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

Mark Howison, E. Wes Bethel, Hank Childs
Visualization Group, Lawrence Berkeley National Laboratory
Email: {mhowison,ewbethel,hchilds}@lbl.gov

Abstract

This work studies the performance and scalability characteristics of "hybrid" parallel programming and execution as applied to raycasting volume rendering – a staple visualization algorithm – on a large, multi-core platform. Historically, the Message Passing Interface (MPI) has become the de-facto standard for parallel programming and execution on modern parallel systems. As the computing industry trends towards multi-core processors, with four- and six-core chips common today and 128-core chips coming soon, we wish to better understand how algorithmic and parallel programming choices impact performance and scalability on large, distributed-memory multi-core systems. Our findings indicate that the hybrid-parallel implementation, at levels of concurrency ranging from 1,728 to 216,000, performs better, uses a smaller absolute memory footprint, and consumes less communication bandwidth than the traditional, MPI-only implementation.

1 Introduction

It is well accepted that the future of parallel computing involves chips that are comprised of many (smaller) cores. With this trend towards more cores on a chip, 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 – raycasting volume rendering – implemented with different parallel programming models and run on a large supercomputer comprised of six-core chips. We compare a traditional implementation based on message-passing against a "hybrid" parallel implementation, which uses a blend of traditional message-passing (inter-chip) and shared-memory (intra-chip) parallelism.

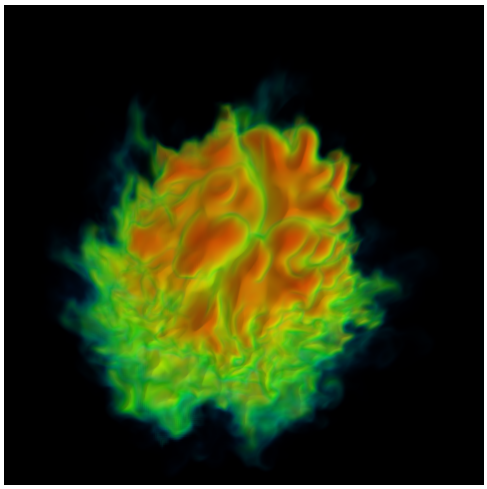

Figure 1: This 4608^2 image of a combustion simulation result was rendered by our MPI+pthreads implementation running on 216,000 cores of the JaguarPF supercomputer.

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, 3]). 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 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-parallel implementation runs faster, requires less memory, and for our particular algorithm and set of implementation choices, consumes less communication bandwidth than the traditional, MPI-only implementation.

2 Implementation

Our implementation of a parallel raycasting volume renderer draws on a large body of prior research (see [4] for a detailed overview). The primary contributions of our work is a comparison of the performance and resource requirements between hybrid- and traditional-parallel programming models.

From a high level view, our parallel volume renderer distributes a source data volume S across n parallel MPI PEs. Each PE reads in $\frac{1}{n}$ 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 PE 0 for display or I/O to storage. Our distributed memory parallel implementation is written in C++ and C and makes calls to MPI. The portions of the implementation that are shared-memory parallel are written using a combination of C++ and C and either POSIX threads or OpenMP, so that we are comparing two hybrid implementations, MPI+pthreads and MPI+OpenMP.

Our raycasting volume rendering code implements Levoy's method [5]: we compute the intersection of a ray with the data block, and then compute the color at a fixed step size along the ray through the volume. All such colors along the ray are integrated front-to-back using the "over" operator. Output from our algorithm consists of image fragments that contain an x, y pixel location, R, G, B, α color information, and a z-coordinate. The z-coordinate is the location in eye coordinates where the ray penetrates the block of data. In the MPI-only case, this serial implementation is invoked on each of the PEs. Each operates on its own disjoint block of data. As 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 later composition step.

In the MPI-hybrid case, the raycasting volume renderer on each MPI PE is a shared-memory parallel implementation consisting of T threads all executing concurrently to perform the raycasting on a single block of data. As in [6], we use an image-space partitioning: each thread is responsible for raycasting a portion of the image. Our image-space partitioning is interleaved, where the image is divided into many small tiles that are distributed amongst the threads. Through a process of manual experimentation, we determined that an image tile size of 32×32 pixels produced consistently better performance than other tile sizes for a variety of source volume sizes on the six-core AMD Opteron processor. We also found that a dynamic work assignment of tiles to threads minimized load imbalance.

Compositing begins by partitioning the pixels of the final image amongst the PEs. Next, an all-to-all communication step exchanges each fragment from the PE where it was generated in the raycasting phase to the PE that owns the region of the image in which the fragment lies. This exchange is implemented using an `MPI_Alltoallv` call, which provides the same functionality as direct sends and receives, but bundles the messages into fewer point-to-point exchanges for greater efficiency. Each PE 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 PE 0.

Peterka et al. [3] reported scaling difficulties for compositing when using more than 8,000 PEs. They solved this problem by reducing the number of PEs receiving fragments to be no more than 2,000. We emulated this approach, again limiting the number of PEs receiving fragments, although we experimented with values higher than 2,000.

In the hybrid 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 PEs. This approach results in fewer messages than if all threads in the hybrid parallel implementation were to send messages to all other threads on all other CPUs. Our aim here is to better understand the opportunities for improving performance in the hybrid-parallel implementation. The overall effect of this design choice is an improvement in communication characteristics, as indicated in Section 3.3.

3 Results

We conducted a strong scaling study where we rendered a 4608^2 image from a 4608^3 dataset (roughly 97.8 billion cells) at concurrency levels from 1,728-way parallel to 216,000-way parallel. Our test system, JaguarPF, is a Cray XT5 located at Oak Ridge National Lab. 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 1.3GB per core.

On JaguarPF, a minimum of 1,728 cores is required to accommodate this particular problem size. In the hybrid case, we shared a data block among six threads and used one sixth as many MPI PEs. Although we could have shared a data block among as many as twelve threads on each dual-socket six-core node, we chose not to because 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. 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 five different ratios of total PEs to compositing PEs. 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 characterize the variability of the trials.

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. 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, LBNL.

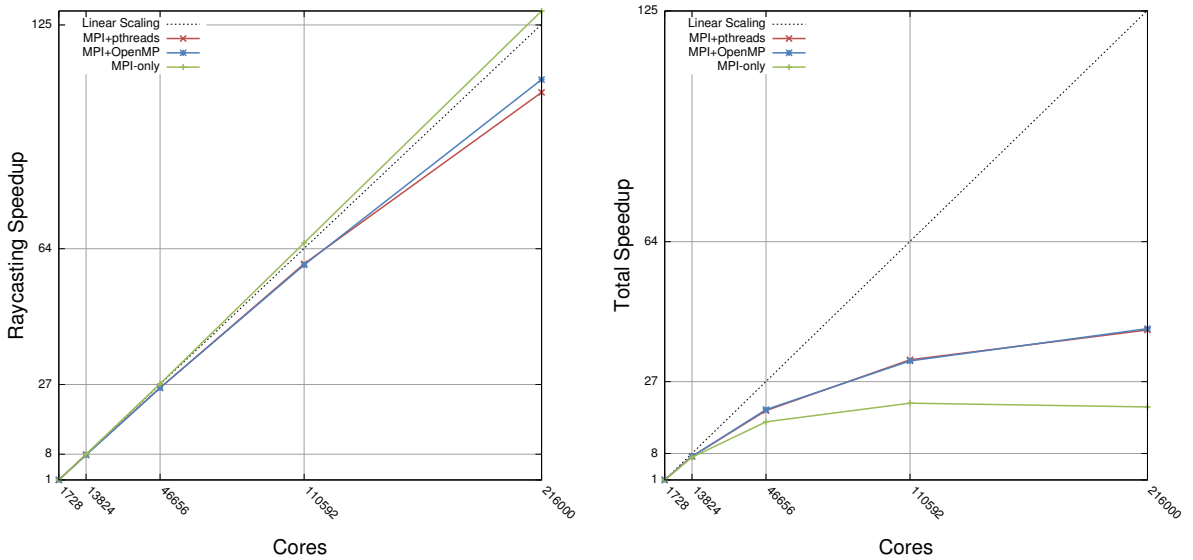

Figure 2: The speedups (referenced to 1,728 cores) for both the raycasting phase and the total render time (raycasting and compositing).

Because the compute nodes on an XT5 system have no physical disk for swap, and hence no virtual memory, exceeding the physical amount of 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. We chose concurrencies that are cubic numbers to allow for a clean decomposition of the entire volume into cubic blocks per MPI PE.

3.1 Memory Usage

Because MPI-hybrid uses fewer MPI PEs, it incurs less memory overhead from the MPI runtime environment and from program-specific data structures that are allocated per PE. We measured the memory footprint of the program directly after calling `MPI_Init` using the `VmRSS`, or "resident set size," value from the `/proc/self/status` interface. Memory usage was sampled only from MPI PEs 0 through 6, but those values agreed within 1–2%. At 216,000 cores, the per-PE runtime overhead of MPI-only was 176MB, more than twice the 82MB required by MPI-hybrid. The MPI-only per-node (2,106MB) and aggregate (37,023MB) memory usage was another factor of six larger than MPI-hybrid (165MB per-node, 2,892MB aggregate) because it uses 6× as many PEs as MPI-hybrid. Thus, MPI-only used nearly 12× as much memory per-node and in-aggregate than MPI-hybrid to initialize the MPI runtime at 216,000-way concurrency.

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 precomputed for our dataset). Because the MPI-hybrid approach uses fewer, larger blocks in its decomposition, it requires less exchange and storage of ghost data by roughly 40% across all concurrencies.

3.2 Raycasting

Our raycasting phase scales nearly linearly because it involves no inter-processor communication (see Figure 2). Each MPI PE obtains its data block, then launches either one (MPI-only) or six (MPI-hybrid) raycasting threads. Working independently, each thread tests for ray-triangle intersections along the data block's bounding box and, in the case of a hit, integrates the ray by sampling data values at a fixed interval along the ray, applying a transfer function to the values, and aggregating the resulting color and opacity in a fragment for that ray's image position. For these runs and timings, we use 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, and this behavior is entirely dependent on the view.

Overall, we have achieved linear scaling up to 216,000 for the raycasting phase with MPI-only (see Figure 2). MPI-hybrid exhibits different scaling behavior because it has a different decomposition geometry: MPI-only has a perfectly cubic decomposition, while MPI-hybrid aggregates $1 \times 2 \times 3$ cubic blocks into rectangular blocks that are longest in the z-direction. The interaction of the decomposition geometry and the camera direction determine the maximum number of ray integration steps, which is the limiting factor for the raycasting time. At lower concurrencies, this interaction benefits MPI-hybrid, which outperforms MPI-only by as much as 11%. At higher concurrencies the trend flips, and MPI-only outperforms MPI-hybrid by 10%. We expect that if we were able to run on an eight-core system with a $2 \times 2 \times 2$ aggregation factor for MPI-hybrid, both implementations would scale identically. We also note that at 216,000 cores, raycasting is less than 20% of the total runtime (see Figure 4), and MPI-hybrid is over 50% faster because of gains in the compositing phase that we describe next.

3.3 Compositing

We observe that with increasing concurrency, the MPI-hybrid compositing times are systematically better than the MPI-only implementation. The primary cost of compositing is the `MPI_Alltoallv` call

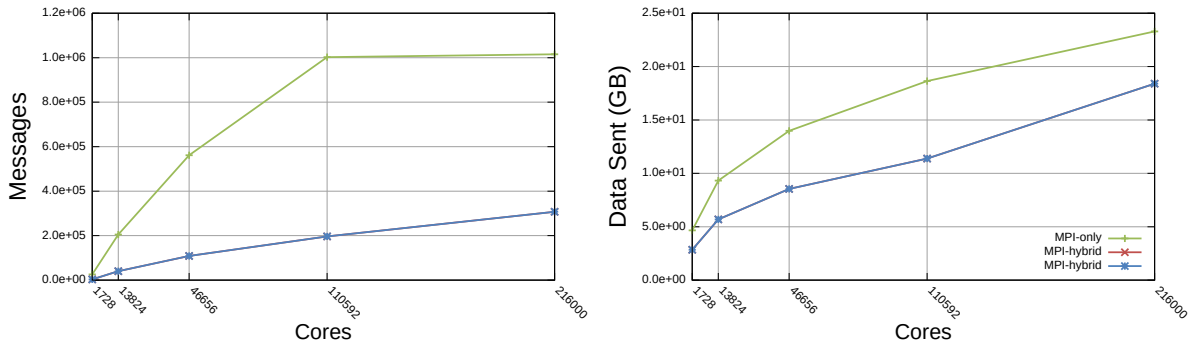

Figure 3: The number of messages and total data sent during the compositing fragment exchange.

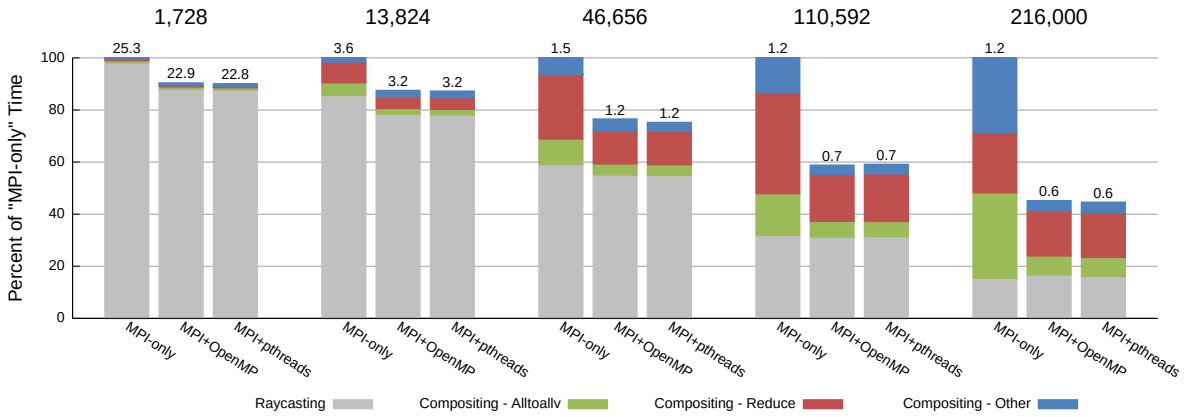

Figure 4: In terms of total render time, MPI-hybrid outperforms MPI-only at every concurrency level, with performance gains improving at higher concurrency.

that moves each fragment from the PE where it originated during raycasting to the compositing PE that owns the region of image space where the fragment lies. Because MPI-hybrid aggregates these fragments in the memory shared by six threads, it uses on average about 6× fewer messages than MPI-only (see Figure 3). In addition, MPI-hybrid exchanges less fragment data because its larger data blocks allow for more compositing to take place during ray integration.

Overall, the best compositing time at 216,000 cores for MPI-hybrid (0.35s, 4500 compositors) is 67% less than for MPI-only (1.06s, 6750 compositors) because the MPI-hybrid approach results in fewer and smaller messages than the MPI-only implementation. Furthermore, at this scale compositing time dominates rendering time, which is roughly 0.2s for both MPI-only and MPI-hybrid. Thus, the total render time is 55% faster for MPI-hybrid (0.56s versus 1.25s). The advantage of MPI-hybrid over MPI-only also becomes greater as the number of cores increases (see Figure 4).

4 Conclusion and Future Work

The multi-core era offers new opportunities and challenges for parallel applications. This study has shown that hybrid parallelism offers performance improvements and better resource utilization for raycasting volume rendering on a large, distributed-memory supercomputer comprised of multi-core CPUs. The advantages we observe for hybrid parallelism 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.

We found that at high concurrency, fragment exchange during the compositing phase is the most expensive operation. Our compositing algorithm relies entirely on the implementation of the `MPI_Alltoallv`

call in the Cray MPI library. Our finding that using a subset of compositing PEs reduces communication overhead agrees with what Peterka et al. [3] found for the MPI implementation on an IBM BG/P system, suggesting that both libraries use similar implementations and optimizations of `MPI_Alltoallv`. A separate paper by Peterka et al. [7] introduces a new compositing algorithm, "Radix-k," that shows good scaling up to 16,000-way concurrency. This approach, which is compatible with our hybrid parallel system, may lead to even faster compositing times. Studying their combined performance, especially at concurrencies in the hundreds of thousands, is an avenue 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.

References

[1] Asanovic, K., et al. (2006) The landscape of parallel computing research: A view from Berkeley. Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

[2] Kniss, J., McCormick, P., McPherson, A., Ahrens, J., Painter, J., Keahey, A., and Hansen, C. (2001) Interactive texture-based volume rendering for large data sets. *IEEE Computer Graphics and Applications*, **21**, 52–61.

[3] Peterka, T., Yu, H., Ross, R., Ma, K.-L., and Latham, R. (2009) End-to-end study of parallel volume rendering on the IBM Blue Gene/P. *ICPP '09: Proceedings of the 2009 International Conference on Parallel Processing*, Washington, DC, USA, pp. 566–573, IEEE Computer Society.

[4] Howison, M., Bethel, E. W., and Childs, H. (2010) MPI-hybrid parallelism for volume rendering on large, multi-core systems. *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, Norrköping, Sweden, May, lBNL-3297E.

[5] Levoy, M. (1988) Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, **8**, 29–37.

[6] Nieh, J. and Levoy, M. (1992) Volume rendering on scalable shared-memory MIMD architectures. *Proceedings of the 1992 Workshop on Volume Visualization*, October, pp. 17–24, ACM SIGGRAPH.

[7] Peterka, T., Goodell, D., Ross, R., Shen, H.-W., and Thakur, R. (2009) A configurable algorithm for parallel image-compositing applications. *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, New York, NY, USA, pp. 1–10, ACM.