Streamline Integration Using MPI-Hybrid Parallelism on a Large Multicore Architecture

David Camp, *Member*, *IEEE*, Christoph Garth, *Member*, *IEEE*, Hank Childs, Dave Pugmire, and Kenneth I. Joy, *Member*, *IEEE*

Abstract—Streamline computation in a very large vector field data set represents a significant challenge due to the nonlocal and data-dependent nature of streamline integration. In this paper, we conduct a study of the performance characteristics of hybrid parallel programming and execution as applied to streamline integration on a large, multicore platform. With multicore processors now prevalent in clusters and supercomputers, there is a need to understand the impact of these hybrid systems in order to make the best implementation choice. We use two MPI-based distribution approaches based on established parallelization paradigms, *parallelize over seeds* and *parallelize over blocks*, and present a novel MPI-hybrid algorithm for each approach to compute streamlines. Our findings indicate that the work sharing between cores in the proposed MPI-hybrid parallel implementation results in much improved performance and consumes less communication and I/O bandwidth than a traditional, nonhybrid distributed implementation.

Index Terms—Concurrent programming, parallel programming, modes of computation, parallelism and concurrency, picture/image generation, display algorithms.

1 Introduction

Streamlines or more generally integral curves are one of the most illuminating techniques to obtain insight from simulations that involve vector fields, and they are a cornerstone of visualization and analysis across a variety of application domains. Drawing on an intuitive interpretation in terms of particle movement, they are an ideal tool to illustrate and describe a wide range of phenomena encountered in the study of scientific problems involving vector fields, such as transport and mixing in fluid flows. Moreover, they are used as building blocks for sophisticated visualization techniques (see, e.g., [1], [2], [3]), which typically require the calculation of large amounts of integral curves. Successful application of such techniques to large data must crucially leverage parallel computational resources to achieve well-performing visualization. Streamline computations are notoriously hard to parallelize in a distributed memory setting [4] because runtime characteristics are highly problem and data dependent.

Supercomputers are increasingly relying on nodes that contain multiple cores to achieve FLOP performance while minimizing power consumption. While current supercomputer nodes contain two to six cores, current trends indicate that future supercomputers will consist of individual nodes with tens to hundreds of cores. This hardware approach gives rise to an important software question: which parallel programming model can effectively utilize such architectures? The classical method advocated, for example, by the widely prevailing Message Passing Interface (MPI) is to assign an MPI task to every core on every node; this approach is often the simplest way to write parallel programs. An increasingly popular approach, however, is to use *hybrid parallelism*, where fewer MPI tasks are used (typically one per node) and shared-memory parallelism is employed within a node. This approach, although more challenging to implement, can enable significant performance and efficiency gains. In this paper, we study the difference between a traditional MPI-based implementation and an MPI-hybrid parallel approach applied to the problem of streamline integration in a large vector-field data set. This study is complementary to a scaling study; we are studying the benefits of hybrid parallelism applied to streamline integration, which is related to but distinct from scalability.

The aim of this work is to explore the performance of parallel distributed streamline computation when implemented using both hybrid and nonhybrid programming models. The hybrid parallel implementation is a blend of traditional message passing between CPUs and shared memory parallelism between cores on a CPU. We investigate the thesis that a hybrid parallel implementation can leverage significant improvements in performance via factors such as improved efficiency, reduced communication, and reduced I/O costs. The problem of streamline integration should especially benefit from such an approach, since its runtime complexity and I/O varies greatly with respect to both the data set under investigation and the number and distribution of streamlines to be computed. Based on a wide range of experiments, we perform for typical streamline computation scenarios (see Section 4), our findings (see Section 5) indicate that there is an opportunity for significant performance gains under the hybrid approach, resulting from reductions in memory footprint, communication, I/O, and improvements in

- *C. Garth and K.I. Joy are with the Department of Computer Science, University of California, Davis, One Shields Avenue, Davis, CA 95616. E-mail: {cgarth, kijoy}@ucdavis.edu.*
- *D. Camp and H. Childs are with Lawrence Berkeley National Laboratory and the Department of Computer Science, University of California, Davis, One Shields Avenue, Davis, CA 95616. E-mail: {dcamp, hchilds}@lbl.gov, hrchilds@ucdavis.edu.*
- *D. Pugmire is with Oak Ridge National Laboratory, Oak Ridge, TN 37831. E-mail: pugmire@ornl.gov.*

Manuscript received 14 July 2010; revised 11 Oct. 2010; accepted 17 Oct. 2010; published online 7 Dec. 2010.
Recommended for acceptance by M. Chen.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-2010-07-0145.
Digital Object Identifier no. 10.1109/TVCG.2010.259.

parallel efficiency. Our experiments are conducted with the VISIT visualization tool, and hence, the performance observations, we arrive at in this paper directly apply to real-world production visualization scenarios.

2 PREVIOUS WORK

2.1 Parallel Streamline Integration

The parallel solution of streamline-based problems has been considered in previous work using a multitude of differing approaches. Generally, both data sets represented as a number of disjoint blocks and computation in the form of integration work can be distributed. An early treatment of the topic was given by Sujudi and Haimes [5], who made use of distributed computation by assigning each processor one data set block. A streamline is communicated among processors as it traverses different blocks. Other examples of applying parallel computation to streamline-based visualization include the use of multiprocessor workstations to parallelize integral curve computation (see, e.g., [6]), and research efforts were focused on accelerating specific visualization techniques [7]. Similarly, PC cluster systems were leveraged to accelerate advanced integration-based visualization algorithms, such as time-varying Line Integral Convolution (LIC) volumes [8] or particle visualization for very large data [9].

Focusing on data size, out-of-core techniques are commonly used in large-scale data applications, where data sets are larger than main memory. These algorithms focus on achieving optimal I/O performance to access data stored on disk. For vector field visualization, Ueng et al. [10] presented a technique to compute streamlines in large unstructured grids using an octree partitioning of the vector field data for fast fetching during streamline construction using a small memory footprint. Taking a different approach, Bruckschen et al. [11] described a technique for real-time particle traces of large time-varying data sets by isolating all integral curve computation in a preprocessing stage. The output is stored on disk and can then be efficiently loaded during the visualization phase.

More recently, different partitioning methods were introduced with the aim of optimizing parallel integral curve computation. Yu et al. [12] introduced a parallel integral curve visualization that computes a set of representative, short integral segments termed pathlets in time-varying vector fields. A preprocessing step computes a binary clustering tree, that is, used for seed point selection and block decomposition. This seed point selection method mostly eliminates the need for communication between processors, and the authors are able to show good scaling behavior for large data. However, this scaling behavior comes at the cost of increased preprocessing time and, more importantly, loses the ability to choose arbitrary, user-defined seed points, which is often necessary when using streamlines for data analysis as opposed to obtaining a qualitative data overview. Chen and Fujishiro [13] applied a spectral decomposition using a vector-field derived anisotropic differential operator to achieve a similar goal with similar drawbacks.

More recently, Pugmire et al. [4] presented a systematic study of the performance and scalability of three parallelization algorithms : the first two correspond to parallelization over seed points and over data blocks, respectively, while the third algorithm (termed *Master-Slave*) adapts its behavior between these two extremes based on the observed behavior during the algorithm run. Such an adaptive strategy was found to roughly equal the better performance of either extreme, thus making a priori choice of parallelization strategy unnecessary (see Section 4.1). This study is partially based on this previous work as we focus on the two nonadaptive parallelization approaches, and our implementations are improved from the original ones done in Pugmire et al. [4]. Comparing hybrid and traditional parallelism with the Master-Slave approach is a difficult undertaking due to the high complexity of the algorithm; we leave this to future work. However, we understand streamline parallelization choices as a spectrum between parallelizing over data and parallelizing over computation. In this paper, we show that both ends of the spectrum are greatly improved with hybrid parallelism and presume that it is reasonable to conclude that algorithms in the middle of the spectrum (such as Master-Slave) will also strongly benefit. Finally, we exclude any notion of (typically time consuming) data set preprocessing and focus on unmodified data, as our intent is to quantify the potential benefits of a hybrid parallel approach over a traditional one in a production visualization scenario, where exhaustive preprocessing is not feasible.

2.2 Hybrid Parallelism

In the recent past, the MPI evolved as the de-facto standard for parallel programming and execution on supercomputers and clusters [14] due to its ability to effectively abstract machine architecture and communication modes. To make use of MPI, applications must explicitly invoke MPI library calls to implement parallel execution and communication paradigms such as data scatter and gather and execution synchronization. In MPI terminology, a *task* is the fundamental unit of execution. A parallel MPI application consists of multiple tasks, all executing a single identical program, distributed over all processors of a parallel system. Historically, each MPI task maps one-to-one to the processors of a parallel system. More recently, with the evolution of multicore processors, MPI implementations provide support to map tasks onto one or more cores of such chips. In this configuration, each task maps to a single core of a multicore chip. However, treating each core of a multicore chip as an individual processor with isolated resources incurs a number of avoidable penalties. Since tasks are treated as individual processes with separate address spaces, there is little opportunity for direct sharing of data residing in the same physical memory shared by all cores on a single chip. Because all communication between tasks is abstracted through the MPI library and a corresponding penalty is incurred in such scenarios, it stands to reason that MPI applications may not perform optimally on multicore platforms, where there is the opportunity for more efficient communication through local, high speed, shared memory that completely bypasses the MPI interface.

It is considerably easier to develop shared-memory parallel applications than distributed memory ones since

the need for explicit data movement between the parallel program elements is obviated, since all CPUs have access to the same, shared memory. In this model, an application typically consists of one or more execution *threads*, and two common programming models for building shared-memory parallel codes are POSIX threads [15] and OPENMP [16]. These APIs allow applications to manage creation and termination of threads, and synchronize thread execution through semaphores, mutexes, and condition variables. The scalability of shared-memory codes is typically limited by physical constraints: there are usually only a few cores in a single CPU—four to six cores per CPU are common today, although current trends indicate the future availability of multicore chips containing hundreds to thousands of cores.

In both of the above programming models, a developer must explicitly design for parallelism, as opposed to relying on a compiler to discover and implement parallelism. Other approaches are data-parallel languages (see, e.g., CUDA [17]), languages with data parallel extensions (see, e.g., High Performance Fortran [18], and *Partitioned Global Address Space* (*PGAS*) languages (see, e.g., Unified Parallel C [19]), which provide a unified address space of distributed memory platforms; here, parallelism is expressed implicitly via language syntax and program design.

In this paper, we focus on exploiting a *hybrid parallelism* approach that makes use of a distributed-memory approach between nodes, and leverages shared-memory parallelism using threads across the cores of a node. In the following, we will refer to this also as *MPI hybrid*, as opposed to *MPI only* or *traditional parallelism* for a purely distributed-memory technique.

Prior implementations of parallel algorithms in MPI-hybrid form have focused on benchmarking well-known computational kernels. Hager et al. [20] describes potentials and challenges of the dominant programming models on multicore SMP nodes systems. Mallon et al. [21] work evaluated the MPI performance against UPC and OpenMP on multicore architectures. In contrast, ours is the first study that takes aim at this space from the point of view of integration-based vector field visualization. Previous studies do not provide a clear answer as to which approach is preferable in terms of performance in this scenario since elements of overall runtime under both approaches are complex, and include a mixture of problem and configuration specific factors (refer also to the discussion in Section 4.1).

The same studies highlight several issues for consideration when comparing hybrid and traditional parallelism. Generally, hybrid approaches can be expected to require less memory due to avoiding duplication of data in a data distribution scenario, as well as using operating system resources more efficiently. Similarly, they typically require less time-consuming internode communication. In this work, we examine the validity of these assumptions for two parallelization strategies that center on streamline integration in large data sets; our results indicate that significant performance and efficiency improvements can be leveraged by a hybrid parallel implementation. The speedup factors, we have observed range from two to ten on a quad-core architecture. The communication and I/O bandwidth are also lowered by significant amounts.

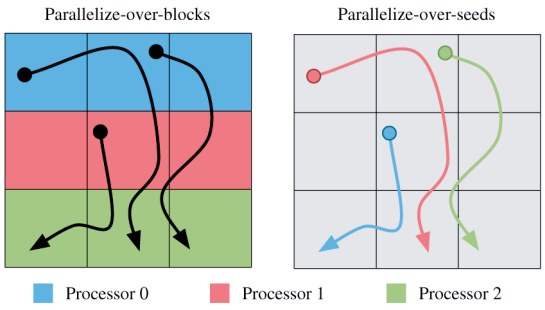


Fig. 1. Immediate parallelization approaches for distributed streamline integration. *Parallelize-over-blocks* (left) relies on a fixed assignment of data blocks to processors and passes streamlines between processor as they advance into new blocks. *Parallelize-over-seeds* (right) distributed streamlines evenly among processors and data blocks are loaded on demand when required to integrate a streamline.

3 Hybrid Parallel Streamline Integration

In all algorithms, the problem mesh is decomposed into a number of spatially disjoint blocks. The two parallelization strategies that we present differ fundamentally in how blocks are assigned and reassigned among processors, changing the I/O, memory, and processing profiles so as to address the challenges in data set size, seed set size, seed set distribution, and vector field complexity, as discussed in Section 4.1.

There are two straightforward parallelization approaches that partition either the computation workload, with seed points as the elementary unit of work, or the data set, where data blocks are distributed (see Fig. 1). In the following, we describe how we implement both strategies using MPI only and MPI-hybrid models.

3.1 Parallelization over Seeds

The parallelize-over-seeds algorithm parallelizes over the set of seed points, assigning each MPI task a fixed number of seed points from which streamlines are then integrated. Data blocks are loaded on demand when required, i.e., when a streamline integration must be continued in a block, that is not present in memory. Multiple such blocks are kept in memory in a block cache of size N_{blocks}, and new blocks are only loaded when no streamline can be continued on the current resident blocks. Since blocks might be used repeatedly during the integration of streamlines, they are kept in the cache as long as possible. Blocks are evicted in least recently used order to make room for new blocks. Fig. 2 provides an overview of both MPI only and MPI-hybrid implementations of this algorithm. This algorithm benefits from a small amount of communication; the only communication occurs at initialization and termination.

The initial assignment of seed points to nodes is based on spatial proximity following the reasoning that the integral curves traced from spatially close seed points are likely to traverse the same regions, and thus, blocks of a data set. We find this approach effective in reducing overall I/O cost; this is especially true for dense seeding.

In the MPI-only version, a single thread, corresponding to the MPI task's process, maintains an overview of both the set of streamlines and the cache, and performs integration and I/O as required.

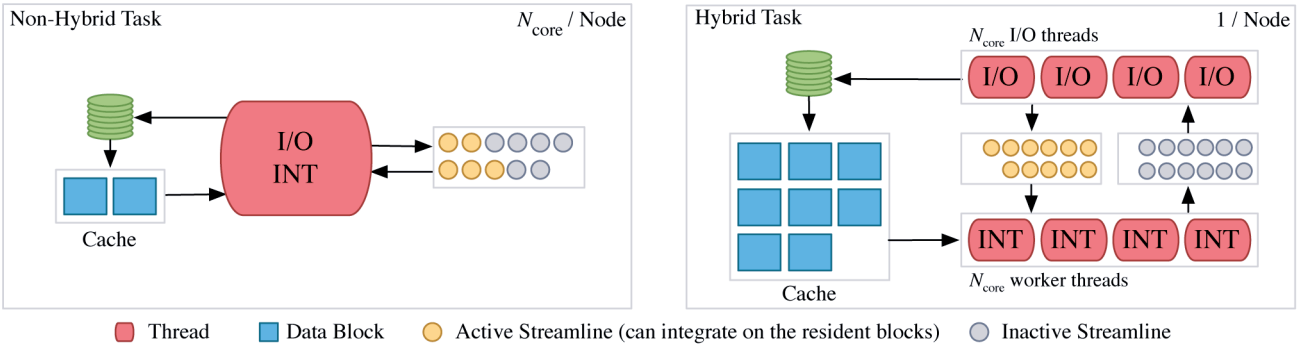


Fig. 2. Implementation of the *parallelize-over-seeds* algorithm with nonhybrid (left) and hybrid (right) versions. In the nonhybrid model, each MPI task integrates streamlines (INT) and manages its own cache by loading blocks from disk (I/O), whereas N_{core} worker threads share a cache in the hybrid implementation. In this case, multiple I/O threads manage the cache and observe, which streamlines can (*active*) and cannot (*inactive*) integrate with the resident data blocks. Future blocks to load are determined from the list of inactive streamlines. MPI communication is limited to gathering results and is not shown.

In the MPI-hybrid implementation, streamlines are maintained in two sets. Streamlines contained in the *active* set can be integrated on the blocks currently residing in the cache, while *inactive* streamlines require further block I/O to integrate. The I/O threads identify blocks to be loaded from the inactive set, and then, initiate I/O if there is room in the cache. After a block is loaded, the streamlines waiting on it are migrated to the active set. A team of worker threads fetches streamlines from the active set performs integration of each one on the available blocks, and then, retires them to the inactive set. Streamlines that have completed integration 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. In our implementation, the number of worker and I/O threads is arbitrary. However, we typically choose it to reflect the number of cores (N_{core}) dedicated to each MPI task.

Overall, the performance of the parallelize-over-seeds scheme depends crucially on the data loads and cache size N_{blocks}; if the cache is too small, blocks must be brought in from external storage repeatedly. We expect MPI hybrid to have three main advantages over MPI only: 1) MPI hybrid will have a larger shared cache and will do less redundant I/O, 2) when a significant number of streamlines are inside the same data block, MPI only must load that block separately on each MPI task, where MPI hybrid can perform one read and immediately share it among its threads, and 3) since I/O and integration are separated into different threads, they can execute asynchronously, and new blocks to load can be identified as soon as streamlines are returned to the inactive queue.

3.2 Parallelization over Blocks

The parallelize-over-blocks approach distributes data blocks over MPI tasks using a fixed assignment. Streamlines are then communicated between the tasks to migrate them to the task owning the block required to continue integration. This algorithm performs minimal I/O: before integration commences, every task loads all blocks assigned to it, leveraging maximal parallel I/O bandwidth.

Fig. 3 provides an overview of both MPI only and MPI-hybrid implementations of this algorithm. As in the parallelize-over-seeds case, the MPI-only version consists of a single thread that maintains a set of streamlines to integrate. After each streamline is integrated as far as possible on the available data blocks, the next block is determined and the streamline is sent to the corresponding task; then, further streamlines are received from other tasks and stored for integration. Streamline communication is limited to a small amount of integration state. Streamline geometry generated during the integration remains with the task that generated it.

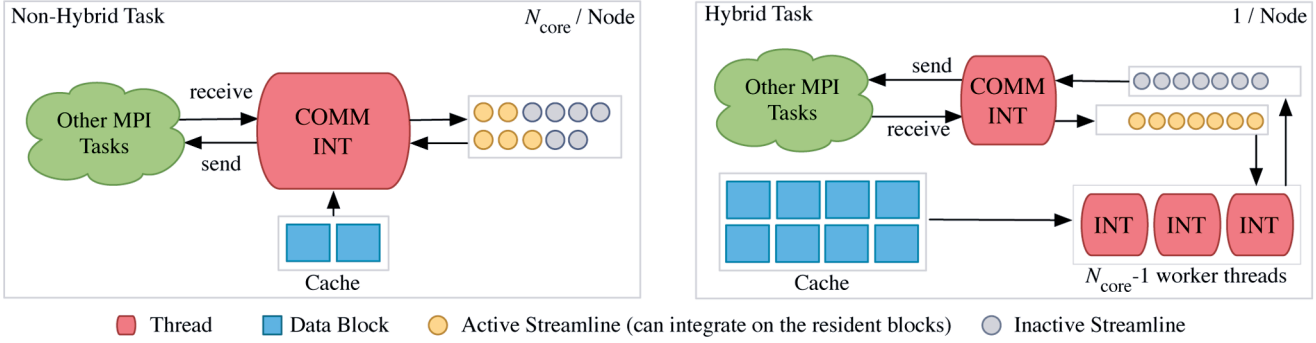


Fig. 3. Implementation of the *parallelize-over-blocks* algorithm with nonhybrid (left) and hybrid (right) versions. In the nonhybrid model, each process integrates streamlines (INT) and communicates with the cloud to receive and send them (COMM). In the hybrid implementation, $N_{\text{core}} - 1$ worker threads only integrate over a shared set of domains, while one thread additionally handles communication and integration. Data I/O is limited to initially loading the blocks assigned to each MPI task and is not shown.

In the MPI-hybrid implementation, streamlines are maintained in two queues, with the MPI task again split between a supervisor thread responsible for communication and (initial) I/O, and a team of worker threads. Newly received streamlines are again kept in the *active* queue; workers fetch from it integrate and put streamlines in the *inactive* queue, if they exit data blocks loaded by this task. From there, they are sent off to other tasks by the supervisor, or retired to a separate list when complete. All MPI tasks' supervisor threads also maintain a global count of active and complete streamlines to determine when to terminate. Synchronization between threads is performed as in the parallelize-over-seeds case.

Technically, it proves difficult to have the supervisor thread wait on both incoming streamlines and those returning to the active queue simultaneously. We initially used a polling approach with a fixed sleep interval; however, finding an optimal interval proved to be difficult and dependent on the problem characteristics. If the timeout was chosen too large, streamlines would remain in the inactive queue for too long. On the other hand, a small timeout resulted in many cycles used up by the polling, visibly decreasing the amount of computation performed by the workers. We thus settled on the approach of reducing the worker team to $N_{\text{core}} - 1$ threads, and let the supervisor thread participate in the integration with communication performed in between streamline integrations. The communication interval is thus coupled to the average duration of integration, and balancing is automatic. Note that for $N_{\text{core}} = 1$, this exactly corresponds to the MPI-only case (as opposed to the parallelize-over-seeds approach that still maintains two threads in this case).

The parallelize-over-blocks algorithm presupposes that the combined memory over all tasks can accommodate the entire data set. Aside from this, the algorithm behavior is not dependent on any further parameters.

We expect the parallel efficiency of the parallelize-over-seeds algorithm to be improved in the MPI-hybrid case. In the case, where a majority of the streamlines traverse the same block at the same time, the MPI tasks that own those blocks become bottlenecks. With the MPI-hybrid approach, multiple cores can access that block and advance the streamlines, where only one core can do so with the MPI-only implementation. Further, we expect the overall amount of communication of streamlines in the MPI-hybrid implementation to be reduced from the MPI-only case. This depends on how often a streamline needs to change tasks to continue integration, and is thus, a function of the vector field complexity and the chosen distribution scheme for the data blocks. The increased amount of memory available to each task in the hybrid setting allows more blocks to reside in a given task, thus decreasing the *probability* that a streamline needs to be communicated over the nonhybrid approach. However, if an unfortunate block distribution is chosen, the actual amount of communication might not be lowered. Examining the distribution of blocks with respect to the vector field structure is a complex task (see, e.g., [12]) and beyond the scope of this paper; since we are primarily aiming at quantifying the benefits of a hybrid approach for integration in this work, we choose a simple, fixed data distribution scheme that assigns blocks to tasks in numerical order. Our results (see Section 5) indicate that over three data sets with very different structural characteristics and four distinct test cases per data set, communication is substantially reduced in the MPI-hybrid integration scheme.

4 Experiments

In the following, we describe the test cases used to quantify performance in our experiments. First, we briefly discuss a number of factors that determine the characteristics of a streamline integration problem and provide the basis for our selection of tests.

4.1 Problem Complexity

As pointed out above, a number of strongly varying problems and configuration specific factors are inherent in streamline integration and need to be taken into account when evaluating performance and efficiency of a parallel streamline algorithm. Generally, streamline based problems can be classified according to four criteria.

4.1.1 Data Set Size

The size of the data set describing the vector field under consideration is crucial in choosing a parallelization strategy. If the considered field is *small* in the sense that it fits into main memory in its entirety, then optimally performing integral curve computation profits most from distributed computation and to a lesser amount from distributed data. However, for data so *large* that it cannot be loaded in its entirety, more complex schemes are required. Here, adaptive distribution of data over available parallel resources and optimal scheduling and dispatch of integral curve computation are necessary traits of a well performing parallelization approach. Large data is commonly compressed to save space and I/O load time, but this saving is at the expense of CPU decompression time which adds to the complete I/O time.

4.1.2 Seed Set Size

If the problem at hand requires only the computation of a thousand streamlines, parallel computation takes a secondary place to optimal data distribution and loading; we refer to the corresponding seed set as *small*, and they are most often encountered in interactive exploration scenarios where few integral curves are interactively seeded by a user. A *large* seed set encompasses many thousands of seed points for integral curves. For such problems to remain computationally feasible, it is paramount that the considered data distribution scheme allows for parallel computation of integral curves.

4.1.3 Seed Set Distribution

Similar to the seed set size, the distribution of seed points is an important problem characteristic. In the case, where seed points are located *densely* within the spatial and temporal domain of definition of a vector field, it is likely that it will traverse a relatively small amount of the overall data. For some applications such as streamline statistics, on the other hand, a *sparse* seed point set covers the entire vector field evenly. This results in integral curves traversing the entire data set. Hence, the seed set distribution determines

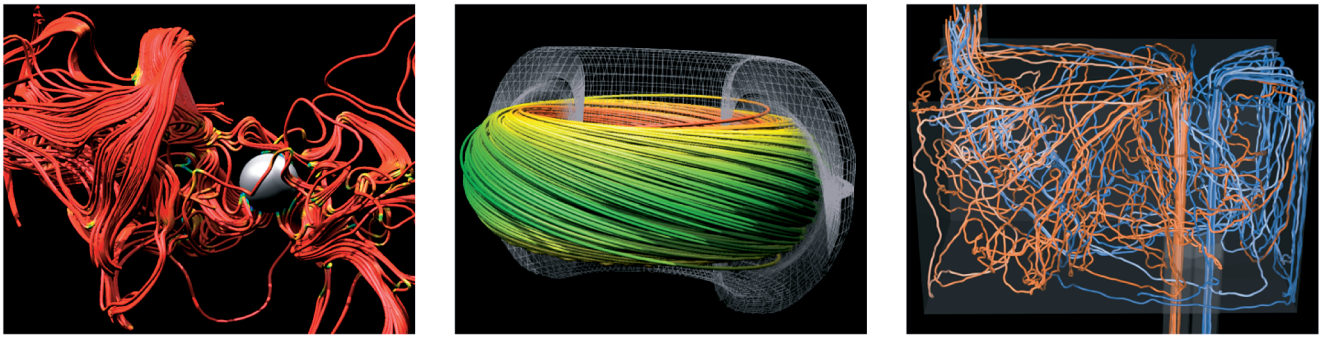


Fig. 4. The data dependent nature of streamlines creates a large opportunity for biased results when studied in the context of a single data set. To safeguard against such a bias, we used three data sets with widely varying vector field behavior in our study. The data sets were from astrophysics, fusion, and thermal hydraulics simulations (left to right).

strongly if performance stands to gain most from parallel computation, data distribution, or both.

4.1.4 Vector Field Complexity

Depending on the choice of seed points, the structure of a vector field can have a strong influence on which parts of the data need to be taken into account in the integral curve computation process. Critical points or invariant manifolds of strongly attracting nature draw streamlines toward them, and the resulting integral curves seeded in or traversing their vicinity remain closely localized. On the other hand, the opposite case of a nearly uniform vector field requires integral curves to pass through large parts of the data.

Overall, these factors determine to what extent a given integration-based problem can profit from parallel computation and data distribution. We next describe a number of prototypical test cases derived from practical visualization problems that exhibit varying characteristics and upon which we base our performance studies.

4.2 Test Cases

To cover a wide range of potential problem characteristics, four tests addressing all combinations of seed set size (small or large) and distribution (sparse or dense) are defined for each of three data sets (see Fig. 4). Although our techniques are readily applicable to any mesh type and decomposition scheme, the data sets we study here are multiblock and rectilinear. We have intentionally chosen this simplest type of data representation to exclude additional performance complexities that arise with more complex mesh types from this study.

4.2.1 Astrophysics

This data set results from the simulation of the magnetic field surrounding a solar core collapse resulting in a supernova. The search for the explosion mechanism of core-collapse supernovae and the computation of the nucleosynthesis in these spectacular stellar explosions is one of the most important and most challenging problems in computational nuclear astrophysics. Understanding the magnetic field around the core is very important and streamlines are a key technique for doing so. The simulation was computed by a GENASIS simulation [22], a multiphysics code being developed for the simulation of astrophysical systems involving nuclear matter [23]. GENASIS computes the magnetic field at each cell face. For the purposes of this study, a cell-centered vector is created by differencing the values at faces in the X, Y, and Z directions. Node-centered vectors are generated by averaging adjacent cells to each node. To see how this algorithm would perform on very large data sets, the magnetic field was upsampled onto a total 512 blocks with 1 million cells per block. The dense seed set corresponds to streamlines placed randomly in a small box around the collapsing core, whereas the sparse test places streamline seed points randomly throughout the entire data set domain. The small and large seeds sets contained 2,500 and 10,000 seed points, respectively, with integration times of 4,000 and 1,000 time units.

4.2.2 Fusion

The second data set is from a simulation of magnetically confined fusion in a tokamak device. The development of magnetic confinement fusion, which will be a future source for low-cost power, is an important area of research. Physicists are particularly interested in using magnetic fields to confine the burning plasma in a toroidal shape device, known as a tokamak. To achieve stable plasma equilibrium, the field lines of these magnetic fields need to travel around the torus in a helical fashion. Using streamlines the scientist can visualize the magnetic fields. The simulation was performed using the NIMROD code [24]. This data set has the unusual property that most streamlines are approximately closed and traverse the torus-shaped vector field domain repeatedly, which stresses the data cache. For the tests conducted here, we resampled onto 512 blocks with 1 million cells per block. Dense seeding is performed randomly on a small box inside the torus, while sparse seeding again randomly distributes seeds over the entire domain. Here, 2,500 seed points with an integration time of 20 were used for the small seed, and the large seed sets contain 10,000 seeds with an integration time of 5.

4.2.3 Thermal Hydraulics

The third data set results from a thermal hydraulics simulation. Here, 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. Nonoptimal 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 [25] on an unstructured grid comprised of 23 million hexahedral elements. Again, resampling was performed to a regular

TABLE 1
Results for the Parallelize-over-Seeds Algorithm

Test case	T_{total}		N_{load}		N_{purged}		$T_{\text{I/O}}$		T_{int}		R_{int}	
	MPI-only	MPI-hybrid	MPI-only	MPI-hybrid	MPI-only	MPI-hybrid	MPI-only	MPI-hybrid	MPI-only	MPI-hybrid	MPI-only	MPI-hybrid
A (LD)	12.5 s	12.5 s	5,568	2,185	1,832	0	2,504 s	1,007 s	81.9 s	98.3 s	1.55 %	6.16 %
A (LS)	41.2 s	19.3 s	8,405	3,525	4,573	175	3,670 s	1,565 s	92.2 s	103.1 s	1.16 %	4.18 %
A (SD)	62.1 s	14.6 s	6,699	2,914	2,995	2	3,142 s	1,272 s	38.2 s	43.1 s	0.71 %	2.31 %
A (SS)	63.4 s	19.7 s	9,787	4,151	5,986	493	4,159 s	1,783 s	49.2 s	54.8 s	0.61 %	2.17 %
T (LD)	27.3 s	9.3 s	6,544	2,194	2,710	0	2,276 s	805 s	55.9 s	62.3 s	1.60 %	5.20 %
T (LS)	43.5 s	12.3 s	4,535	2,755	1,130	2	1,816 s	1,085 s	21.6 s	24.0 s	0.39 %	1.53 %
T (SD)	53.6 s	22.0 s	12,715	5,153	8,875	1,313	4,834 s	1,966 s	23.4 s	27.0 s	0.34 %	0.96 %
T (SS)	59.9 s	24.8 s	9,776	5,449	5,969	1,610	3,836 s	2,111 s	17.7 s	19.8 s	0.23 %	0.62 %
F (LD)	43.2 s	13.0 s	6,621	1,461	2,781	0	2,848 s	691 s	196.7 s	214.1 s	3.55 %	12.85 %
F (LS)	360.3 s	36.1 s	44,864	9,173	41,024	5,333	17,238 s	3,534 s	140.5 s	158.4 s	0.30 %	3.42 %
F (SD)	381.6 s	39.9 s	74,487	5,345	70,647	1,726	29,917 s	3,534 s	196.9 s	235.6 s	0.40 %	4.62 %
F (SS)	717.1 s	125.1 s	117,672	30,524	113,842	26,684	44,912 s	11,166 s	129.4 s	146.6 s	0.14 %	0.91 %

For each test case X(YZ), X indicates data set (Astrophysics, Thermal hydraulics, Fusion), Y describes seed set size (Small or Large), and Z denotes seed distribution (Sparse or Dense). See Section 4.4 for a description of each test.

mesh of 512 blocks with 1 million cells each. 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, we place seeds 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 contained 1,500 seed points with an integration time of 12 units, and the large case consists of 6,000 seed points propagated for 3 time units.

4.3 Runtime Environment

All measurements discussed in the following were obtained 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 8 GB of memory (2 GB per core). Compute nodes are connected through HyperTransport for high performance, low-latency communication for MPI and SHMEM jobs. I/O is handled through the parallel Lustre file system that provides access to approximately 436 TB of user disk space.

The two hybrid algorithms discussed above were implemented into the VISIT [26], [27] visualization system, which is available on Franklin and routinely used by application scientists. Nonhybrid variants of both parallelize over blocks and parallelize over seeds are already implemented in recent VisIt releases and were instrumented to provide the measurements discussed below. Benchmarks were performed during full production use of the system to capture a real-world scenario. No special measures were taken to exclude operating system I/O caching. The default queuing system (QSUB) was used to distribute the nodes and cores as required; however, while this system can bind individual tasks to cores, we manually bind threads to individual cores to avoid bouncing, resulting in a notable performance increase. Here, bouncing refers to the phenomenon where a single thread is scheduled on multiple cores, resulting in cache thrashing.

Each benchmark run was performed using 128 cores (32 nodes). For the nonhybrid algorithm tests, 128 MPI tasks (one per core) are spawned. The hybrid tests were conducted such that the total number of worker threads over all nodes is 128. For example, 32 MPI tasks (one per node) are run, with each spawning four worker threads. Note that the additional I/O or communication thread running per MPI task in the hybrid approach is not counted in this scheme; here, we are purely focused on employing a constant number of worker threads performing actual integration work to determine the impact of hybrid parallelism. All tests were run for the nonhybrid case as well as for the hybrid implementation with four worker threads per MPI task.

4.4 Measurements

To obtain insight into the relative benefits of the hybrid parallel approach to streamline integration, we have obtained a number of timings and other statistics beyond the pure execution time T_{total} of the corresponding combination of algorithm and test case.

Every thread in the entire system, including workers, communication and I/O threads, keeps track of the time spent executing various functions using an event log consisting of pairs of timestamp and event identifier. Events include, for instance, start and end of integration for a particular streamline for worker threads, begin and end of an I/O operation for corresponding threads, and time spent performing communication using MPI calls. Timestamps are taken as wall time elapsed since the start of the MPI task. These event logs are carefully implemented to have negligible impact on the overall runtime behavior, and analyzed later to provide the summary statistics discussed in the following and represented in Table 1 and Table 2. Furthermore, they allow us to illustrate the distribution of work across tasks and threads using Gantt charts (see Figs. 7 and 8) to obtain a differentiated picture of the runtime distribution of work, communication, and I/O.

The pure integration time T_{int} represents the actual computational workload and is the sum of all times taken by the worker threads to integrate streamlines. 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 streamlines is identical in each case. Similarly, $T_{\text{I/O}}$ and T_{comm} accumulate the time spent doing I/O and communication, respectively. To obtain better insight into the role of the block cache in our benchmarks, we furthermore sum the numbers of

TABLE 2
Results for the Parallelize-over-Blocks Algorithm

Test case	T_{total}		N_{comm}		T_{comm}		T_{int}		R_{int}	
	MPI-only	MPI-hybrid	MPI-only	MPI-hybrid	MPI-only	MPI-hybrid	MPI-only	MPI-hybrid	MPI-only	MPI-hybrid
A (LD)	21.2 s	10.95 s	26.69 MB	11.96 MB	2,006 s	279 s	82.6 s	82.7 s	3.04 %	5.90 %
A (LS)	13.8 s	5.76 s	29.63 MB	11.08 MB	894 s	143 s	93.3 s	93.0 s	5.27 %	12.62 %
A (SD)	8.7 s	3.50 s	9.76 MB	4.82 MB	690 s	91 s	38.2 s	38.3 s	3.41 %	8.54 %
A (SS)	6.9 s	3.39 s	11.75 MB	5.18 MB	493 s	87 s	49.3 s	49.3 s	5.52 %	11.35 %
T (LD)	25.3 s	4.71 s	22.68 MB	12.3 MB	990 s	120 s	56.4 s	56.5 s	1.74 %	21.69 %
T (LS)	3.3 s	0.79 s	11.26 MB	3.75 MB	104 s	20 s	22.2 s	21.9 s	5.17 %	15.94 %
T (SD)	5.6 s	1.14 s	7.37 MB	4.3 MB	230 s	29 s	23.2 s	23.3 s	3.24 %	23.48 %
T (SS)	3.2 s	0.59 s	5.62 MB	2.86 MB	71 s	14 s	17.8 s	17.6 s	4.35 %	0.62 %
F (LD)	36.4 s	33.20 s	52.51 MB	26.92 MB	3,789 s	980 s	199.1 s	197.4 s	4.27 %	4.65 %
F (LS)	6.0 s	3.57 s	44.77 MB	20.11 MB	326 s	80 s	141.8 s	140.2 s	18.27 %	30.72 %
F (SD)	22.8 s	11.10 s	46.08 MB	23.63 MB	2,197 s	272 s	190.0 s	189.0 s	6.49 %	13.31 %
F (SS)	5.1 s	3.13 s	31.53 MB	15.77 MB	268 s	72 s	121.9 s	121.1 s	18.61 %	30.27 %

For each test case X(YZ), X indicates data set (Astrophysics, Thermal hydraulics, Fusion), Y describes seed set size (Small or Large), and Z denotes seed distribution (Sparse or Dense). See Section 4.4 for a description of each test.

blocks loaded and purged from the cache in N_{load} and N_{purged}. The amount of MPI communication between tasks in bytes is measured by N_{comm}. Finally, as a derived measure of efficiency, we examine the integration ratio R_{int} as the fraction of the total algorithm runtime that was used to integrate streamlines.

In total, 12 tests of the form X(YZ) were run per algorithm in hybrid and nonhybrid variants (see Table 1 and Table 2), 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 the seed set density (**S**parse or **D**ense).

5 Results and Analysis

We found that both parallelize-over-seeds and parallelize-over-blocks had improved performance in a hybrid setting. However, we found that the causes of their improved performance differed. We discuss the technique's results in the following sections.

5.1 Parallelization over Seeds

A hybrid *parallelize-over-seeds* algorithm has three fundamental advantages:

1. Both the MPI-hybrid and MPI-only algorithms keep a cache to store domains for later reuse, sparing additional I/O. For the hybrid case, the cache is shared, and hence, can be larger by the proportion of the threading factor. For our experiments, the cache was four times larger. The cache size could have been larger, but we wanted the scale factor to be the same between the MPI only and MPI hybrid.

   We can observe the effect of this factor by looking at the number of blocks purged due to a full cache. In Table 1, column N_{purged}, the number of purges in the MPI-hybrid case is significantly less than in the MPI-only case, which results in reduced I/O calls to reload data blocks.
2. For the MPI-only algorithm, when multiple MPI tasks on the same node need to access the same block, they must read the block redundantly from the disk. For the hybrid case, only a single read is required and the block is shared between all threads. This use case occurs frequently when the seed points are densely located in a small region.

   We can observe the effect of this factor by looking at the number of blocks loaded (see Table 1 column N_{load}), and the time it takes to load these blocks (see Table 1 column $T_{\text{I/O}}$). Looking at the A(LD) test in Table 1 column N_{load}, we can see that the MPI only had to load over double the number of blocks of data compared to the MPI hybrid. On average, the MPI-hybrid loads 64 percent less data with the low of 39 percent to the max of 93 percent less data loads. The graph of the number of blocks loaded is shown in Fig. 5. Note that this measure conflates with the purge metric. However, since the number of purges is small for some use cases, we can safely conclude that the remainder of the benefit, which is significant, is from this factor.
3. The time to calculate each streamline varies from streamline to streamline because of the data dependent nature of the advection step. We refer to the streamlines that take longer to execute as "slow streamlines." Since streamlines are permanently assigned to MPI tasks, the MPI tasks that get slow streamlines will take longer to execute. Toward the end of the calculation, the MPI tasks with slow streamlines will still be executing while the MPI tasks with "fast streamlines" will be done, meaning there is poor parallel efficiency. In Fig. 7, we can see that the MPI-hybrid implementation has less volatility between tasks. With the hybrid implementation, a larger number of streamlines are shared between the worker threads, which creates a more even distribution of slow streamlines. For example, if an MPI task in the MPI-only case received many slow streamlines there is only one thread to handle them, but in the hybrid case, there will be more worker threads to advance these slow streamlines. We were surprised by the significance of this advantage.

   We can measure this factor by dividing the time to execute for the average MPI task execution time by the slowest time to execute, which is a way to

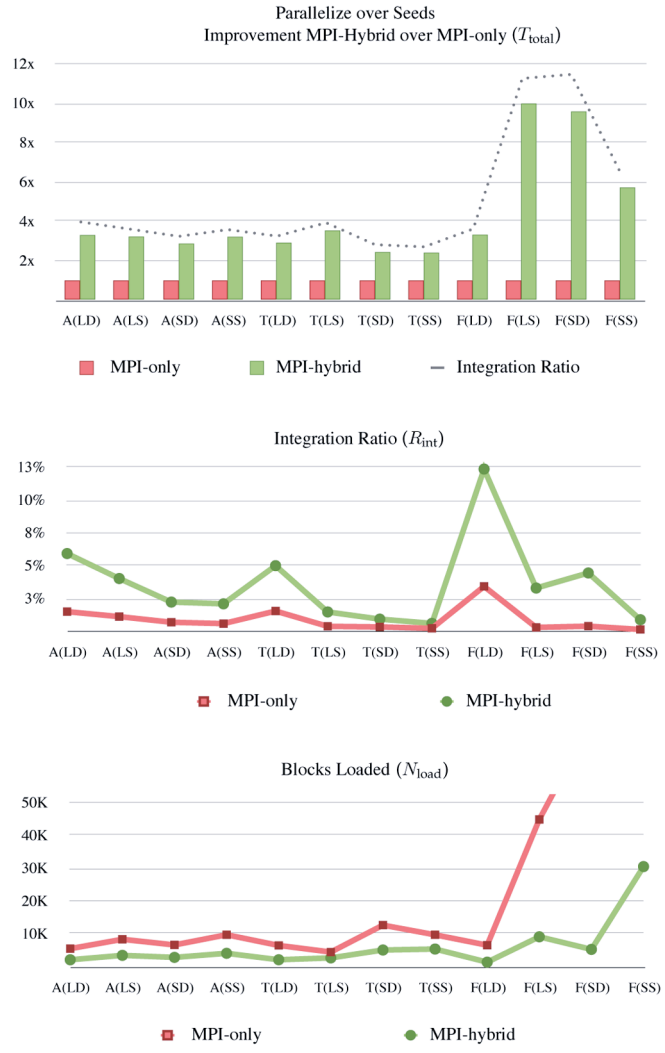


Fig. 5. A comparison of performance for hybrid and nonhybrid variants of the *parallelize-over-seeds* algorithm, in terms of the metrics from Section 4.4. The hybrid version is faster in all cases because fewer blocks are loaded, allowing for an increased integration ratio. See Section 5.1 for a detailed discussion.

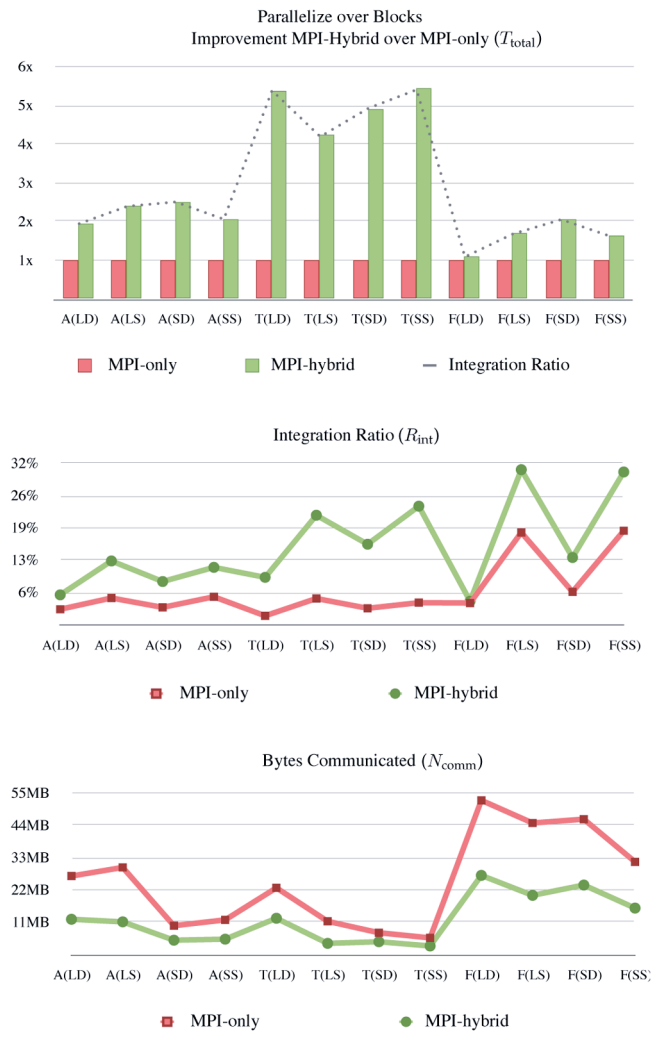


Fig. 6. Performance of hybrid versus nonhybrid variants of the *parallelize-over-blocks* algorithm as measured according to Section 4.4. The hybrid version has strongly decreased runtime for the same test because reduced communication enables a higher integration ratio is higher. See Section 5.2 for a detailed discussion.

measure parallel efficiency. These values can be seen in Table 1 column R_{int} and in Fig. 5.

One unexpected outcome was with I/O performance. Our original design had only a single I/O thread, assuming that I/O performance would be bandwidth bound, not latency bound. However, the blocks were relatively small (approximately four megabytes each) and so bandwidth was not an issue. However, we found that each read took approximately half a second, regardless of the number of requests per node. With the MPI-only implementation, there were up to four block reads at any time, meaning that four blocks could be read in a half second. With the original hybrid implementation, we were limited to one block read, meaning that only one block could be read in a half second. As a result, the MPI-only representation outperformed our original hybrid implementation in many of our tests. After making the switch to four I/O threads, the hybrid implementation was the clear winner in all tests. We believe that additional performance could be gained by adding more I/O threads. However, we did not pursue this approach because the MPI-only implementation could also use additional asynchronous I/O requests and we wanted to ensure the fairness of our comparisons. Of course, this phenomenon will decrease as the I/O performance approaches the bandwidth limit of the system.

5.2 Parallelization-over-Blocks

A hybrid parallelize-over-blocks algorithm has two fundamental advantages:

1. The hybrid parallelize-over-blocks algorithm's fundamental advantage is that it can improve parallel efficiency. Because data are partitioned over the MPI tasks, the only MPI task that can advance a given streamline is the MPI task that own the block the streamline resides in. When many streamlines traverse the same block, the corresponding MPI task becomes a bottleneck. With the hybrid algorithm, more cores can be used to relieve the bottleneck. Fig. 8 illustrates this point. The MPI-only implementation has only two MPI tasks working on the

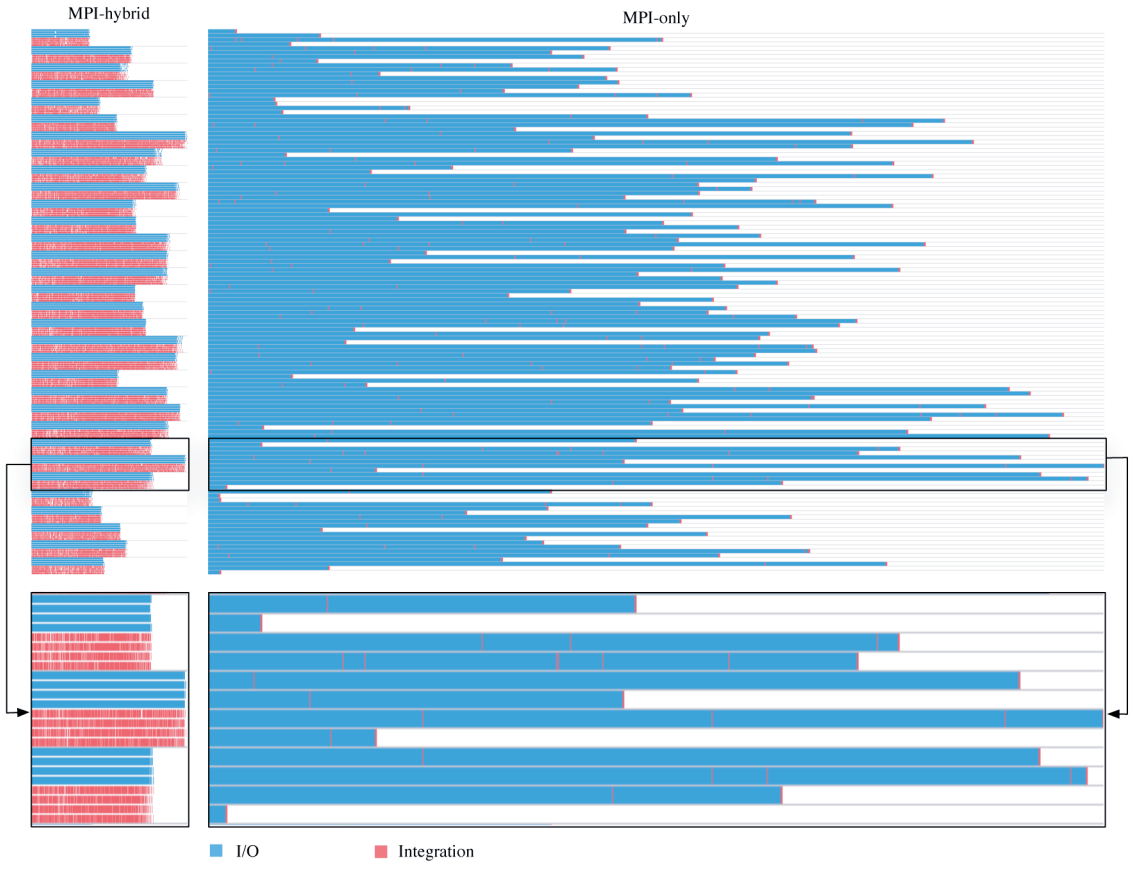


Fig. 7. A *parallelize-over-seeds* algorithm Gantt chart of the integration and I/O activity for the F(LD) test (cf. Section 4 and Table 1) comparing MPI-hybrid and MPI-only implementations. Each line represents one thread (left column) or task (right column). The hybrid approach outperforms the nonhybrid implementation by about $10\times$, since the 4 I/O threads in the hybrid model can feed new blocks to the 4 integration threads at an optimal rate, and each node uses I/O and computation maximally; however, work distribution between nodes is not optimally balanced. In the nonhybrid implementation, I/O wait time dominates computation by a large margin, due to redundant block reads, and work is distributed less evenly. This can be easily seen in the enlarge section of the Gantt chart. See Section 5.1 for more details.

longest streamlines, which translates to two cores. The hybrid implementation also has two MPI tasks working on these streamlines, but that equates to eight cores. The additional workers allow the hybrid case to finish more quickly. We can quantify this factor 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 hybrid implementation.

2. The communication cost of moving streamlines between MPI processes is much lower in most of the MPI-hybrid cases of the parallelize-over-blocks method, which can be seen in the communication time and data transmitted between tasks (see Fig. 6 and Table 2). The MPI-hybrid parallelize-over-blocks method has four times less number of MPI tasks than the MPI parallelize-over-blocks method and because each process holds four times more data it is less likely to have to send the streamline to another process which reduces the communication over head.

As with parallelize-over-seeds, we did not expect the I/O performance difference, which is reflected in the different length of the blue I/O time in Fig. 8. Since the loading of the data only occurred once at the start of the program it would not affect the integration time and was left to future work to achieve the same behavior as the MPI-only version.

6 Conclusion

In this paper, we examined the benefits of a hybrid parallel programming approach to distributed streamline integration. We investigated two parallel streamline algorithms, and measured the performance of a straightforward distributed implementation that assigns an MPI task to each core, making each core an isolated resource, against a hybrid parallelism approach that leverages the potential for local shared memory on multicore nodes. To fully explore the performance characteristics and differences of these parallelization approaches, we examined a wide variety of real-world scenarios in which streamlines must be computed.

Our findings indicate that the work shared between cores in the MPI-hybrid parallel implementation results in much improved performance and consumes less communication and I/O bandwidth than a traditional MPI-only implementation. We observed speedups ranging from two to ten (even though our threading factor was limited to four). Overall, we conclude that it is well worth using hybrid-parallel implementation strategies in the context of streamline-based algorithms.

In the future, we wish to examine specific applications of streamline-based visualization solutions in more detail, such as integral surfaces [2] and Lagrangian methods [1]. Furthermore, we are interested in observing scalability

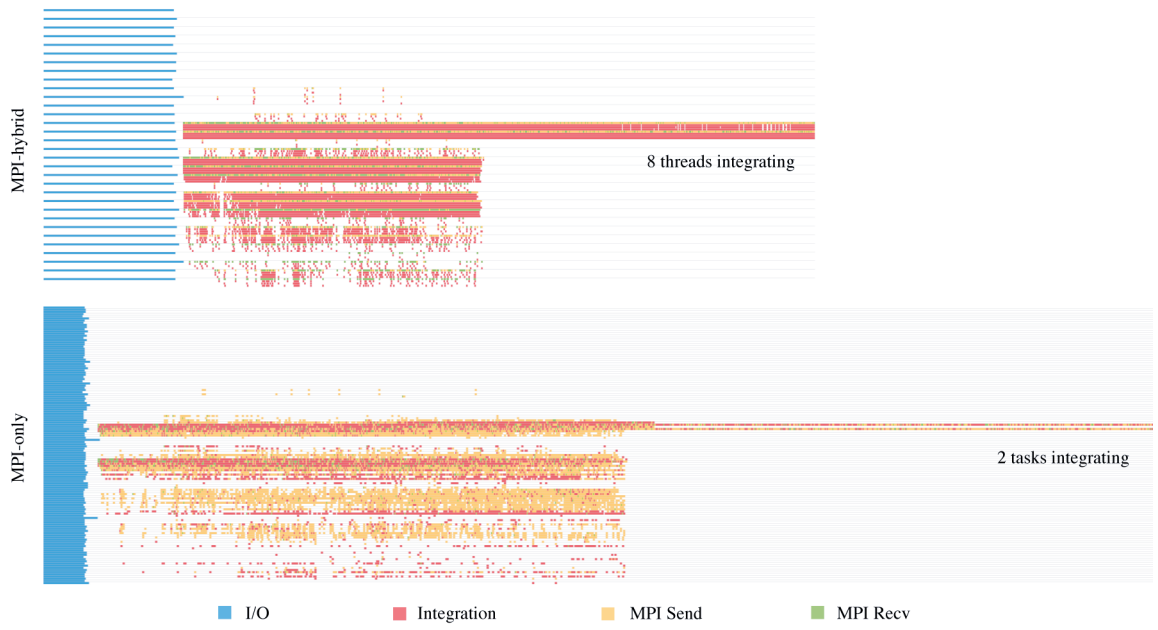


Fig. 8. A *parallelize-over-blocks* algorithm Gantt chart of the integration, I/O, MPI Send, and MPI Recv activity for the F(LS) test (cf. Section 4 and Table 2) comparing MPI-hybrid and MPI-only implementations. Each line represents one thread (top) or task (bottom). The comparison reveals that the 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. Toward the end, 8 threads are integrating in the hybrid approach, as opposed to only two tasks in the MPI-only model. See Section 5.2 for more details.

properties of corresponding hybrid parallel algorithms, as well as research adaptive parallel streamline algorithms [4].

Acknowledgments

This work was supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the US 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 work was supported in part by the US National Science Foundation (NSF) under contract IIS-0916289. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the US Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] C. Garth, F. Gerhardt, X. Tricoche, and H. Hagen, "Efficient Computation and Visualization of Coherent Structures in Fluid Flow Applications," *IEEE Trans. Visualization and Computer Graphics,* vol. 13, no. 6, pp. 1464-1471, Nov./Dec. 2007.

[2] H. Krishnan, C. Garth, and K.I. Joy, "Time and Streak Surfaces for Flow Visualization in Large Time-Varying Data Sets," *IEEE Trans. Visualization and Computer Graphics,* vol. 15, no. 6, pp. 1267-1274, Nov./Dec. 2009.

[3] T. McLoughlin, R.S. Laramee, R. Peikert, F.H. Post, and M. Chen, "Over Two Decades of Integration-Based, Geometric Flow Visualization," *Computer Graphics Forum,* vol. 29, no. 6, pp. 1807-1829, 2010.

[4] D. Pugmire, H. Childs, C. Garth, S. Ahern, and G. Weber, "Scalable Computation of Streamlines on Very Large Datasets," *Proc. Int'l Conf. Supercomputing,* 2009.

[5] D. Sujudi and R. Haimes, "Integration of Particles and Streamlines in a Spatially-Decomposed Computation," *Proc. IEEE Parallel Computational Fluid Dynamics Conf.,* 1996.

[6] D.A. Lane, "UFAT—A Particle Tracer for Time-Dependent Flow Fields," *Proc. IEEE Conf. Visualization,* pp. 257-264, 1994.

[7] B. Cabral and L.C. Leedom, "Highly Parallel Vector Visualization Using Line Integral Convolution," *Proc. SIAM Conf. Parallel Processing for Scientific Computing (PPSC '95),* pp. 802-807, 1995,

[8] S. Muraki, E.B. Lum, K.-L. Ma, M. Ogata, and X. Liu, "A PC Cluster System for Simultaneous Interactive Volumetric Modeling and Visualization," *Proc. the IEEE Symp. Parallel and Large-Data Visualization and Graphics (PVG '03),* p. 13, 2003,

[9] D. Ellsworth, B. Green, and P. Moran, "Interactive Terascale Particle Visualization," *Proc. IEEE Conf. Visualization,* pp. 353-360, 2004,

[10] S.-K. Ueng, C. Sikorski, and K.-L. Ma, "Out-of-Core Streamline Visualization on Large Unstructured Meshes," *IEEE Trans. Visualization and Computer Graphics,* vol. 3, no. 4, pp. 370-380, Oct.- Dec. 1997.

[11] R. Bruckschen, F. Kuester, B. Hamann, and K.I. Joy, "Real-Time Out-of-Core Visualization of Particle Traces," *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics (PVG '01),* pp. 45-50, 2001,

[12] H. Yu, C. Wang, and K.-L. Ma, "Parallel Hierarchical Visualization of Large Time-Varying 3D Vector Fields," *Proc. Int'l Conf. Supercomputing,* 2007.

[13] L. Chen and I. Fujishiro, "Optimizing Parallel Performance of Streamline Visualization for Large Distributed Flow Datasets," *Proc. IEEE VGTC Pacific Visualization Symp. '08,* pp. 87-94, 2008,

[14] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI—The Complete Reference: The MPI Core,* second ed. MIT Press, 1998.

[15] D.R. Butenhof, *Programming with POSIX Threads.* Addison-Wesley Longman Publishing, 1997.

[16] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP.* Morgan Kaufmann Publishers Inc., 2001.

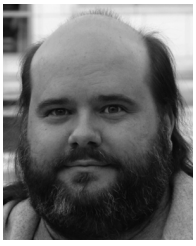
[17] *CUDA Programming Guide Version 2.3.,* NVIDIA Corporation, 2008.

[18] "High-Performance Fortran Language Specification, Version 1.0," Technical Report CRPC-TR92225, High Performance Fortran Forum, 1997.

[19] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC—Distributed Shared Memory Programming.* John Wiley & Sons, 2005.

[20] G. Hager, G. Jost, and R. Rabenseifner, "Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes," *Proc. Cray User Group Conf.,* 2009.

[21] D. Mallón, G. Taboada, C. Teijeiro, T.J., B. Fraguela, A. Gómez, R. Doallo, and J. Mourino, "Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures," *Proc. European PVM/MPI Users' Group Meeting (EuroPVM/MPI),* Sept. 2009.
[22] E. Endeve, C.Y. Cardall, R.D. Budiardja, and A. Mezzacappa, "Generation of Strong Magnetic Fields in Axisymmetry by the Stationary Accretion Shock Instability," *ArXiv E-Prints,* Nov. 2008.
[23] C.Y. Cardall, A.O. Razoumov, E. Endeve, E.J. Lentz, and A. Mezzacappa, "Toward Five-Dimensional Core-Collapse Supernova Simulations," *J. Physics: Conf. Series,* vol. 16, pp. 390-394, 2005.
[24] C. Sovinec, A. Glasser, T. Gianakon, D. Barnes, R. Nebel, S. Kruger, S. Plimpton, A. Tarditi, M. Chu, "Nonlinear Magnetohydrodynamics with High-Order Finite Elements," *J. Computational Physics,* vol. 195, pp. 355-386, 2004.
[25] P. Fischer, J. Lottes, D. Pointer, and A. Siegel, "Petascale Algorithms for Reactor Hydrodynamics," *J. Physics: Conf. Series,* vol. 125, pp. 1-5, 2008.
[26] "VisIt—Software that Delivers Parallel, Interactive Visualization," http://visit.llnl.gov/, 2011.
[27] H. Childs, E.S. Brugger, K.S. Bonnell, J.S. Meredith, M. Miller, B.J. Whitlock, and N. Max, "A Contract-Based System for Large Data Visualization," *Proc. IEEE Conf. Visualization,* pp. 190-198, 2005.



David Camp received the BS and MS degrees in computer science from the University of California, Davis. He is currently working toward the PhD degree at University of California, Davis. He is a member of the visualization group at Lawrence Berkeley National Laboratory. His research interests include scientific visualization, analysis methods for vector fields, parallel/scalable algorithms, and computer graphics. He is a member of the IEEE.

Christoph Garth received the PhD degree in computer science from the University of Kaiserslautern, in 2007. He is currently a postdoctoral researcher with the Institute for Data Analysis and Visualization at University of California, Davis. His research interests include scientific visualization, analysis methods for vector and tensor fields, topological methods, query-driven visualization, parallel/scalable algorithms, and computer graphics. He is a member of IEEE.

Hank Childs received the PhD degree in computer science from the University of California, Davis. He is a member of the visualization group at Lawrence Berkeley National Laboratory and a researcher in the Computer Science Department at University of California, Davis. His research interests include large data visualization, parallel visualization, and parallel algorithms.

Dave Pugmire received the PhD degree in computer science from the University of Utah. He is a computer scientist at Oak Ridge National Laboratory's Scientific Computing Group. His research interests include parallel scientific data analysis and visualization.

Kenneth I. Joy received the BA and MA degrees in mathematics from University of California, Los Angeles and the PhD degree from the University of Colorado, Boulder. He is a professor in the Department of Computer Science at University of California, Davis, and the director of the Institute for Data Analysis and Visualization. In this position, he leads research efforts in scientific visualization, geometric modeling, and computer graphics. He is a faculty computer scientist at Lawrence Berkeley National Laboratory and participating guest researcher at Lawrence Livermore National Laboratory. He is a member of the IEEE.