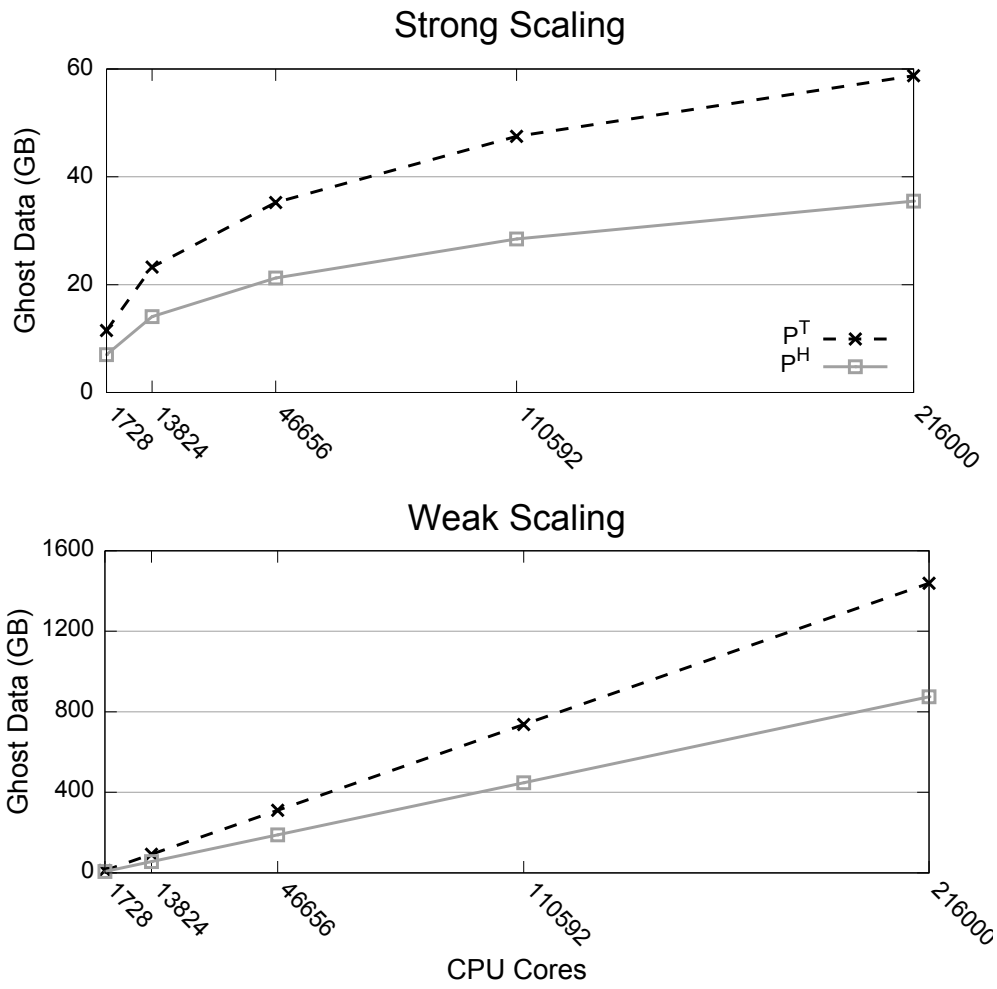

Figure 3: Because it uses fewer and larger data blocks, the P^H volume renderer requires 40% less memory for ghost data than the P^T implementation. As a result, the P^H code uses less interprocessor communication for ghost data exchange than the P^T version. Image source: Howison et al., 2011 [15].

P^H runs. The results show 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-data set scaling study, the expected scaling behavior is neither linear nor constant, since the amount of work for the ray casting phase is dependent on both data size and image size. With a varying data size but fixed image size, the scaling curve for weak-data set scaling should lie between those of the pure weak and pure strong scaling, which is what they observe. Overall, 216,000-way concurrency was 10× faster than 1,728-way concurrency.

2.4.4 Compositing

Above 1,728-way concurrency, the study reveals that compositing times are systematically better for the P^H implementation (see Howison et al. [15], Figure 7). The compositing phase has two communication costs: (1) the `MPI_Alltoallv` call that exchanges fragments from the task where they originated during ray casting to the compositing task that owns their region of image space; and (2) the `MPI_Reduce` call that reduces the final image components to the root task for assembly and output to a file or display. (See Fig. 6 for a breakdown of these costs.) During the fragment exchange, the P^H implementation can aggregate the fragments in the memory,

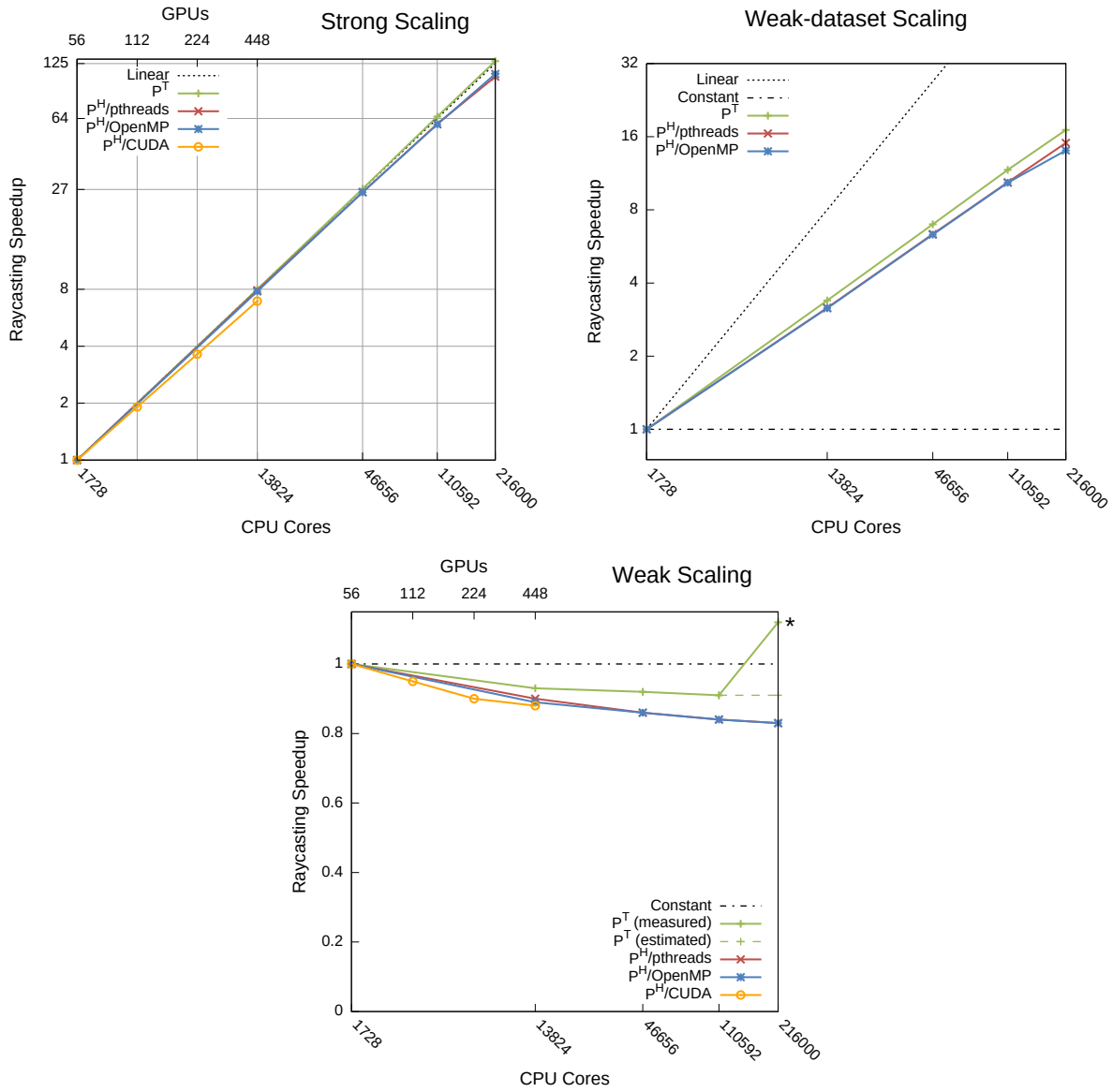

Figure 4: The speedups (referenced to 1,728 cores) for both the ray casting phase and the total render time (ray casting and compositing). The ray casting speedup is linear for the P^T version, but is sublinear for the P^H version: this effect is caused by the difference in decomposition geometries (cubic vs. rectangular). Image source: Howison et al., 2011 [15].

shared by all worker threads, in this case six threads, and therefore, it uses on average about 6× fewer messages than the P^T implementation (see Fig. 5). In addition, the P^H 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 P^H tasks participate in the image reduction, which leads to better performance.

2.4.5 Overall Performance

In the strong scaling study at a 216,000-way concurrency, the best compositing time with P^H (0.35 seconds, 4500 compositors) was 3× faster than with P^T (1.06 seconds, 6750 compositors). Furthermore, at this scale, compositing time dominated rendering time, which was roughly 0.2 seconds for both implementations. Thus, the total render time was 2.2× faster with P^H (0.56

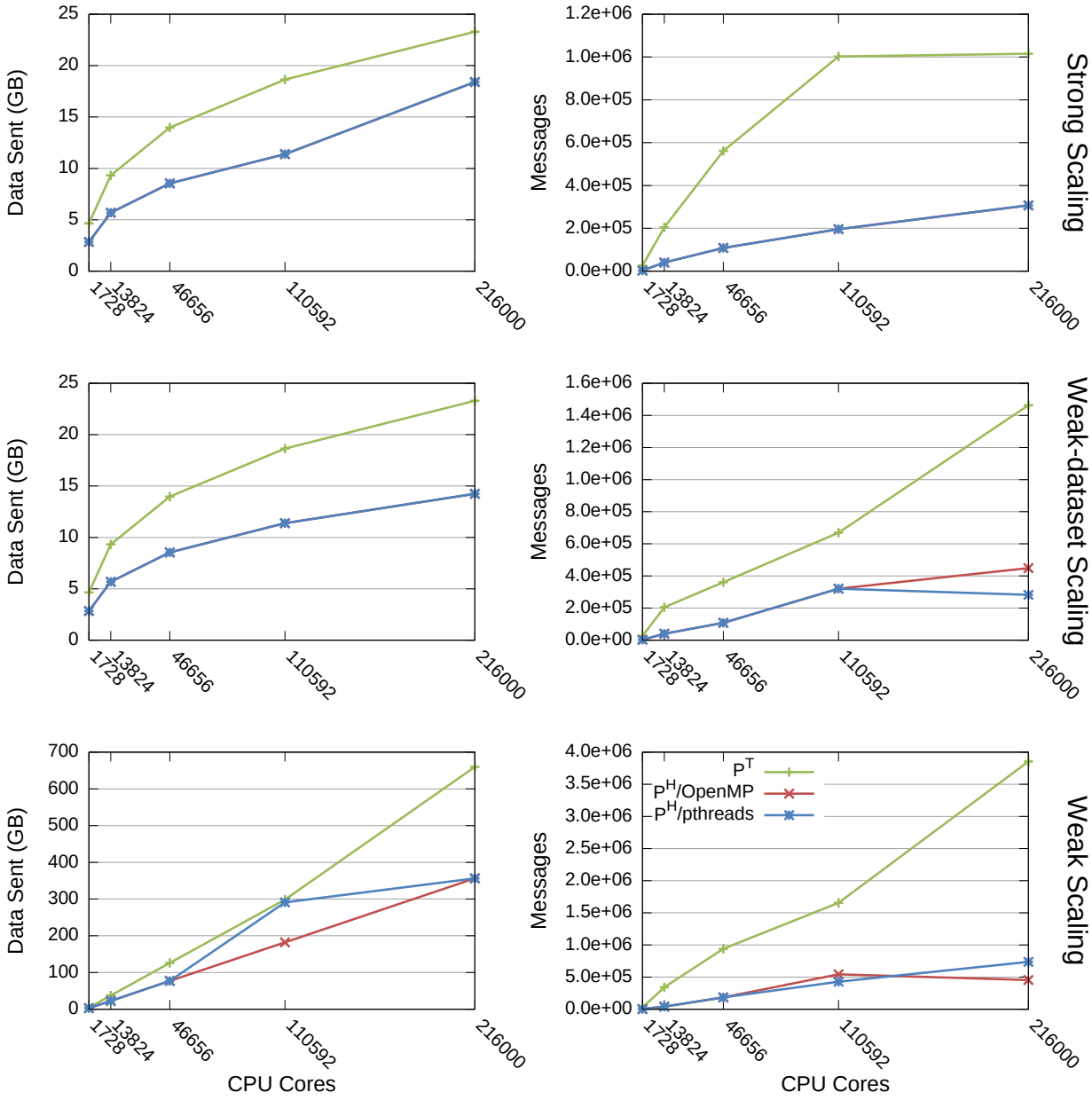

Figure 5: Comparison of the number of messages and total data sent during the fragment exchange in the compositing phase for P^H and P^T runs. The P^H version uses fewer messages and requires less total communication during compositing than the P^T implementation. Image source: Howison et al., 2011 [15].

seconds vs. 1.25 seconds). Overall, the strong scaling study shows that the advantage of P^H over P^T becomes greater as the number of cores increases (Fig. 6), primarily due to the P^H's faster compositing time.

For weak-data set and weak scaling, P^H still shows gains over the P^T, but they are less pronounced because the ray casting phase dominates. Although the 216,000-way breakdown for weak scaling looks like it favors P^T, this is actually an artifact of the reduced data size (19200^3); a reduced size data set was required for the P^T implementation to avoid out-of-memory errors. Comparing an estimated value for a 23040^3 P^T data size suggests that the P^H implementation

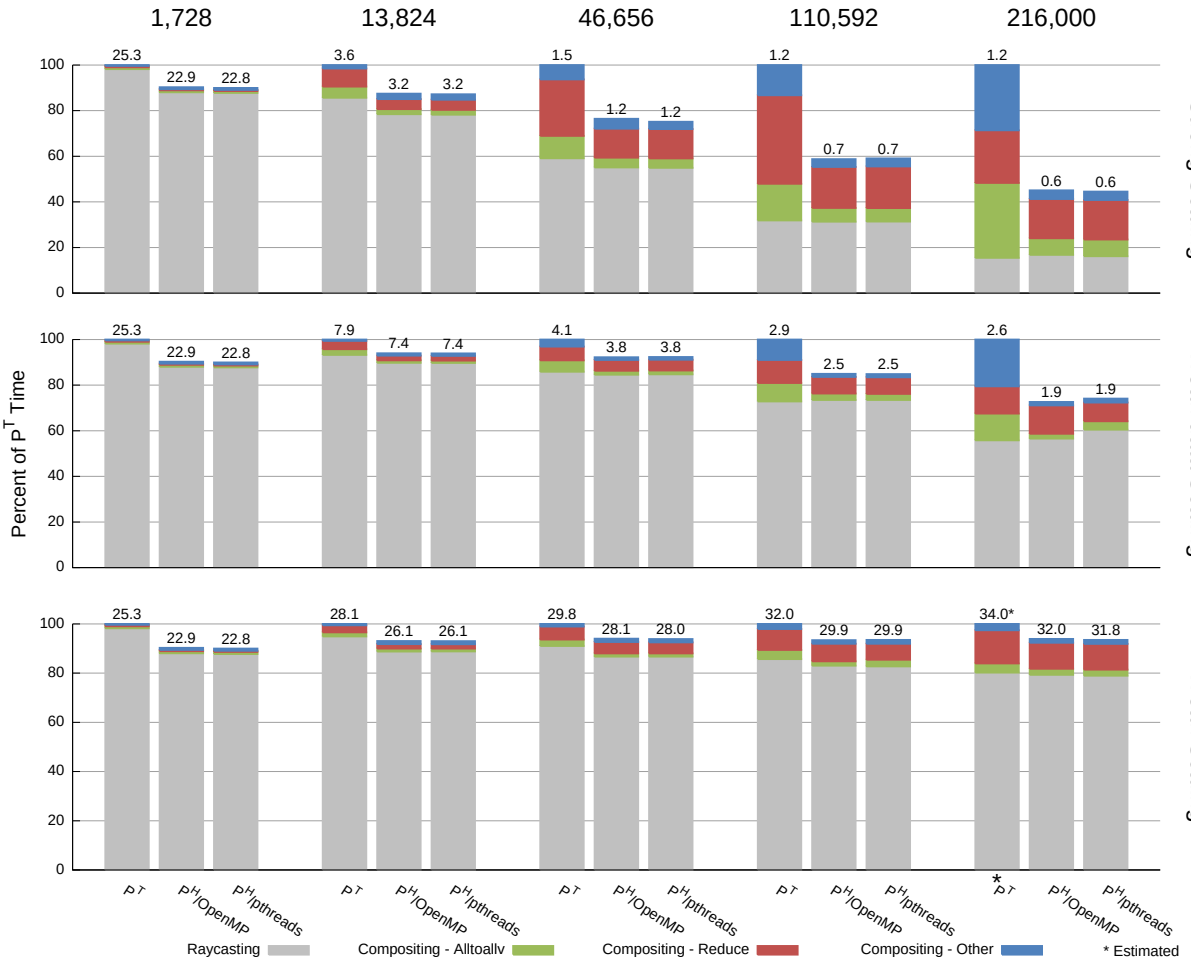

Figure 6: Total render time in seconds split into ray casting and compositing components and normalized at each concurrency level as a percentage of the distributed-memory time. "Compositing-Other" includes the time to coordinate the destinations for the fragments and to perform the over operator. Image source: Howison et al., 2011 [15].

would be slightly faster.

On Longhorn, using 448 GPUs and processing the 4608^3 data set, the hybrid/CUDA implementation averaged 1.18 seconds for ray casting and exhibited a minimum compositing time of 0.15 seconds (with a median of 0.19 seconds). The ray casting performance positions this run close to the 46,656-way run on JaguarPF (see Howison et al. 2011 [15], Fig. 10). However, the compositing time for the hybrid/CUDA run on Longhorn (0.15 seconds) is half that of hybrid-memory runs at the 46,656-way concurrency on JaguarPF (0.31–0.33 seconds).

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

Figure 7: Integral curve computation lies at the heart of visualization techniques like streamlines, which are very useful for seeing and understanding complex flow-based phenomena. This image shows an example from computational thermal hydraulics. Algorithmic performance is a function of many factors, including the characteristics of the flow field in the underlying data set. Image courtesy of Hank Childs and David Camp (LBNL).

3 Hybrid Parallelism and Integral Curve Calculation

3.1 Background and Context

There are two primary ways of performing a parallel IC computation: parallelize over data blocks (POB), and parallelize over seeds (POS). In POB, data is divided and distributed as evenly as possible across all processors. Each processor computes ICs that fall within the data block it owns. As an IC leaves one data block, a processor must communicate the IC to another processor, where the IC computation will continue. In POS, the IC's themselves are distributed as evenly as possible among processors, and each processor computes an entire IC, loading data blocks as needed.

This study focuses on how each of these two classes of algorithms differ between P^T and P^H implementation and presents the results from a study that explores the characteristics of each [7]. Considering limits to scalability, this study evaluates performance in terms of absolute runtime, the amount of data moved at runtime between processors, the amount of I/O required by each configuration, and load balance characteristics.

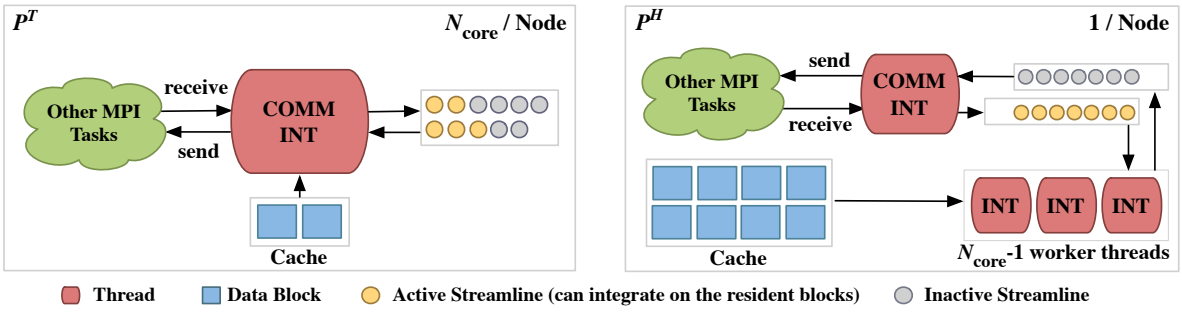

Figure 8: Comparison of P^T(left) and P^H(right) implementations of the *parallelize-over-seeds* algorithm. In the P^Tversion, each task performs the IC computation, an integration (INT), and manages its own cache by loading blocks from a disk (I/O), whereas N_{core} worker threads share a cache in the P^Himplementation. In the P^Hversion, multiple I/O threads manage the cache and observe which ICs can (*active*) and cannot (*inactive*) continue with the resident data blocks. Future blocks to load are determined from the list of inactive ICs. MPI communication, not shown here, is limited to gathering results. Image source: Camp et al., 2011 [7].

3.2 Design and Implementation

3.2.1 Parallelize Over Seeds

The parallelize-over-seeds (POS) algorithm assigns $1/n$ of the IC seeds to each parallel task. Each task will then perform a numerical integration over its seed set, loading data blocks as needed. In general, the POS algorithm does not require any inter-task communication at run-time, although it may incur redundant data block loads if the ICs in different tasks enter the same data block. Figure 8 provides an overview of both $P^T$and P^Himplementations of this algorithm.

In the P^Tversion, a single thread, corresponding to the task's process, performs all operations: data block I/O and IC integration. There is a single queue containing the ICs that are owned by a task. A task will begin work on the IC at the head of the queue, load any data blocks needed for the IC's current location, and continue integration until the IC exits all data blocks currently in the task's cache. At that point, that IC goes to the tail of the queue, and processing resumes on the IC now at the head of the queue. This process repeats until all ICs have been integrated.

In contrast, the P^Himplementation maintains ICs in two sets, or queues. ICs in the *active* set can be integrated using the blocks currently residing in the cache, while *inactive* ICs require access to data blocks not resident in the cache for computation to proceed. The P^Himplementation uses two pools of threads for execution. In the first thread pool, I/O threads identify which data blocks need to be loaded to satisfy the needs of the IC calculation in the inactive set, and then, initiate I/O if there is room in the cache. After a block is loaded, the ICs waiting on it are migrated to the active set. In the second thread pool, a worker thread fetches ICs from the active set, performs integration on each one using the cached data blocks, and then retires them to the inactive set when the IC exits all currently loaded data block domains. ICs for which integration has been completed, are sent to a separate list and the algorithm terminates once both active and inactive sets are empty. Access to active and inactive sets, as well as the block cache, is synchronized through standard mutex and condition variable primitives.

The performance of the POS scheme depends primarily on the data loads and cache size N_{blocks}; if the cache is too small, blocks must be repeatedly brought in from external storage. Recent work has studied the effects of alternate strategies for caching data blocks in systems with an extended memory hierarchy [6]. The idea is to expand the data block cache into a two-level hierarchy that would include both primary memory, as discussed here, as well as node-local

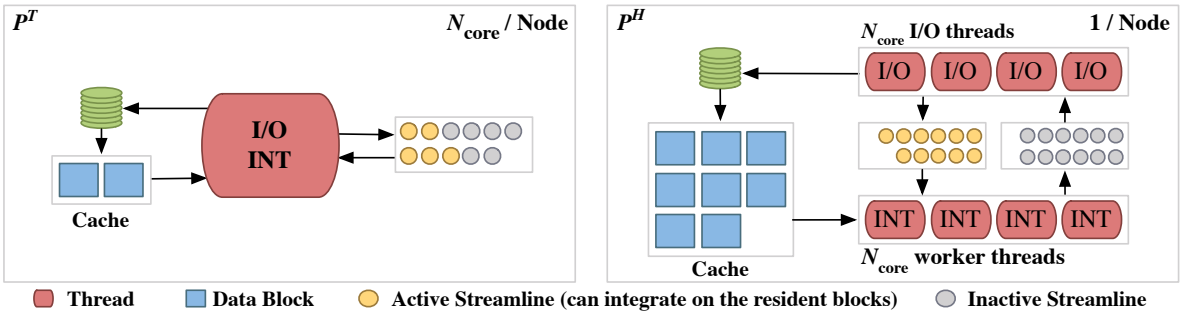

Figure 9: Comparison of P^T(left) and P^H(right) implementations of the *parallelize over blocks* algorithm. In the P^Tversion, each task performs the IC computation, an integration (INT), and communicates with other tasks to send and receive streamlines as they leave or enter different data partitions (COMM). In the P^Hversion, $N_{\text{core}} - 1$ worker threads perform IC computation over a set of data blocks. There is one additional thread that performs both IC computation, as well as communication operations. I/O is limited to an initial load of data blocks assigned to each task. Image source: Camp et al., 2011 [7].

storage devices, such as a local hard drive or a local solid-state drive.

The P^Himplementation has three main advantages over the P^Tversion: (1) the P^Hversion will have a larger shared cache and will perform less redundant I/O; (2) when a significant number of ICs are inside the same data block, each P^Ttask must load that block separately, whereas the P^Hversion will perform only one read and immediately share it among its threads; and, (3) since I/O and IC integration are performed by separate thread pools, those two operations can execute asynchronously and simultaneously—algorithm performance will likely improve as long as I/O and IC integration can proceed concurrently.

3.2.2 Parallelize Over Blocks

The parallelize-over-blocks (POB) algorithm assigns $1/n$ of the data blocks to each parallel task. Seed points for each IC are distributed to the tasks that own the data block containing the location of the seed. Tasks then perform numerical integration of each IC. When an IC exits a data block owned by a task, the IC is sent to the task that owns the data block where the IC will go next. (The processing details are explained in the original study [7].) Figure 9 shows an overview of both the $P^T$and P^Himplementations for this algorithm.

The POB algorithm performs minimal I/O: before integration commences, every task loads all blocks assigned to it, leveraging maximal parallel I/O bandwidth. The trade-off is a recurring inter-task communication phase, where ICs are exchanged between tasks as they migrate through the computational domain during integration.

As in POS algorithm, the P^Tversion consists of a single thread that maintains a set of ICs to integrate. Each thread performs integration of ICs that lie within the spatial region of data blocks in the cache.

The P^Himplementation maintains ICs in two queues. Newly received ICs are stored in the *active* queue, where workers fetch from it, integrate, and put ICs in the *inactive* queue when they exit the data blocks loaded by this task. From there, the ICs are sent off to other tasks by the supervisor thread, or retired to a separate list when complete. All tasks' supervisor threads also maintain a global count of active and complete ICs to determine when to terminate. Synchronization between threads is performed as in the POS implementation.

3.3 Experiment Methodology

3.3.1 Factors Influencing Parallelization Strategy

The parallel *IC* problem is complex and challenging. To design an experimental methodology that provides robust coverage of different aspects of algorithmic behavior (some of which is data set dependent), one must take into account several factors that influence parallelization strategy in designing an effective and robust performance test design methodology.

3.3.2 Test Cases

To cover a wide range of potential problem characteristics, this study uses four tests that address all combinations of seed set size (small or large) and seed point distribution (sparse or dense) for each of the three data sets.

The three data sets come from computational applications to astrophysics, fusion, and thermal hydraulics.

In the thermal hydraulics simulation, twin inlets pump water into a box, with a temperature difference between the water inserted by each inlet. Eventually, the water exits through an outlet. The mixing behavior and the temperature of the water at the outlet are of interest. Suboptimal mixing can be caused by long-lived recirculation zones that effectively isolate certain regions of the domain from heat exchange. The simulation was performed using the Nek5000 code [12] on an unstructured grid comprised of twenty-three million hexahedral elements. Streamlines are seeded according to two application scenarios. First, sparse seeds are distributed uniformly through the volume to show areas of high velocity, areas of stagnation, and areas of recirculation. Second, seeds are placed densely around one of the inlets to examine the behavior of particles entering through it. The resulting streamlines illustrate the turbulence in the immediate vicinity of the inlet. Small seed sets contain 1,500 seed points with an integration time of 12 units and the large sets consist of 6,000 seed points propagated for 3 time units. Figure 7 shows the streamlines computed for this data set.

3.3.3 Runtime Environment

The study conducted all tests on the NERSC Cray XT4 system *Franklin*. The 38,288 processor cores available for scientific applications are provided by 9,572 nodes, equipped with one quad-core AMD Opteron processor and 8GB of memory (2GB per core). Compute nodes are connected through HyperTransport for high performance, low-latency communication for MPI and SHMEM jobs. This platform supports parallel I/O, via a Lustre file system, which provides access to approximately 436TB of user disk space.

The authors implemented the two P^H algorithms into the VisIt [1, 9] visualization system, which is available on Franklin and routinely used by application scientists. P^T variants of both POB and POS algorithms were already implemented in recent VisIt releases and were also instrumented to provide the measurements discussed below.

The authors ran each benchmark run using 128 cores (32 nodes). For the P^T algorithm tests, they used 128 MPI tasks, one per core. The P^H test configuration was such that the total number of worker threads over all nodes was 128. One configuration, for example, used 32 MPI tasks, one per node, with each MPI task, in turn, spawning four worker threads. Note, the additional I/O or communication thread running per-MPI task in the P^H approach is not counted in this scheme; here, the study focuses on employing a constant number of worker threads for performing actual integration work to determine the impact of P^H.

3.3.4 Measurements

While absolute runtime is important, the authors added an instrumentation code so as to obtain more fine-grained information about different algorithmic steps. The steps include, for instance, the start and end of integration for a particular *IC* for worker threads, the begin and end of

an I/O operation for corresponding threads, and time spent performing communication using MPI calls. Timestamps are taken as elapsed wall-clock time since the start of the MPI task. These event logs are carefully implemented to a have negligible impact on the overall runtime behavior.

The pure integration time, T_{int}, represents the actual computational workload and it is the sum of all times taken by the worker threads to perform the IC computation. This time should be almost independent across all runs for a specific test case, since the integration workload in terms of the number of integration steps taken over all ICs is identical in each case. Similarly, $T_{\text{I/O}}$ and T_{comm} contain time spent doing I/O and communication, respectively. To gain a deeper insight into the role of the block cache, the authors count the number of blocks loaded, N_{load}, and purged, N_{purged}. The amount of MPI communication between tasks in bytes is shown as N_{comm}. Finally, as a derived measure of efficiency, they examine the integration ratio, R_{int}, as the fraction of the total algorithm runtime that was used to compute ICs.

In total, twelve tests of the form X(YZ) were run per algorithm in hybrid and non-hybrid variants, where X indicates the data set (**A**stro, **T**hermal Hydraulics, **F**usion), Y denotes the seed set size (**L**arge or **S**mall), and Z denotes the seed set density (**S**parse or **D**ense).

3.4 Results

Overall, the results show that for each of the two parallelization approaches, P^Hhas better performance characteristics than its P^Tcounterpart. However, the reasons for the performance differences vary between the different parallelization approach.

3.4.1 Parallelization Over Seeds

A P^H POS algorithm has three fundamental advantages:

Larger data cache. Both the $P^H$and P^Talgorithms use a memory-based cache to store data blocks/domains for later reuse to avoid performing redundant I/O. For the P^Hcase, the cache is shared amongst worker threads, hence, the worker threads can be larger than the P^Tcache by an amount proportional to the threading factor. For these experiments, the P^Hcache was four times larger than the P^Tcache, though cache size is a runtime, tunable parameter. Figure 10 shows that in all cases, the P^Hversion loads fewer data blocks than its P^Tcounterpart for each given problem.

Avoids redundant I/O. For the P^Talgorithm, when multiple tasks on the same node need to access the same data block, they must each read the same data block from disk. In effect, they are performing redundant loads of the same data block into the same node's memory. In the P^Hcase, only a single read is required and the data block is shared between all threads. This use case occurs frequently when the seed points are densely located in a small region. On average, the P^Hversion loads 64% less data than the P^Tversion. Figure 10 is a graph showing the number of data block loads.

Better load balancing. The time to calculate each IC varies from curve to curve because of the data dependent nature of the advection step. The study refers to ICs that take longer to compute as "slow ICs." Since ICs are permanently assigned to tasks, the tasks that process slow ICs will take longer to execute. Towards the end of the calculation, the tasks with slow ICs will still be executing, while the tasks with fast ICs will be finished, meaning there is a relatively poor load balance. Figure 11 shows that the P^Himplementation has less load imbalance between tasks. With the P^Himplementation, a larger number of ICs are shared between the worker threads, which creates a more even distribution of slow ICs. For example, if a task in the P^Tcase received many slow ICs, there is only one thread to handle them. In the P^Hcase, there will be more worker threads to advance these slow ICs.

3.4.2 Parallelization Over Blocks

A P^H POB algorithm has two fundamental advantages:

Better load balancing. Because data is partitioned over tasks, the only task that can advance a given IC is the task that owns the data block in which the IC resides. When many ICs traverse the same block, the corresponding task becomes a bottleneck. With the P^Halgorithm, more tasks can be used to relieve the bottleneck. Figure 13 illustrates this point. The P^Timplementation has only two tasks working on the longest ICs, which translates to two cores. The P^Himplementation also has two tasks working on these ICs, but since each task has four worker threads, that equates to a total of eight worker threads vs. two worker threads for the P^Timplementation. The additional workers allow the P^Hcase to finish more quickly. This factor can be quantified by measuring the percentage of time each core spends doing integration. The integration ratio R_{int} is very low for both implementations, but it is higher for the P^Himplementation.

Less communication. The communication cost of moving ICs between MPI processes is much lower in most of the P^Hcases of the POB method, which can be seen in the communication time and data transmitted between tasks (see Fig. 12). The P^HPOB implementation has four times fewer tasks than P^T. Since each P^Htask holds four times more data, it is less likely to send the IC to another task, which results in less overall inter-task communication.

4 Conclusion and Future Work

The two studies presented in this report, both of which compare the performance of traditional MPI-only with MPI-hybrid implementations of two staple visualization algorithms, reveal a consistent theme: on current multi- and many-core platforms, the MPI-hybrid design and implementation enjoys clear performance advantages.

First, both MPI-hybrid implementations require a significantly smaller memory footprint at initialization. Per-chip MPI overhead, in the form of internal buffers and so forth, expands as a function of the number of per-core MPI tasks, and, as the volume rendering study shows, a difficult-to-predict scaling factor that is likely related to how vendors implement MPI. Extrapolating from the volume rendering study results shown here, it seems clear that continuing along the path of an MPI-only implementation will eventually exhaust all available memory at extreme concurrency simply for the application initialization. The implication is clear: future high performance visual data analysis and exploration applications must, by necessity, use some form of hybrid-parallel design to make effective use of computational platforms, comprised of multiple or many cores per chip.

Second, the MPI-hybrid implementations perform significantly less inter-chip communication. The reasons for the lighter communication load vary, depending on the algorithm, and in some cases the particular problem or data set. Nonetheless, one common trait that results in less communication is the fact that the hybrid-parallel implementations use multiple processing threads, typically one per core, that all have access to a common shared memory. Communication between processing threads on a CPU that has global shared memory visible to all its cores takes place through that memory. And, depending on the underlying shared-memory programming model, that communication can take place without any operating system or MPI overhead like message buffering and so forth. A second common trait concerns the use of ghost zones. With a data decomposition consisting of smaller and more data partitions, there is more surface area compared to a decomposition that results in fewer and larger data partitions. Less surface area means there is less information—ghost zones—that needs to be communicated during the course of processing. The MPI-only configuration results in more and smaller data partitions compared to the MPI-hybrid configurations.

In the future, all trends suggest computational platforms comprised of an increasing number of cores per chip. One unknown is whether or not those future architectures will continue to support a shared memory that is visible to all cores. The present hybrid-parallel implementations perform well because all threads have access to a single shared memory on a CPU chip. If future architectures eliminate this shared memory, future research will need to explore alternative algorithmic formulations and implementations that both exploit available architectural traits as

well as achieve low memory footprint utilization and reduced communication when compared to traditional, MPI-only designs and implementations.

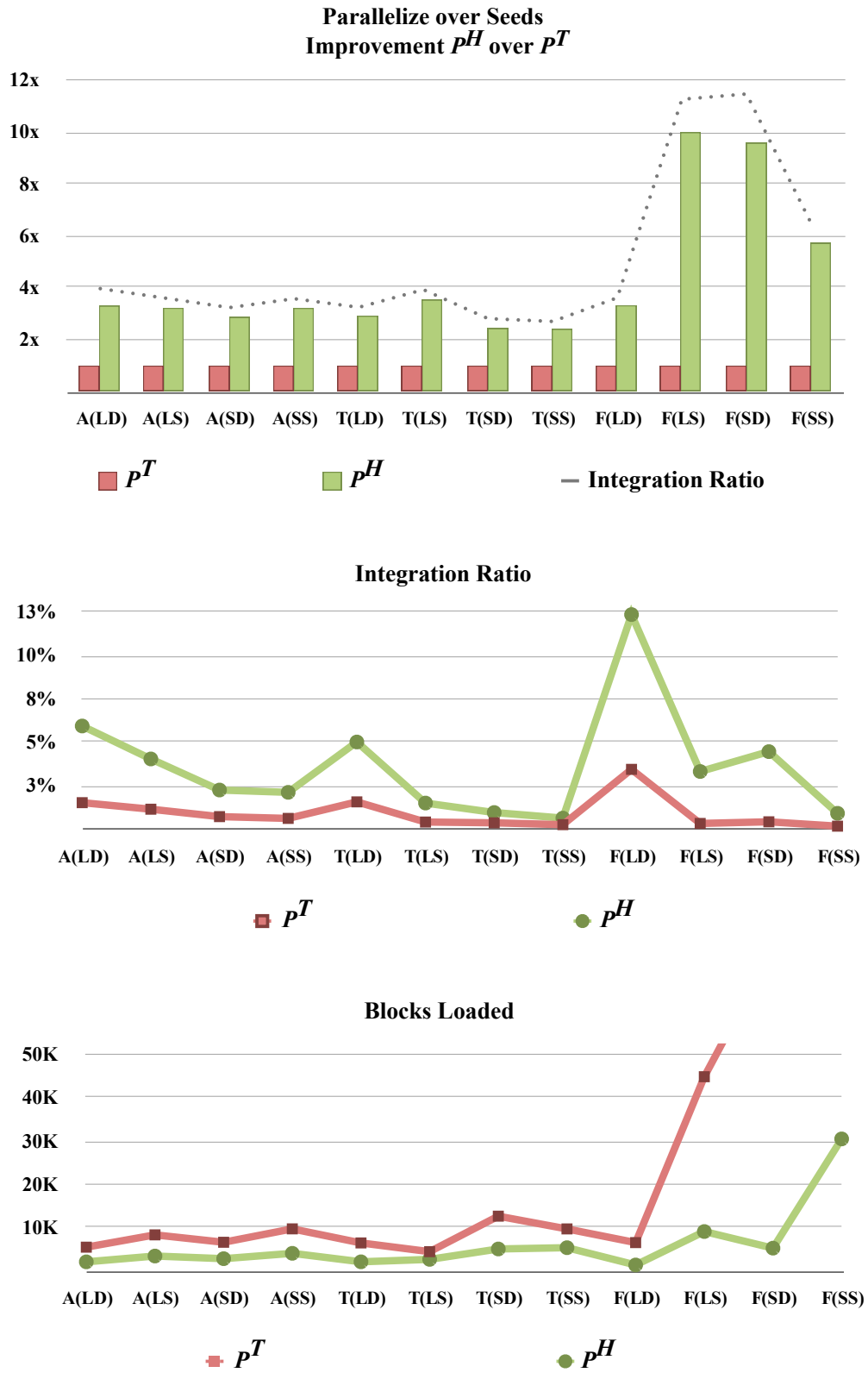

Figure 10: Comparison of P^H and P^T performance for the parallelize-over-seeds IC computation approach. the P^H version is faster in all cases because fewer blocks are loaded, allowing for an increased integration ratio, along with better load balancing due to there being more processors to work on a given number of ICs. (See 3.2.1 for a detailed discussion.) Image source: Camp et al., 2011 [7].

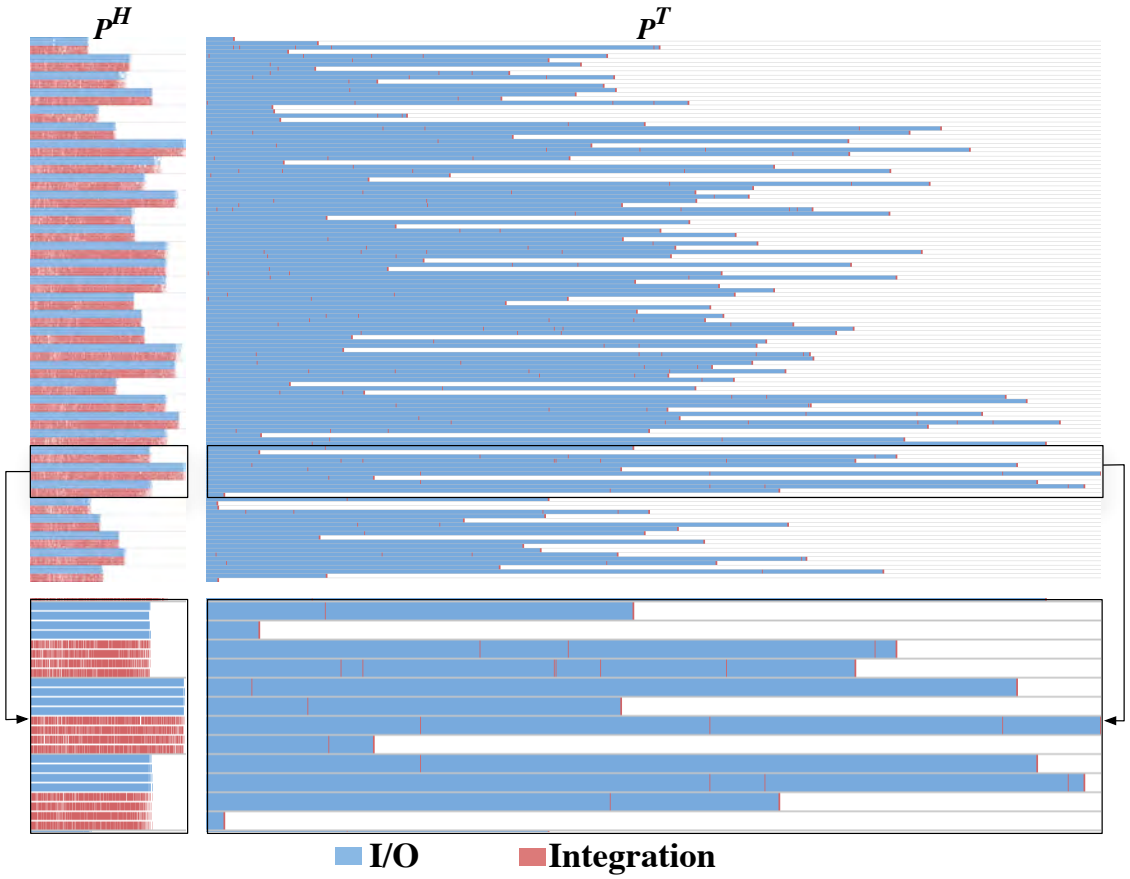

Figure 11: This Gantt chart shows a comparison of integration and I/O performance/activity of the *parallelize-over-seeds* P^T and P^H versions for one of the benchmark runs. Each line represents one thread (left column) or task (right column). The P^H approach outperforms the P^T one by about 10×, since the four I/O threads in the P^H can supply new data blocks to the four integration threads at an optimal rate. However, work distribution between nodes is not optimally balanced. In the P^T implementation, the I/O wait time dominates the computation by a large margin, due to redundant data block reads, and work being distributed less evenly. This can be easily seen in the enlarged section of the Gantt chart. (See Section 3.2.1 for more details.) Image source: Camp et al., 2011 [7].

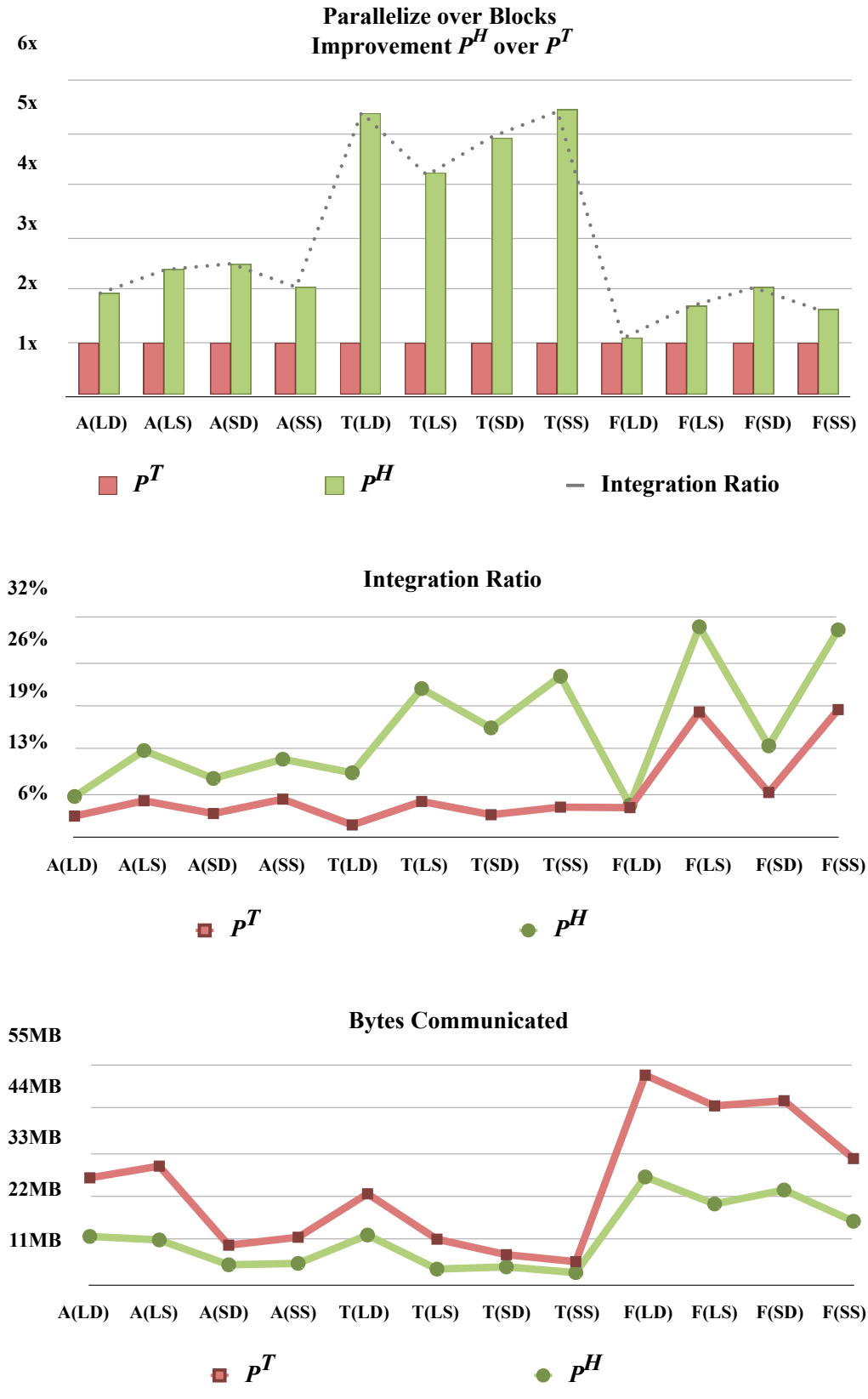

Figure 12: Performance comparison of the P^H and P^T variants of the *parallelize-over-blocks* algorithm. The P^H version has a much lower runtime for each test due to better load balancing, and because less communication enables a higher integration ratio. (See Section 3.2.2 for a detailed discussion.) Image source: Camp et al., 2011 [7].

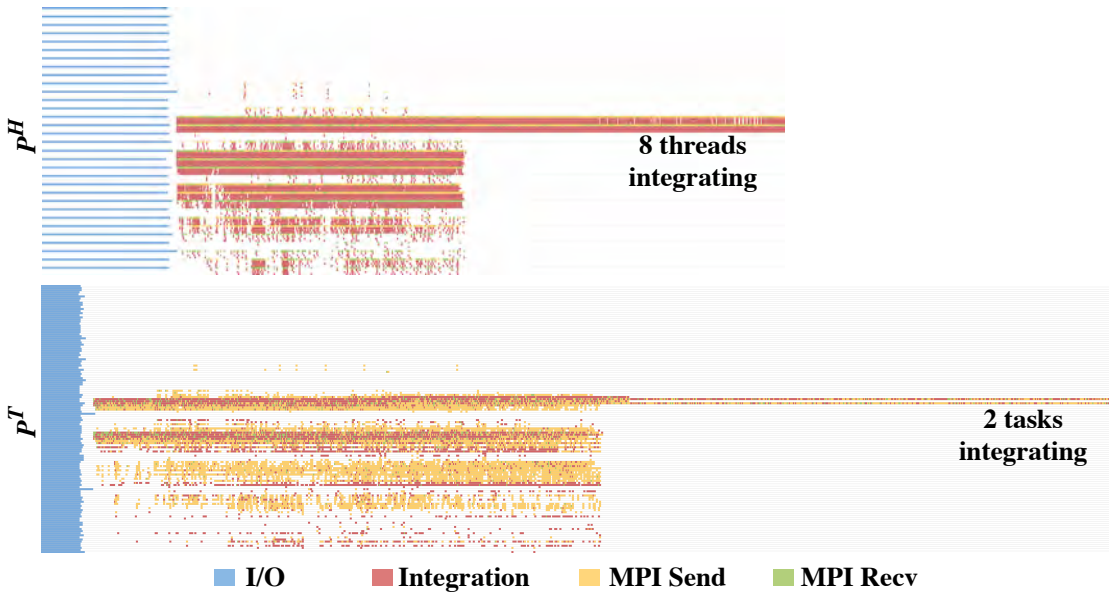

Figure 13: This Gantt chart shows a comparison of integration, I/O, `MPI_Send`, and `MPI_Recv` performance/activity of the *parallelize-over-blocks* P^T and P^H versions for one of the benchmark runs. Each line represents one thread (top) or task (bottom). The comparison reveals that the initial I/O phase, using only one thread, takes about $4\times$ longer. The successive integration is faster, since multiple threads can work on the same set of blocks, leading to less communication. Towards the end, the eight threads are performing IC integration in the P^H approach, as opposed to only two tasks in the P^T model. (See Section 3.2.2 for more details.) Image source: Camp et al., 2011 [7].

References

[1] VisIt – Software that delivers Parallel, Interactive Visualization. http://visit.llnl.gov/.

[2] 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*, pages 269–276, 1999.

[3] E. Wes Bethel, Hank Childs, and Charles Hansen, editors. *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*. Chapman & Hall, CRC Computational Science. CRC Press/Francis–Taylor Group, Boca Raton, FL, USA, November 2012. http://www.crcpress.com/product/isbn/9781439875728.

[4] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 3.1, July 2011. http://www.openmp.org/wp/openmp-specifications.

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

[6] David Camp, Hank Childs, Amit Chourasia, Christoph Garth, and Kenneth I. Joy. Evaluating the Benefits of an Extended Memory Hierarchy for Parallel Streamline Algorithms. In *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, Providence, RI, USA, October 2011.

[7] David Camp, Christoph Garth, Hank Childs, Dave Pugmire, and Ken Joy. Streamline Integration Using MPI-Hybrid Parallelism on a Large Multi-Core Architecture. *IEEE Transactions on Visualization and Computer Graphics*, 17(11):1702–1713, November 2011.

[8] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[9] Hank Childs, Eric S. Brugger, Kathleen S. Bonnell, Jeremy S. Meredith, Mark Miller, Brad J. Whitlock, and Nelson Max. A Contract-Based System for Large Data Visualization. In *Proceedings of IEEE Visualization*, pages 190–198, 2005.

[10] NVIDIA Corporation. What is CUDA? http://www.nvidia.com/object/what_is_cuda_new.html, 2011.

[11] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. *SIGGRAPH Computer Graphics*, 22(4):65–74, 1988.

[12] P. Fischer, J. Lottes, D. Pointer, and A. Siegel. Petascale Algorithms for Reactor Hydrodynamics. *Journal of Physics: Conference Series*, 125:1–5, 2008.

[13] Khronos Group. OpenCL – The Open Standard for Parallel Programming of Heterogeneous Systems. http://www.khronos.org/opencl/, 2011.

[14] Mark Howison, E. Wes Bethel, and Hank Childs. MPI-hybrid Parallelism for Volume Rendering on Large, Multi-core Systems. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, Norrköping, Sweden, May 2010. LBNL-3297E.

[15] Mark Howison, E. Wes Bethel, and Hank Childs. Hybrid Parallelism for Volume Rendering on Large, Multi- and Many-core Systems. *IEEE Transactions on Visualization and Computer Graphics*, 99(PrePrints), 2011.

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

[17] Kwan-Liu Ma. Parallel Volume Ray-Casting for Unstructured-Grid Data on Distributed-Memory Architectures. In *PRS '95: Proceedings of the IEEE Symposium on Parallel Rendering*, pages 23–30, New York, NY, USA, 1995. ACM.

[18] Kwan-Liu Ma, James S. Painter, Charles D. Hansen, and Michael F. Krogh. A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering. In *Proceedings of the 1993 Parallel Rendering Symposium*, pages 15–22. ACM Press, October 1993.

[19] Jason Nieh and Marc Levoy. Volume Rendering on Scalable Shared-Memory MIMD Architectures. In *Proceedings of the 1992 Workshop on Volume Visualization*, pages 17–24. ACM SIGGRAPH, October 1992.

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

[21] Thomas Porter and Tom Duff. Compositing Digital Images. *Computer Graphics*, 18(3):253–259, 1984. Proceedings of ACM/Siggraph.

[22] Dave Pugmire, Hank Childs, Christoph Garth, Sean Ahern, and Gunther H. Weber. Scalable Computation of Streamlines on Very Large Datasets. In *Proceedings of Supercomputing (SC09)*, Portland, OR, USA, November 2009.

[23] Paolo Sabella. A Rendering Algorithm for Visualizing 3D Scalar Fields. *SIGGRAPH Computer Graphics*, 22(4):51–58, 1988.

[24] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI – The Complete Reference: The MPI Core, 2nd ed.* MIT Press, Cambridge, MA, USA, 1998.

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

[26] The Top 500 Supercomputers, 2011. `http://www.top500.org`.

[27] Craig Upson and Michael Keeler. V-buffer: Visible Volume Rendering. In *SIGGRAPH '88: Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, pages 59–64, New York, NY, USA, 1988. ACM.